Security in Programming Languages

Dominik Schreiber
Universität Stuttgart
Universitätsstraße 38, 70569 Stuttgart
Email: mail@dominikschreiber.de

***Zusammenfassung*—In der vorliegenden Arbeit wird erörtert, welchen Beitrag die Wahl einer Programmiersprache zur Sicherheit der implementierten Anwendung leistet. Anhand zweier Studien werden Sprachen bezüglich der Sicherheit ihrer Anwendungen miteinander verglichen, und dabei beobachtet, dass keine Sprache grundsätzlich sichere Anwendungen garantieren kann. Daraufhin wird die Sprache Rust vorgestellt und als eine vielversprechende und sicherere Alternative zu anderen Systemsprachen bewertet. Im Anschluss wird ein sprachtheoretischer Ansatz (*Language-Theoretic Security*) thematisiert, der besonderes Augenmerk auf eine sichere und verifizierbare Verarbeitung von Eingaben legt, und dessen Realisierung in Form der Sprache *Crema* behandelt. Aus der Gesamtheit der Ergebnisse der Arbeit folgt, dass erhöhte Sicherheit im Bereich von Programmiersprachen stets mit Einschränkungen bei der Entwicklung einher geht.**

I. Einführung

Durch die stetige Vernetzung der Soft- und Hardware ist es für immer mehr Bereiche von Anwendungssoftware essentiell, dass für die Anwendung eine hohe Informationssicherheit gewährleistet ist. Doch welche Rolle spielt hier die Wahl der Programmiersprache? Kann mit optimaler Wahl der Programmiersprache so etwas wie eine „Garantie“ erkauft werden, dass die entstehende Anwendung durchgehend sicher ist? Oder hängt die Sicherheit nicht von Programmiersprachen, sondern einzig von der generellen Code-Qualität ab?

Falls ein Programm fertig implementiert ist, möchte man naheliegenderweise überprüfen, ob die Anwendung tatsächlich sicher ist. Anstelle von Tests (wie Penetrationstests) kann auch eine formale Verifikation des Programmes erfolgen [1]. Doch inwieweit ist eine solche Verifikation für gängige Programmiersprachen möglich? Gibt es so etwas wie „Beweisbare Sicherheit“ für Programme, und wenn ja, unter welchen Bedingungen?

Die genannten Fragen werden in der vorliegenden Seminararbeit anhand von aktuellen Studien und Forschungsarbeiten erörtert. Die Programmiersprachen Rust und Crema werden in den beiden Teilen jeweils als wichtige Fallbeispiele dienen.

II. Sicherheit gängiger Programmiersprachen

A. Veracode – State of Software Security

In der Studie *Veracode: State of Software Security. Focus on Application Development* wurden verschiedene beliebte Programmiersprachen und Plattformen auf Sicherheitslücken verschiedener Bereiche untersucht [2]. Berücksichtigt wurden die Sprachen Java, .NET, C/C++, PHP, ASP und Coldfusion[1] sowie die Plattformen Android und iOS.

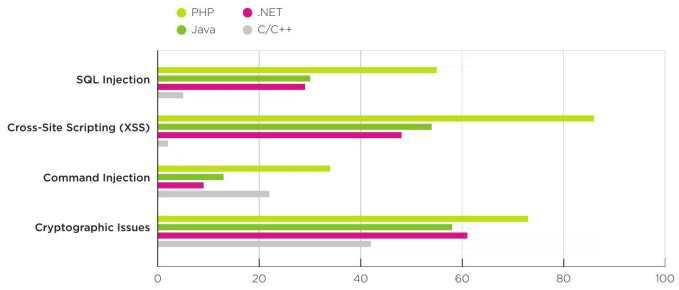

Abbildung 1. Prozentualer Anteil gefährdeter Anwendungen in Bezug auf spezifische Sicherheitslücken [2, S. 8]

Unter den Ergebnissen findet sich unter anderem die Erkenntnis, dass zwischen den Sprachen erhebliche Unterschiede in der Zahl der Anwendungen bestehen, die von zentralen Gefährdungen betroffen sind. So wurde gezeigt, dass unter den untersuchten Anwendungen, die mit Coldfusion programmiert wurden, über 60% anfällig gegenüber *SQL Injection*[2] sind, während dagegen unter 30% der mit .NET programmierten Anwendungen diese Schwäche zeigen (siehe Abb. 1).

Eine weitere wichtige Erkenntnis liegt darin, dass einige Sprachen von Grund auf so entworfen sind, dass sie gewisse Gefährdungsklassen vollständig (oder fast vollständig) vermeiden. Ein zentrales Beispiel ist hier die manuelle Speicherverwaltung, die in C/C++ zu hoher Effizienz, jedoch auch zu schwerwiegenden Gefährdungen führen kann.

Als eine der größten sicherheitstechnischen Gefährdungen gilt der *Buffer Overflow*, also Pufferüberlauf. Ihm zugrunde liegt die Eigenschaft einiger Sprachen, dass Speichersicherheit nicht gewährleistet ist, und damit der Entwickler selbst auf Konsistenz der Speicherung von Daten achten muss.

```
char buffer[16];
cin >> buffer;
```

In diesem Fall wird ein Puffer der Länge 16 allokiert, etwa um eine Nutzereingabe einzulesen. Falls nun die Nutzereingabe

[1]Eine Middleware für Webanwendungen von Adobe Systems; siehe https://www.adobe.com/products/coldfusion-family.html

[2]SQL-Einschleusung: Der Angreifer gibt eine Eingabe ins System, die als SQL-Befehl interpretiert wird, und kann damit unter Umständen Daten lesen, verändern, und/oder löschen. [3, S.1]

(hier angegeben als `cin`-Stream) die Länge 16 übersteigt, so wird nicht etwa ein Fehler bekanntgegeben, sondern der Speicher hinter dem Puffer überschrieben; der Puffer „läuft über“.

Sprachen, die einen solchen Überlauf zulassen, laufen Gefahr, dass ein Angreifer den Speicher des Programms manipuliert, was zum Absturz oder sogar zur bösartigen Veränderung der Funktionsweise führen kann (etwa wenn die Rückkehradresse mit etwas Anderem überschrieben wird). Sprachen, die die Speicherverwaltung hingegen intern verwalten und dem Programmierer abstrahierte Schnittstellen zur Verfügung stellen (etwa die `Array`-Klasse von Java), vermeiden Anfälligkeiten in diese Richtung, und sind daher in dieser Beziehung meist inhärent sicherer.

B. Whitehat Security Report

	.NET	Java	Perl	PHP
Cross-Site Scripting	35	57	67	56
Information Leakage	44	15	11	17
Content Spoofing	5	8	6	7
SQL Injection	6	1	3	6
Cross-Site Request Forgery	2	4	4	2
Insufficient Transport Layer Protection	0.9	1	0.3	4
Abuse of Functionality	0.3	0.9	0.5	0.2

Abbildung 2. Prozentualer Anteil gefährdeter Anwendungen in Bezug auf verschiedene Lücken [4, S. 12]

Gegenüber *Veracode* suggeriert die Studie *WhiteHat Security – 2014 Website Security Statistics Report* etwas andere Ergebnisse. Es wurden die sechs häufigsten Programmiersprachen im Web-Anwendungsbereich analysiert; namentlich .NET, Java, PHP, ASP, Perl und Coldfusion [4]. Hier wurde hervorgehoben, dass keine großen Unterschiede im durchschnittlichen Anteil gefährdeter Anwendungen zwischen den Sprachen bestehen, aber die Anwendungen verschiedener Sprachen dafür Lücken an verschiedenen Stellen aufweisen. Abb. 2 zeigt beispielsweise, dass sich .NET-Anwendungen gegenüber Cross-Site Scripting Attacken am robustesten zeigen, dafür jedoch im Bereich der Information Leakage die mit Abstand höchsten Lücken aufweisen.

Einschränkend muss hier berücksichtigt werden, dass bei dieser Studie Gefährdungen, die auf der Speicher-Unsicherheit von Sprachen beruhen, wie der bereits genannte Buffer Overflow, nicht beachtet wurden. Das gezeichnete Bild der Sprachen weist damit signifikante Lücken auf.

Zudem stellt *WhiteHat* fest, dass aus hoher Popularität und Komplexität einer Sprache wie Java auch eine höhere Angriffsoberfläche folgt, sowie, dass viele Klassen von Gefährdungen sprach-unabhängig auftreten.

C. Zwischenfazit

Die Studien zeigen, dass keine der aktuell genutzten Programmiersprachen im Anwendungsbereich annähernd frei von Sicherheitslücken ist. Stattdessen treten abhängig von der Programmiersprache an ganz verschiedenen Stellen Gefährdungen auf. Die Wahl der Programmiersprache kann also durchaus einen Ausschlag in Bezug auf die spezifische Sicherheit der Anwendung geben, jedoch nie absolute Sicherheit gewähren – an dieser Stelle ist vor allem der Entwickler in der Bringschuld, umsichtig auf Fehler in der Programmierung zu achten. Wie in den Studien betrachtet, verleiten unterschiedliche Programmiersprachen in der Tendenz zu unterschiedlichen Fehlern. Dies lässt den Wunsch aufkommen, eine Sprache zu verwenden, die mögliche Sicherheitslücken, die durch Programmierfehler entstehen, so weit wie möglich minimiert. Unter diesem Gesichtspunkt soll nun die Sprache Rust betrachtet werden.

III. Die Sprache Rust

Rust ist eine moderne Programmiersprache, die von der Mozilla Foundation seit 2010 entwickelt wird. Unter dem Gesichtspunkt von Security ist diese Sprache deshalb interessant, da die Speichersicherheit und die Vermeidung weiterer häufiger Bugs wie *Data Races*[3] eines der zentralen Ziele der Entwickler darstellt. Zudem soll Rust einfache und korrekte Nebenläufigkeit garantieren und außerdem sehr schnell in der Ausführung sein [5]. Realisiert werden diese Ziele hauptsächlich durch eine spezielle Semantik, die einige Restriktionen mit sich bringt, und daher zunächst als etwas umständlich für die Programmierung eingestuft werden mag. Im Folgenden werden einige der Grundlagen der Programmiersprache Rust erläutert.[4]

A. Variablen

(Listing 1, Z. 1-2) Variablen werden durch das Schlüsselwort `let` deklariert. Die Angabe eines Typen ist bei sofortiger Initialisierung nicht zwingend notwendig; stattdessen findet zur Kompilierzeit eine Typinferenz statt.

Im Gegenzug zu Sprachen wie C oder Java, bei welchen Variablen veränderlich sind, außer durch Schlüsselworte wie `final`, sind Variablen in Rust zunächst standardmäßig unveränderlich, und müssen explizit mit dem Schlüsselwort `mut` deklariert werden, um veränderlich zu sein.

B. Structs

Die Definition eines `struct Circle` könnte in etwa so aussehen wie in Listing 1, Z. 6-10.

Die Methoden eines `struct` werden separat davon definiert (Listing 1, Z. 12-18); innerhalb der `area`-Methode ist zu sehen, dass keine *Anweisung* für die Rückgabe notwendig ist (durch `return`-Schlüsselwort), sondern der *Ausdruck* des Werts genügt. Rust ist eine Ausdruck-basierte Sprache.

C. Zuweisungen

(Listing 1, Z. 22-24) Es wird ein Objekt des Typen `Circle` deklariert und initialisiert. Daraufhin lassen sich die Attribute und Methoden des Objekts gewöhnlich aufrufen.

In Listing 1, Z. 26-27 ist eine spezielle Eigenheit von Rust zu sehen: Die zweite Zeile dieses Blocks verursacht

[3] siehe III.C „Zuweisungen“

[4] Die Informationen zu Syntax und Semantik der Sprache sowie deren Hintergründe wurden der offiziellen Dokumentation entnommen [6].

```
1  let x : f64 = 0.0;
2  let mut y = 0.0;
3
4  // ...
5
6  struct Circle {
7          x: f64,
8          y: f64,
9          radius: f64,
10 }
11
12 impl Circle {
13         fn area(&self) -> f64 {
14                 std::f64::consts::PI *
15                         (self.radius *
16                         self.radius)
17         }
18 }
19
20 // ...
21
22 let mut c = Circle { x: x, y: y,
23         radius: 5.0 };
24 println!("Area: {}", c.area());
25
26 let c2 = c;
27 println!("Area: {}", c.area()); // ERROR
28
29 let c3 = &c2;
```

Listing 1: Rust-Beispielcode

einen Kompilierfehler mit der Beschreibung `use of moved value: 'c'`. Durch die erste Zeile wurde das Objekt nicht etwa von `c` nach `c2` kopiert, sondern vielmehr *verschoben* – die Variable `c` ist von nun an unbenutzbar.

Dieser speziellen Mechanik liegt das Ziel zugrunde, *Data Races* zu verhindern. Ein solches liegt dann vor, wenn für ein gegebenes Objekt zur selben Zeit zwei verschiedene Parteien einen Lese– und einen Schreibzugriff, beziehungsweise mehr als einen Schreibzugriff besitzen. In Rust wird daher zu jedem Zeitpunkt sichergestellt, dass entweder beliebig viele *unveränderliche* Referenzen auf ein Objekt weisen, oder aber höchstens eine einzige *veränderliche*.

Unveränderliche Referenzen werden gemäß Listing 1, Z. 29 definiert.

Sowohl `c2` als auch `c3` haben nun einen „Lesezugriff“, können also Eigenschaften und Methoden des Objekts aufrufen, aber nicht verändern. Um eine veränderliche Referenz auf ein Objekt zu erzeugen, kann das Schlüsselwort `mut` ergänzt werden – allerdings ist dann zu beachten, dass die ursprüngliche Variable, deren Objekt referenziert wird, nicht mehr benutzbar ist, da ja nun ein neuer Schreibzugriff vorliegt.

D. Einsatz und Bewertung

Rust ist eine Sprache, die sehr reich an verschiedenen Features ist. Mehrere Paradigmen wie Funktionale und Objektorientierte Programmierung ermöglichen elegante und zugleich effiziente Implementierungen. Zudem werden durch die restriktive Semantik sehr viele potentielle Fehler bereits vom Compiler bemerkt, und mit einer aussagekräftigen Fehlermeldung dem Programmierer mitgeteilt. Auch den Ursachen vieler Sicherheitslücken, wie der insgeheimen Manipulation von Variablen durch Seiteneffekte, wird damit entgegengewirkt. Die Sprache wirkt indes sehr durchdacht entworfen, und ermöglicht sicherlich, mit einer ähnlichen Effizienz wie C++ Anwendungen zu schreiben, die jedoch im Gesamten sicherer gegenüber Speicher– und anderen Programmierfehlern sind.

Jedoch erfordert die Sprache eine gründliche Einarbeitung, um damit gut programmieren zu können (etwa mithilfe der ausführlichen und verständlich formulierten Dokumentation [6]). Viele der Anwendungsentwickler, die Systemsprachen wie C++ gewohnt sind, werden von sich aus nicht die für Rust notwendige Umgewöhnung des Programmierstils in Kauf nehmen.

Rust wird bisher in einigen ausgewählten Projekten verwendet, etwa der serverseitigen Speicherverwaltung von Dropbox [7], dem *Servo*-Projekt für eine parallele Browser-Engine [8], oder dem unixoiden Microkernel-Betriebssystem *Redox* [9]. Vor allem aber nutzt Mozilla die eigens entwickelte Sprache für den Freien Internetbrowser Firefox, dessen Module nach und nach auf Rust portiert werden. Die bald erscheinende Firefox-Version 48 soll ersten Rust-Code enthalten; es handelt sich um eine Multimedia-Schnittstelle, die gerade im Aspekt der Informationssicherheit als kritisch zu betrachten ist [10]. Da Multimedia-Daten Schadcode enthalten können, der schlimmstenfalls ausgeführt wird, was bereits zu schwerwiegenden Gefährdungen auf Android-Geräten geführt hat [10], kann sich hier Rust von vornherein als eine sichere Programmiersprache beweisen.

Aus Gründen erhöhter Sicherheit sowie auch vorteilhafter Platformunabhängigkeit haben sich zudem Projekte gebildet, Software wie die GNU Coreutils oder ausgewählte Module des Linux-Kernels auf Rust zu portieren [11]. Mit dem Hintergrund, dass Version 1.0 erst vor einem Jahr veröffentlicht wurde [12], deutet dies auf eine vielversprechende Zukunft der Sprache hin, was sich im Umkehrschluss auch positiv auf die Sicherheit der Anwendungen auswirken dürfte – selbst wenn hinter der Nutzung von Rust mitunter andere Gründe stehen als erhöhte Sicherheit.

IV. Beweisbare Sicherheit – Sprachtheoretischer Ansatz

Um die Sicherheit von Programmen zu bestimmen, können Methoden der (formalen) Verifikation eingesetzt werden. Anders als beim Testen, bei dem sinnvolle Eingabedaten identifiziert werden und das Programm dann darauf ausgeführt wird, kann eine Verifikation für gewisse Codeabschnitte tatsächlich zuverlässig bestätigen, dass es sich um korrekten Code handelt, der genau das (und nur das) tut, was die Intention des Entwicklers war, und somit auch keine ungewollten Gefährdungen mit sich bringt.

Diese Methode ist dennoch mit Vorsicht zu genießen, da im Allgemeinen eine in einer gängigen Programmiersprache programmierte Anwendung nicht vollständig verifiziert werden kann. Der Grund dieser Tatsache erschließt sich durch die

Theorie der Formalen Sprachen, in welche daher im Folgenden kurz eingeführt wird.

A. Einführung in Formale Sprachen

Eine *Formale Sprache* ist definiert als eine Menge von Wörtern über einem gegebenen Alphabet (der Menge von Buchstaben) Σ. Gilt beispielsweise $\Sigma = \{a, b\}$, so sind $L_1 = \{a^n \mid n \in N\}$ oder $L_2 = \{a^p \mid p\ prim\}$ gültige Sprachen. Unmittelbar lässt sich erkennen, dass L_1 etwa eine sehr einfache Sprache ist – die Entscheidung über die Mitgliedschaft einer beliebigen Zeichenkette w in L_1 besteht nur darin, herauszufinden, ob w ausschließlich a enthält ($\Rightarrow w \in L_1$) oder nicht ($\Rightarrow w \notin L_1$). Für L_2 ist das nicht der Fall; hier muss die entscheidende Instanz über Mittel und Wege verfügen, die Anzahl der a abzuzählen und zu überprüfen, ob es sich um eine Primzahl handelt. L_2 ist also intuitiv *schwerer* als L_1.

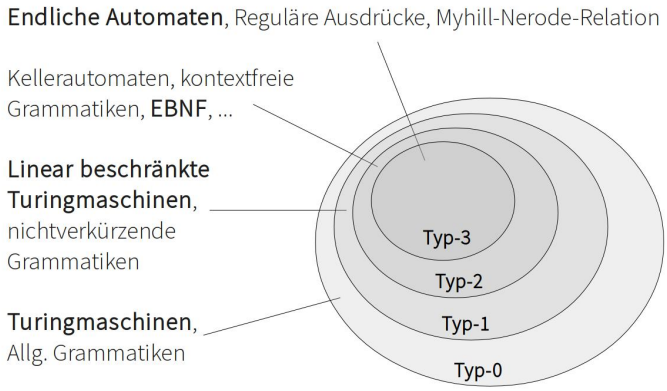

Abbildung 3. Die Chomsky-Sprachhierarchie, mit einigen charakterisierenden Instanzen

Diese Einordnung von Sprachen in Stufen der Schwierigkeit wird durch die *Sprachhierarchie* von Noam Chomsky vorgenommen [13]. Es existieren verschiedene Typen von Sprachen, die von Typ-0 bis Typ-3 benannt werden. Jede Typklasse enthält dabei alle Klassen, deren Index über dem eigenen liegen. Typ-0 enthält also auch die Typen 1 bis 3, während Typ-3 der kleinste Typ ist. Typ-3, auch als die *regulären Sprachen* bezeichnet, enthält die am einfachsten zu erzeugenden und erkennenden Sprachen (wie etwa L_1); diese Sprachen sind in ihrer Mächtigkeit jedoch stark eingeschränkt. Beim Herabsteigen der Typhierarchie werden die enthaltenen Sprachen zunehmend komplexer, und mächtiger. Typ-0-Sprachen schließlich sind im Allgemeinen nicht mehr entscheidbar; das bedeutet, das für eine gegebene Typ-0-Sprache L und eine Zeichenkette w nicht mehr in endlicher Zeit entschieden werden kann, ob $w \in L$.

Diese Typ-0-Sprachen werden in der Theorie erkannt von allgemeinen *Turingmaschinen*; jenen Konstrukten, die genau die Berechnungen in endlicher Zeit durchführen können, die im intuitiven Sinne berechenbar sind[5].

[5]Churchsche These; vgl. [14]

Diese Theorie überträgt sich unmittelbar auf Programmiersprachen. Dabei können nicht nur die Programmiersprachen selbst als formale Sprachen angesehen werden, sondern auch die entstehenden Programme als *Automaten*, also als Konstrukte, die gewisse *Wörter* (Eingabedaten) erhalten, und auf dieser Grundlage Entscheidungen treffen (Ausführung des Programms). Die potenzielle Komplexität und Mächtigkeit der Anwendung ergibt sich dabei aus den Eigenschaften der Programmiersprache. Und da von einer Programmiersprache im Normalfall erwartet wird, dass mit ihr jede erdenkliche Berechnung durchgeführt werden kann (dabei spricht man von der *Turing-Mächtigkeit* einer Sprache), muss es sich bei den Programmen um ebenso komplexe Konstrukte handeln wie allgemeine Turingmaschinen.

Dies ist in mehrerlei Hinsicht sehr hinderlich für die Verifikation von Programmen.

Zum einen wächst mit größerem Sprachentyp auch die mögliche Zahl der *Zustände* des Automaten. Die sehr eingeschränkten Endlichen Automaten für Typ-3-Sprachen etwa haben unabhängig von der Eingabelänge stets eine konstante Anzahl von Zuständen; allgemeine Turingmaschinen dagegen können selbst bei kleinem Programm und kleinen Eingaben einen sehr großen, i. A. unendlich großen Zustandsraum nutzen. Eine Verifikation, die für bestimmte Eingaben den Zustandsraum (hier: die Speicherbelegung aller Variablen) systematisch durchsucht, gerät bei Turing-mächtigen Sprachen also sehr schnell an praktische Grenzen.

Doch auch von theoretischer Seite, ohne Betrachtung der Komplexität der Verifikation, ergibt sich ein Problem. Das *allgemeine Halteproblem* [15] sagt aus, dass für eine Turingmaschine M im Allgemeinen nicht entschieden werden kann, ob sie auf eine bestimmte, feste Eingabe in endlicher Zeit anhält. Angewandt auf die Betrachtung von Programmen als Automaten folgt daraus unmittelbar, dass für ein beliebig komplexes Programm einer Turing-mächtigen Sprache nicht entschieden werden kann, ob das Programm überhaupt terminiert. Da Endlosschleifen insbesondere zu unerwünschtem Verhalten zählen können, weiß man durch das Halteproblem, **dass ein beliebiges Programm grundsätzlich nicht formal verifizierbar ist**.

B. Language-Theoretic Security

Eine Forschungsrichtung mit dem Namen *Language-Theoretic Security* (LangSec) geht davon aus, dass Sicherheitslücken von Anwendungen hauptsächlich aus einer unvorsichtigen und daher inkorrekten oder unvollständigen Verarbeitung von Nutzereingaben folgen [16]. Diese Eingaben, so LangSec, seien selbst als Formale Sprachen zu betrachten und zu spezifizieren. Ist dies gelungen, so kann das Programmmodul, das die entsprechende Eingabe verarbeitet, einem Automaten nachempfunden werden, der genau diese Sprache erkennt.

Der zentrale Mehrwert dieser Methode liegt darin, dass die Mächtigkeit des *erkennenden* Programmmoduls minimal gewählt wird. Ein Beispiel: Soll eine Nutzereingabe einem Vornamen entsprechen, so kann diese durch einen bestimmten Regulären Ausdruck spezifiziert werden, zum Beispiel

`[A-Z][a-z]+` (Zusätze wie Umlaute oder Bindestriche können problemlos hinzugefügt werden). Da Reguläre Ausdrücke stets regulären Sprachen entsprechen, darf das Programmmodul, das diese Eingabe entgegennimmt, *nicht mächtiger sein* als ein Endlicher (Typ-3) Automat. Ist dies garantiert, so kann das Programmmodul sehr einfach verifiziert werden – die Zustandszahl des Programms ist sehr klein (konstant, unabhängig von der Eingabelänge), und die Äquivalenz eines Regulären Ausdrucks zu einem Endlichen Automaten lässt sich konstruktiv bestimmen.

Zusammengefasst setzt also LangSec den ernüchternden Ergebnissen aus der Theorie der formalen Sprachen die Idee entgegen, die Mächtigkeit von den Modulen, die Eingaben verarbeiten, bewusst zu minimieren. Durch die fehlende Turing-Mächtigkeit können die Module dann formal verifiziert werden – umso besser, je höher die Sprachklasse ist.

C. Die Sprache Crema

Eine konkrete Anwendung dieser Idee stellt die Sprache Crema dar [17]. Sie ist so konzipiert, dass ihr Berechnungsmodell genau einem Typ-1 Automaten entspricht. Damit ist gewährleistet, dass jedes in Crema geschriebene Programm formal verifizierbar ist, und die Zustandzahl der Programme nicht derart „explodiert“ wie bei Turing-mächtigen Sprachen. Erreicht wird dies durch sehr eingeschränkte Kontrollstrukturen. Die einzige in Crema erlaubte Schleife ist eine `foreach`-Schleife wie folgt:

```
foreach(crema_seq(0, len) as i) {
    // Berechnung, z.B. mit Wert i
}
```

Es wird über eine *im Voraus bestimmte und dann unveränderliche* Liste iteriert. Damit ist die Anzahl der Durchläufe der Schleife bereits im Voraus bekannt.

Ein Konstrukt wie das Folgende für die Iteration über eine Eingabe ist in Sprachen wie C nicht unüblich:

```
for (;;) {
    if (input[i] == '\0') {
        break;
    }
    // ...
    i++;
}
```

Es wird zur Laufzeit anhand der Eigenschaften einer Eingabe bestimmt, wie oft die Schleife ausgeführt wird – in diesem Fall, an welcher Stelle das Endzeichen `'\0'` auftritt.

Die potenzielle Zahl der möglichen Programmzustände steigt dadurch stark an. Eine gängige Form der Verifikation ist etwa, durch *Symbolische Ausführung*[6] alle möglichen *Zustands-Verzweigungen* des Programmes auf Fehler zu überprüfen [17, II. 2)]. Der Programmzustand ist gegeben durch die Menge der Werte aller Variablen; eine Verzweigung tritt immer bei der Abwägung einer *Bedingung* auf (also bei Schlüsselworten wie `if` und `else`, aber auch `while` und `for`, solange die zu überprüfende Bedingung nicht konstant `true` oder `false` ist). Für das gegebene Beispiel in C eröffnet ein derartiger Verifikator also bei jeder Ausführung von Z. 2 eine neue Verzweigung. Enthält der Code nun eine weitere bedingte Ausführung in Z. 5, die den Programmzustand (also den Wert zumindest einer Variablen) verändert, so steigt die Zustandszahl, die vom Verifikator durchlaufen wird, exponentiell an, weil für jeden aktuellen Zustand im Schleifendurchlauf i zwei neue Zustände hinzukommen (*Zustandsexplosion*). Auch schon bei den einfachsten Konstrukten kann hier somit eine Prüfung der möglichen Programmzustände beliebig aufwendig werden. Die herkömmliche Lösung ist hier, bei der Verifikation eine maximale Tiefe anzugeben, bis zu der die Ausführung überprüft werden soll [17, II. 2)]. Damit bleibt die Prüfung letztlich unvollständig.

Im Vergleich dazu eine mögliche Implementierung in Crema:

```
int itr[] = crema_seq(0, str_len(input))
foreach(itr as i) {
    // ...
}
```

Für diese Implementierung ist im Voraus bekannt, wie oft die Schleife ausgeführt wird. Das Wachstum der durchlaufenen Zustände steigt also in einem Rahmen an, der vorhersagbar bleibt. Auch, wenn sich in Z. 3 eine weitere bedingte Ausführung befindet, und damit die Zustandszahl ähnlich schnell ansteigt wie in obigem C-Beispiel, so ist die maximale Tiefe bekannt, und die Prüfung kann potentiell vollständig durchgeführt werden. Dies ist genau analog zur Gegenüberstellung der Zustandszahl einer Turingmaschine mit linear beschränktem Band (Typ-1) und einer allgemeinen Turingmaschine mit unendlichem Band (Typ-0, Turing-mächtig): die Zustände der zuerst genannten bleiben endlich, und können daher vollständig traversiert werden, während bei letzterer durch unkontrollierte Endlosschleifen unendlich viele Zustände durchlaufen werden können.

Die Sprache Crema zwingt also den Entwickler zu simplen Programmiertechniken mit eingeschränkter Mächtigkeit, aber dafür sehr guten Möglichkeiten zur Verifikation.

Bewertung: Bei Crema handelt es sich um die Umsetzung einer vielversprechenden Idee, da bei der Entwicklung von Programmmodulen in Crema durch die erzwungenermaßen beschränkten Kontrollstrukturen der Programmfluss vorhersehbarer und daher sicherer wird. Die Möglichkeit der vollständigen Verifikation des Programmmoduls ist alles andere als selbstverständlich, und kann für Prozesse mit hoher Sicherheitsanforderung ein sehr nützliches Werkzeug darstellen.

Jedoch hat diese Vorgehensweise auch einen Preis, denn es muss das Programmmodul in Crema implementiert, integriert und dann verifiziert werden, was einen erheblichen Mehraufwand darstellen dürfte, der nur bei triftigen Gründen in Kauf

[6] Durchlaufen des Programmes mit allgemeinen Variablen anstatt konkreten Werten als Eingabewerte; vgl. [17, II. 2)]

genommen werden wird. Zudem ist Crema die Realisierung einer Sprache mit Mächtigkeit eines Typ-1 Automaten; doch viele Eingaben sind sogar Typ-2 oder Typ-3. Um der Idee von *LangSec* gerecht zu werden, müssen auch für diese Sprachtypen „Programmiersprachen" verwendet werden, deren Mächtigkeit *genau* dem jeweiligen Typen entspricht. Für Typ-3 Sprachen können Eingaben durch Reguläre Ausdrücke und für Typ-2 Sprachen durch EBNF-Ausdrücke (*Erweiterte Backus-Naur-Form* [18]) verifiziert werden; doch hier müssen wieder, genau wie bei einer Einbindung eines Crema-Moduls, die Schnittstellen äußerst umsichtig entworfen und überprüft werden. Denn ein verifiziertes Modul ist nur von geringem Nutzen, wenn bei der Übergabe der Eingabe an das Modul bereits ein Fehler geschieht.

Insgesamt stellt der Mehraufwand dieser Entwicklungsmethode durch die Identifikation des Sprachtypen, die Einbindung von Crema, und die anschließende Verifikation eine große Hürde dar, die Ansätze wie bei Crema – zumindest im Moment – von einer Verbreitung im Anwendungsbereich abhält.

Die Frage nach der *Beweisbarkeit* der korrekten Ausführung eines Programms kann indes wie folgt beantwortet werden: Für eine Unterklasse von Programmiersprachen, die jedoch in ihrer Mächtigkeit begrenzt sind, existieren Methoden, die darin geschriebenen Anwendungen vollständig zu verifizieren. Dies gilt aber nur für die Ausführung des Codes selbst, und insbesondere für die korrekte Verarbeitung von Nutzereingaben ohne unerwünschte Nebeneffekte. Es können nach wie vor Sicherheitslücken bestehen, etwa durch Seitenkanal-Angriffe, die auf Emissionen des Systems, wie etwa Ausführungszeit oder Energieverbrauch, zurückgreifen. Einhundertprozentige Sicherheit kann also auch hier nicht gewährt werden; doch gegenüber einer große Klasse von Eingabe-basierten Gefährdungen, unter die unter anderem die zuvor genannten SQL Injections und Buffer Overflows zählen, können Ansätze wie Crema erheblich zur Sicherheit des Systems beitragen.

V. Fazit

Unter anderem gehen aus den behandelten Themengebieten folgende Erkenntnisse im Speziellen hervor:

- Sprachen, die Entwicklern die Verwaltung des Speichers entziehen, sind gegenüber schwerwiegenden Gefährdungen wie Buffer Overflows geschützt, und in dieser Hinsicht die sichereren Sprachen.
- Die Sprache Rust erzielt durch eine restriktive Semantik eine im Gesamten höhere Sicherheit der darin programmierten Anwendungen.
- Durch eine Sprache wie Crema, deren Mächtigkeit bewusst eingeschränkt ist, können Programmmodule (vor allem zum Einlesen von Eingaben) sicherer und (effizient) verifizierbar implementiert werden.

Es ergibt sich im Gesamtbild der Eindruck, dass Sprachen-basierte Verbesserungen der Anwendungssicherheit im Normalfall mit Einschränkungen in der Entwicklungsmethodik einhergehen. Dies kann größtenteils darauf zurückgeführt werden, dass es letztendlich die Menschen sind, die für die Sicherheit des Codes verantwortlich sind, und an denen die Sicherheit des Codes auch maßgeblich scheitert. Gut gewählte Programmiersprachen sind indes in der Lage, den Entwickler in eine bestimmte Richtung zu weisen und damit die erwartete Sicherheit des Codes deutlich zu erhöhen. Allerdings ist bei einer sichereren Sprache tendenziell auch immer ein höherer Aufwand (bezüglich der Schulung, des Aufwands pro Zeile, und/oder der Nutzung von Tools) einzuplanen.

Literatur

[1] Interlexikon | Verifizierung und Beschreibungen. Dimaweb Network Solutions. Weblink: https://www.dimaweb.at/fachbegriffe/verifizierung.html/ (zuletzt abgerufen am 16.07.2016)

[2] State of Software Security. Focus on Application Development. Veracode, 2014. Weblink: http://www.helpnetsecurity.com/2014/04/15/the-security-of-the-most-popular-programming-languages/ (zuletzt abgerufen am 11.07.2016)

[3] Halfond, William et al.: A classification of SQL-injection attacks and countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering. Vol. 1. IEEE, 2006.

[4] Whitehat Security Report. Whitehat, 2015. Weblinks: http://www.upguard.com/blog/which-web-programming-language-is-the-most-secure/ ; http://threatpost.com/security-begins-with-choice-of-programming-language/105441/ (zuletzt abgerufen am 11.07.2016)

[5] The Rust Programming Language. Weblink: https://www.rust-lang.org/en-US/ (zuletzt abgerufen am 13.07.2016)

[6] Hoare, G.: The Rust Programming Language. Weblink: http://www.rust-lang.org/book/ (zuletzt abgerufen am 11.07.2016)

[7] Metz, Cade: The Epic Story of Dropbox' Exodus from the Amazon Cloud Empire. Wired, 14.03.2016. Weblink: http://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/ (zuletzt abgerufen am 16.07.2016)

[8] Yegulalp, Serdar: Mozilla's Rust-based Servo browser engine inches forward. infoworld, 03.04.2015. Weblink: http://www.infoworld.com/article/2905688/applications/mozillas-rust-based-servo-browser-engine-inches-forward.html (zuletzt abgerufen am 16.07.2016)

[9] Yegulalp, Serdar: Rust's Redox OS could show Linux a few new tricks. infoworld, 21.03.2016. Weblink: http://www.infoworld.com/article/3046100/open-source-tools/rusts-redox-os-could-show-linux-a-few-new-tricks.html (zuletzt abgerufen am 16.07.2016)

[10] Neumann, Alexander: „Programmiersprache: Webbrowser Firefox 48 enthält ersten Rust-Code". heise Developer, 13.07.2016. Weblink: http://www.heise.de/developer/meldung/Programmiersprache-Webbrowser-Firefox-48-enthaelt-ersten-Rust-Code-3265382.html (zuletzt abgerufen am 13.07.2016)

[11] Yegulalp, Serdar: 4 projects ripe for a Rust rewrite. Infoworld Tech Watch, 3.5.2016. Weblink: http://www.infoworld.com/article/3064686/open-source-tools/4-projects-ripe-for-a-rust-rewrite.html (zuletzt abgerufen am 11.07.2016)

[12] The Rust Core Team: Announcing Rust 1.0. The Rust Programming Language Blog, 15.05.2016. Weblink: http://blog.rust-lang.org/2015/05/15/Rust-1.0.html (zuletzt abgerufen am 11.07.2016)

[13] Chomsky, Noam: Three models for the description of language. IRE Transactions on Information Theory. Vol.2, 1956, S. 113–124

[14] Church, Alonzo: An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58, 1936. S. 345-363.

[15] Turing, Alan Mathison: On computable numbers, with an application to the Entscheidungsproblem. Journal of Math 58.345-363 (1936): 5.

[16] Language-theoretic Security. Website: http://langsec.org/ (zuletzt abgerufen am 11.07.2016)

[17] Torrey, Jacob I. et al.: Verification State-Space Reduction Through Restricted Parsing Environments. 2015 IEEE CS Security and Privacy Workshops.

[18] Pattis, Richard: Ebnf: A notation to describe syntax. 2013.

17. Juli 2016, im Rahmen eines Seminars zum Thema „Security in Software Engineering" an der Universität Stuttgart