What are Security Patterns?

A Formal Model for Security and Design of Software

Anika Behrens
University of Bremen
Bremen, Germany
anikab@informatik.uni-bremen.de

ABSTRACT

Security patterns aim to be best practices for avoiding security-related design flaws in software. There is a potential demand for model-driven software tools processing security patterns automatically to support designers, developers, and auditors. Automatically generating or detecting security pattern instances in source code requires a formal specification of instructions how to implement secure software, instead of a description form that is designed for humans. We present a formal definition of security patterns more suitable for automated processes of code generation or code verification. We demonstrate that security patterns theoretically lead to formal constraints that range over all Chomsky types and show how formal security patterns of Chomsky type 3 can be reduced to particular expressions under some assumptions. Finally, we formalize abstractions of security patterns and give a mathematical model as a basis for a data structure for security patterns to be processed by software tools.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Software security engineering**; *Security requirements*; • **Software and its engineering → Design patterns**; *Model-driven software engineering*;

KEYWORDS

Security Patterns, Design Patterns, Formal Model, Formal Language, Formal Alphabet, Finite State Machine, Chomsky Type

ACM Reference Format:
Anika Behrens. 2018. What are Security Patterns?: A Formal Model for Security and Design of Software. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3230833.3230834

1 INTRODUCTION

Security patterns aim to help humans to design secure architectures and to develop secure software. We conjecture that security patterns would gain additional value if their textual and graphical description form was supplemented by a formal mathematical model of executable instructions representing the set of solutions. Although this model might not be as intuitive as the current representation, it could be processed automatically within model-driven tools for designers and auditors. Supporting designers with tools might help them to implement security patterns step-by-step. Furthermore, we might provide guidelines for auditors to detect security pattern instances in software. We give a definition of formal security patterns and show formal language properties that should be respected for automated processes, such as security pattern detection or code generation. The Chomsky hierarchy provides four types of formal grammars: 0 (recursively enumerable), 1 (context-sensitive), 2 (context-free), 3 (regular). We show that formal security patterns cover structures ranging over all Chomsky types not limited to sequences or regular expressions in general. This paper does not aim to give a concrete syntax for a description language, but demonstrates properties of this model that need to be respected before defining any formal security pattern syntax, and before implementing security pattern tools. Security patterns mostly present solutions independently from particular programming languages and implementations, such that it is not possible to extract concrete source code from the pattern description directly. Therefore, we define alphabets that can be transformed into different abstractions to define model transformations that formalize the abstraction process from concrete implementation structures to a library-independent programming structure. As for the title of this paper, we focus on a formal definition of the structures that are given by the concept of security patterns:

(1) What are security patterns?

We will also address the following research questions:

(2) How can the formal structure of security patterns be characterized?
(3) Which language properties do security patterns have?
(4) How can we define transformations between different abstractions of security patterns?

2 STATE OF THE ART

There are several collections of security patterns. Yoder *et al.* [17], Schumacher *et al.* [14], Fernandez-Buglioni [10], Steel *et al.* [16], Okubo *et al.* [13], and the Open Group Security Forum [4] present security patterns in a textual and graphical form, readable for humans. The text is written in a natural language (English) and is grouped by sections similar to the GoF template for design patterns of Gamma *et al.* [11]. The graphical description form is mainly characterized by UML class and sequence diagrams, or their derivatives: Okubo *et al.* [13] use other graphical visualizations different from UML. The Open Group Security Forum [4] uses diagrams different from UML for their secure proxy pattern. Yoder *et al.* [17] demonstrate graphical user interfaces (GUI). Security pattern relations are shown in diagrams with textual attributes: Yoder *et al.* [17] denote this by *Pattern Interaction Diagram*, Steel *et al.* [16] by *Core Security Patterns Catalog*, Fernandez-Buglioni [10] by *Pattern Diagram*. To describe security patterns, Steel *et al.* [16] added concrete Java libraries and source code snippets. Bunke *et al.* [7] use UML sequence diagrams to describe structures in source code that represent security pattern instances. Späth *et al.* [15] do not use the term security pattern, but some of their structures are similar to a sub-model of our approach. More concretely, they use finite state machines (FSM) to encode typestate properties of their model. Bartussek *et al.* [3] refer to sequences of calls similar to our alphabet and define an equivalence relation on traces by using assertions. We use formal languages and define abstractions to be sets of maps between alphabets, instead. Dougherty *et al.* [9] show the concept of security design patterns and fill a gap between security patterns and design patterns, but they do not give a mathematical model of security patterns or design patterns. Krüger *et al.* [12] describe a language for correct usage of cryptographic APIs. Yskout *et al.* [18] performed an empirical study that did not prove that security patterns really improve productivity of designers and security of their software. They propose to improve the state of the art in the field of security patterns, but do not suggest any particular approach. There is work on software tools to detect security patterns in source code by Bunke *et al.* [7] and Alvi *et al.* [2], but we have not found any work on model-driven software tools to implement security patterns in source code step-by-step. There is one approach to formalize security patterns by da Silva Júnior *et al.* [8], suggesting Petri nets as a formal language for security patterns to evolve a security pattern detection tool. We give a formal definition of security patterns to analyze characteristic language properties first, necessary for any approach to security pattern detection. We conjecture that security pattern tools would strongly improve the state of the art in the field of software security and design. The formal abstraction maps provided in this paper allow to construct a data structure for security patterns suitable for automated processes, such as model-driven software tools.

3 FORMALIZATION

3.1 Terminology

We start with a short literature research on the terms *Pattern*, *Security Pattern* and *Design Pattern*: Gamma *et al.* [11] cite the following statement, which was originally given by Alexander *et al.* [1], dealing with **patterns** for construction of towns and buildings: *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such way that you can use this solution a million times over, without ever doing the same way twice."* Schumacher *et al.* [14] summarize their pattern definition as follows: *"A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged to form multiple concrete design structures, as well as a process for creating any particular structure."* Gamma *et al.* [11] define the following patterns to be **design patterns**: *"The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context".* Schumacher *et al.* [14] give a similar definition for **security patterns**: *"A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular such structure."* Furthermore, Schumacher *et al.* [14] says: *"Note that security patterns are not limited to architectural or design patterns. Depending of the type of the problem, the fundamental solution principle can be at one or more different levels – organization, architecture, operations, processes, and so on."* Bunke [5] gives the following property of security patterns: *"Security patterns describe best practices to handle recurring security problems."* Bunke *et al.* [6] define *software-security patterns* as follows: *"Later on we will use the terminology software-security patterns that describe software-related security patterns. These patterns describe security aspects relevant in software design, development, and maintenance."* Later they specify *"Software-security patterns describe mostly how to structure parts of software to ensure security requirements."* We see that the definitions for security patterns are given in a natural language (English) and do not provide any formal definition or formal properties that characterize security patterns. Unfortunately, the definitions differ in their choice of words. Finally, we conclude that it seems to be unclear what is commonly understood to be a security pattern. In the following section we show why we conjecture that it is possible to formalize the security pattern properties and we give a formal definition of security patterns.

3.2 Formal Definition

To overcome the discrepancy between textual and mathematical models, we state the following conjecture:

CONJECTURE 3.1. *The solutions given by security patterns or design patterns, that are currently described in textual and graphical form, can be described with a formal language over an alphabet of executable instructions. In this case, we define executable instructions to be any information addressed to machines or humans advising them to run a particular, well-defined step of a procedure.*

For example, an executable instruction might be a concrete advise which lines of code should be implemented in which order. It might also be some part of a step-by-step instruction specifying how to configure a router or firewall using a particular user interface.

Unfortunately, as we have not found any formal definition of security patterns yet, one needs to interpret existing descriptions rather than being able to prove this conjecture formally. We assume that security patterns aim to propagate ideas to help designers or developers implementing secure software. Furthermore, we assume that security patterns contain a solution that can be executed, or that can be at least extracted from the security pattern and that can be defined somehow, although it is not directly given as an example. If Conjecture 3.1 did not hold, this would interfere with the original idea of the pattern properties given by Alexander *et al.* [1] and cited by Gamma *et al.* [11].

Definition 3.2. Let $\mathcal{L}$ be a set of programming languages. Then we define a *formal pattern* to be a formal language

$$P = P\left(\bigcup_{L\in\mathcal{L}} A(L)\right) \text{ over the alphabet } \bigcup_{L\in\mathcal{L}} A(L),$$

where each $A(L)$ is an alphabet dependent on $L \in \mathcal{L}$.

Note that the alphabet is not limited to be finite, and the formal language P is not limited to a particular Chomsky type here.

For this general pattern definition one could allow a quite abstract definition of *programming language*. It could be a formal language leading to instructions for a computer, but, as security patterns can be at organizational or operational levels [14], it might also contain operations that can be executed by humans, e.g. instructions for secure systems integration or correct usage of a particular software interface. This allows us to extend the first conjecture:

Conjecture 3.3. *The problems addressed by a security pattern can be identified with a set of attacks described with a formal language over an alphabet of executable instructions.*

Now we are able to express what we understand to be the characteristic property of a security pattern:

Conjecture 3.4. *Assume that Conjecture 3.1 and 3.3 hold. Then every prefix of every sequence of executable instructions accepted by the solution of a security pattern is equivalent to a system state that is reproducibly secure. Secure means in this context that there is no attack described by the problem of the security pattern that would terminate and succeed.*

As described above, we are not able to prove these conjectures formally, because we need to interpret the pattern description and need to formalize what is commonly understood to be *security*. In the following, we assume that Conjecture 3.1, 3.3, and 3.4 hold.

Addressing the first research question **"How can the formal structure of security patterns be characterized?"** we give the following definition:

Definition 3.5. A *formal security pattern* is a pair of two *formal patterns*, one representing a security problem and one representing a solution for that problem, such that the security property of Conjecture 3.4 holds.

In the following examples, we do not focus on formal models for attacks or executable instructions at organizational or operational levels. We concentrate on solutions of software-related security patterns. In the next section we construct an alphabet of instructions over which software-security patterns can be defined.

3.3 Constructing an Alphabet

In the following, we consider classes and interfaces as used in $L =$ `Java`. Let lib be a countable set of interfaces or abstract or implemented classes in L, and let prim denote the set of primitive data types in L. Further, let type := lib ∪ prim and let ident be a countable set of identifiers for objects, e.g. variables, such that there is a map $\text{inst} : \text{ident} \to \text{type}$. For $c \in \text{lib}$ let $\text{meth}(c)$ denote the set of *method signatures* provided by c. Further, let $\text{static}(c) \subset \text{meth}(c)$ be the subset of `static` methods and *constructors* and let $\text{void}(c) \subset \text{meth}(c)$ be the subset of `void` methods. Let $n(m) \in \mathbb{N}_0$ denote the number of parameters of a method $m \in \text{meth}(c)$. Define $\text{param}(m) \in \text{type}^{n(m)}$ to be the parameter structure of a method m. Let $\text{result}(m) \in \text{type}$ denote a set of allowed return types of a method $m \in \text{meth}(c) \setminus \text{void}(c)$, including interface implementations or extended classes or extended interfaces of the return type. Finally, write $\text{inst}^n : \text{ident}^n \to \text{type}^n$, $n \in \mathbb{N}_0$ for the map from instances to types defined on a list.

Definition 3.6. We define an alphabet $\mathcal{A}(L, \text{lib}, \text{ident}, \text{inst})$ for software pattern instances to be the union $\mathcal{A} := A_0 \cup A_1 \cup A_2 \cup A_3$, where

$$\begin{aligned}
A_0 &:= \textit{Set of static method calls on a class or interface}\\
A_1 &:= \textit{Set of non-static method calls on an identifier}\\
A_2 &:= \textit{Set of non-static method calls on an identifier}\\
&\quad\ \textit{with an identifier representing the return value}\\
A_3 &:= \textit{Set of static method calls on a class or interface}\\
&\quad\ \textit{with an identifier representing the return value.}
\end{aligned}$$

Note that A_2 and A_3 differ from A_0 and A_1 in tracking some return value. More concretely, we define the following:

$$\begin{aligned}
A_0 &:= \{(c,m,S) \mid c \in \text{lib} \wedge m \in \text{static}(c) \wedge\\
&\quad S \in \text{ident}^{n(m)} \wedge \text{inst}^{n(m)}(S) = \text{param}(m)\}\\
A_1 &:= \{(s,m,S) \mid s \in \text{ident} \wedge \text{inst}(s) \in \text{lib} \wedge\\
&\quad m \in \text{meth}(\text{inst}(s)) \wedge S \in \text{ident}^{n(m)} \wedge\\
&\quad \text{inst}^{n(m)}(S) = \text{param}(m)\}\\
A_2 &:= \{(r,s,m,S) \mid r,s \in \text{ident} \wedge \text{inst}(s) \in \text{lib} \wedge\\
&\quad m \in \text{meth}(\text{inst}(s)) \setminus \text{void}(\text{inst}(s)) \wedge\\
&\quad \text{inst}(r) \in \text{result(m)} \wedge S \in \text{ident}^{n(m)} \wedge\\
&\quad \text{inst}^{n(m)}(S) = \text{param}(m)\}\\
A_3 &:= \{(r,c,m,S) \mid r \in \text{ident} \wedge c \in \text{lib} \wedge\\
&\quad m \in \text{static}(c) \setminus \text{void}(c) \wedge \text{inst}(r) \in \text{result(m)}\\
&\quad \wedge S \in \text{ident}^{n(m)} \wedge \text{inst}^{n(m)}(S) = \text{param}(m)\}.
\end{aligned}$$

Note that the alphabet $\mathcal{A}$ is countable, since we defined ident and lib to be countable, and if we assume any fixed library interface to be countable.

3.4 Notation

To make examples more readable, we will use a pseudocode notation for security patterns:
Let $c \in \text{lib}$, $m \in \text{meth(c)}$, $S = (s_1, \ldots, s_n) \in \text{ident}^{n(m)}$, and let $x, r \in \text{ident}$. For elements $u \in \mathcal{A}$ we write:

`u: c.m(s1,..,sn)`	≃	$(c,m,S) =: u \in A_0$
`u: s.m(s1,..,sn)`	≃	$(s,m,S) =: u \in A_1$
`u: r=s.m(s1,..,sn)`	≃	$(r,s,m,S) =: u \in A_2$
`u: r=c.m(s1,..,sn)`	≃	$(r,c,m,S) =: u \in A_3$

For any constructor $C \in \text{meth}(c)$ we write `u: new C(s1,..,sn)` or `u: r = new C(s1,..,sn)`. For $\text{inst} : s \mapsto c$, we write `s : c`.

In the following example, we use a finite state machine to denote a formal pattern of Chomsky type 3.

3.5 Example: URL Connection

Now we formalize a pattern that describes how to connect to a host via an URL object.

Example 3.7. Let L = `java` and let lib be identified with the packages `java.lang` and `java.net`. Any method signatures are given due to the `API`. As an example, we take the following identifiers:

```
url : java.net.URL
protocol : java.lang.String
port : int
base : java.lang.String
spec: java.lang.String
handler : java.net.URLStreamHandler
url1 : java.net.URL
```

We define the following statements `u0,...,u5` representing elements from A_3:

```
u0: url = new URL(spec)
u1: url = new URL(protocol,
      base, port, spec)
u2: url = new URL(protocol,
      base, port, spec, handler)
u3: url = new URL(protocol,
      base, spec)
u4: url = new URL(url1, spec)
u5: url = new URL(url1, spec, handler)
```

Now set `u` := {`u0,u1,u2,u3,u4,u5`} and define `u6` in A_1 as follows:

```
u6: url.openConnection()
```

Then the finite state machine given in Fig. 1 defines a formal *Pattern*, where q_0, q_1, q_2 represent the following states:

q_0 :	start
q_1 :	url is set
q_2 :	connection is open

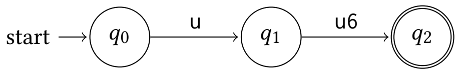

Figure 1: Finite state machine for a URL connection

This pattern is equivalent to the following regular expression:

```
( u0 | u1 | u2 | u3 | u4 | u5 ) u6
```

3.6 Kripke Structures

In the previous example we used a finite state machine to express a formal language. Referring to Conjecture 3.4, security patterns can be defined by security constraints on particular system states. *Kripke* structures can be used to express relations between system states and to bind particular requirements on these states. In the following, we apply *Kripke* structures to security patterns.

Example 3.8. The previous Example 3.7 can be defined to be insecure, if a connection is opened without a secure protocol. Hence, we like to express that the protocol of `URL` has to be equal to `"https"` before opening the connection. Reading the Java API, we see that there are multiple ways to set this protocol. We use `Java` syntax to express three statements `p1`, `p2`, and `p3` and define the security constraint as follows: *At least one of the following statements would return true, if it was executed before the connection is opened:*

```
p1: protocol.equals("https")
p2: spec.startsWith("https")
p3: url1.getProtocol().equals("https")
```

Now we add these constraints on system states to our description form and identify security patterns with *Kripke* structures as follows: Figure 2 shows a FSM that is equivalent to the FSM given in Figure 1, but contains more states p_0 (initial state), p_1 (`protocol` is secure), p_2 (`spec` is secure), p_3 (`url1` is secure), p_4 (`url` is set), and p_5 (connection is open). The labels `u0`,...,`u6` refer to Example 3.7. The empty word is denoted by the symbol ε. The constraints given by `p1`, `p2` and `p3` can be related to the states p_1, p_2, and p_3 of the automaton. Note that the constraints refer to particular values of `String` objects. The constraint given by `p3` is quite complex since it refers to an object of `URL` itself.

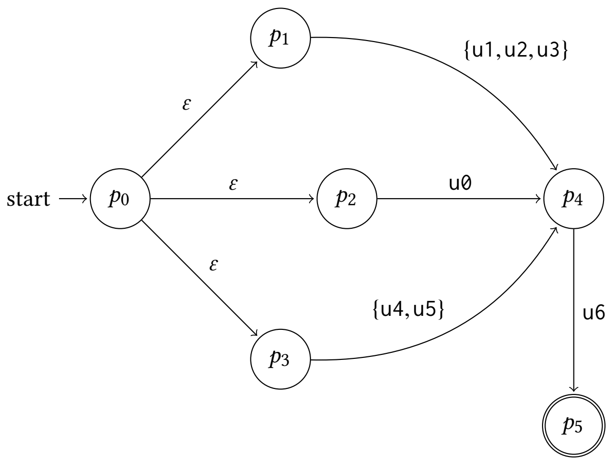

Figure 2: FSM representing a *Kripke* Structure

The pattern does not determine how to build each String in detail, but says that it has to have a particular value or property leading to a secure system state. It might be possible that its value is not constant, but dependent on the execution path of the program.

Other constraints might be thinkable. For example, one could demand that a `String` value passed to another particular method in L causes a certain value (e.g. `true`) to be returned. Hence, the following theorem holds, addressing the research question **"Which language properties do security patterns have?"**.

Theorem 3.9. *Security patterns described by Kripke structures using constraints in L can lead to formal patterns that range over all Chomsky types.*

Proof. This is obvious, because security patterns can be defined by constraints on system states according to Conjecture 3.4 and because L is Turing-complete. □

Corollary 3.10. *If we allow security patterns to be described by Kripke structures using constraints in L, it can be undecidable if a given source code is an implementation of a security pattern or not.*

Proof. This follows directly from Theorem 3.9 by considering the halting problem. □

Hence, formal security patterns of Chomsky type 0 (accepted by Turing machines) are not always suitable for automated processes of security pattern detection. We need a data structure that can be used as an input for software tools, though. So, we advise to characterize security patterns by Chomsky type and to respect the fact that it is theoretically not possible to detect all of them. Defining the alphabet of formal security patterns to be possibly infinite might cause similar problems. For example, algorithms that iterate over an infinite subset of elements of the alphabet may not terminate, either. In the following section we show how to reduce the alphabet for the pattern definition to a possibly finite subset.

3.7 Spanning Alphabet

The alphabet of a formal security pattern is not limited to be finite, because the set of all executable instructions that can be implemented in L is not finite. In this section we define the infinite alphabet covering all possible implementations in L and show how to extract those elements from this general alphabet that are necessarily used to describe the pattern. If this subset is finite, it will be suitable for pattern definition in practice. For $L =$ `Java` let $\mathrm{LIB}(L)$ be the set of all possible libraries programmable in L, and let $\mathrm{IDENT}(L)$ be the set of all possible identifiers allowed in L. Note that LIB and IDENT are countable sets. Let $\mathrm{INST} : \mathrm{IDENT} \to \mathrm{LIB}$ be a map. Further, let $\mathrm{ident} \subset \mathrm{IDENT}$ and $\mathrm{lib} \subset \mathrm{LIB}$ be countable sets and let $\mathrm{inst} : \mathrm{ident} \to \mathrm{lib}$.

Definition 3.11. We define a *complementary alphabet* C representing all letters that have nothing in common with elements from $\mathcal{A}(L, \mathrm{lib}, \mathrm{ident}, \mathrm{inst})$:

$$C := C_0 \cup C_1 \cup C_2 \cup C_3,$$

where

$$\begin{aligned}
C_0 &:= \{(c, m, S) \in A_0(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST}) \mid \\
&\quad c \notin \mathrm{lib} \wedge \forall i \in \{1, \ldots, n(m)\} : S_i \notin \mathrm{ident}\} \\
C_1 &:= \{(s, m, S) \in A_1(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST}) \mid \\
&\quad s \notin \mathrm{ident} \wedge \forall i \in \{1, \ldots, n(m)\} : S_i \notin \mathrm{ident}\} \\
C_2 &:= \{(r, s, m, S) \in A_2(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST}) \mid \\
&\quad r, s \notin \mathrm{ident} \wedge \forall i \in \{1, \ldots, n(m)\} : S_i \notin \mathrm{ident}\} \\
C_3 &:= \{(r, c, m, S) \in A_3(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST}) \mid \\
&\quad r \notin \mathrm{ident} \wedge c \notin \mathrm{lib} \wedge \forall i \in \{1, \ldots, n(m)\} : S_i \notin \mathrm{ident}\}.
\end{aligned}$$

Definition 3.12. Let C be the complementary alphabet according to Definition 3.11. Let P be a pattern over $\mathcal{A}(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST})$. We call $\mathcal{A}(L, \mathrm{lib}, \mathrm{ident}, \mathrm{inst})$ *spanning alphabet* for P, iff for all $x, y \in \mathcal{A}(L, \mathrm{LIB}, \mathrm{IDENT}, \mathrm{INST})$ the following four properties hold:

- $\mathrm{inst} = \mathrm{INST}|_{\mathrm{ident}} : \mathrm{ident} \to \mathrm{lib}$
- $P \subset (\mathcal{A}(L, \mathrm{lib}, \mathrm{ident}, \mathrm{inst}) \cup C)^*$
- $xy \in P \Leftrightarrow xC^*y \in P$
- $x \in P \Leftrightarrow C^*xC^* \in P$,

where the symbol * denotes the *Kleene* star.

Example 3.13. Now we give an example for a spanning alphabet. We take a pattern that describes how to generate a seed for a `SecureRandom`, as part of the encryption pattern. Let $k \in \mathbb{N} \setminus \{0\}$. Take the following identifiers:

```
sr : java.security.SecureRandom
k : int
bseed : byte[k]
numBytes : int
```

Define the following statements:

```
sr0: sr = new SecureRandom()
sr1: sr = new SecureRandom(bseed)
sr2: bseed = sr.generateSeed(numBytes)
sr3: bseed = SecureRandom.getSeed(numBytes)
sr4: sr.setSeed(bseed)
```

We define C to be the complementary alphabet for `sr0`,...,`sr4` according to Definition 3.11. Fig. 3 shows an automaton for this pattern, where $q_0, \ldots, q_4$ represent the following states:

q_0 :	numBytes is set
q_1 :	sr is generated without bseed
q_2 :	sr is generated without bseed and bseed is defined by sr
q_3 :	sr is generated with parameter bseed
q_4 :	bseed is defined by SecureRandom

Then `sr0`,...,`sr4` is a spanning alphabet for this pattern.

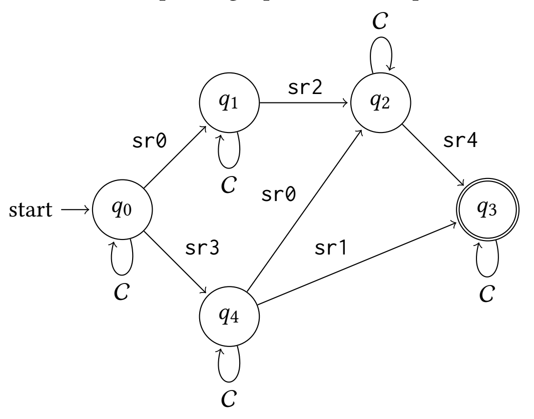

Figure 3: Finite state machine for `SecureRandom` with spanning alphabet

We see that loops are labeled with C only. Addressing research question **"Which language properties do security patterns have?"** again, this leads to the next Conjecture and Theorem:

Conjecture 3.14. *If a security pattern leads to a formal pattern of Chomsky type 3 with a spanning alphabet $\mathcal{A}$ and a complementary alphabet C, then loops in a corresponding FSM are labeled with C only.*

Any loop labeled with an executable instruction of the spanning alphabet at a certain state would define this instruction to be optional, because any path visiting this state can be reduced in length by ignoring the loop. If Conjecture 3.14 did not hold, this would interfere with the idea of security patterns to be best practices.

Theorem 3.15. *Let P be a formal security pattern of Chomsky type 3 with a finite spanning alphabet $\mathcal{A}$. If Conjecture 3.14 holds, then P can be identified with a finite set of sequences over the alphabet $\mathcal{B} := \{C^*aC^* \mid a \in \mathcal{A}\}$ of particular regular expressions.*

Proof. Because P is of Chomsky type 3, there exists a FSM representing the pattern. According to Conjecture 3.14 loops are labeled with C only. Because $\mathcal{A}$ is a spanning alphabet, these loops for C are located at each state, hence $P \subset \mathcal{B}^*$. For each $a \in \mathcal{A}$ the expression C^*aC^* contains an infinite set of sequences. Nevertheless, $\mathcal{B}$ is a finite set of these expressions, because $\mathcal{A}$ is finite and there exists a bijection $\mathcal{A} \to \mathcal{B}$. Because the FSM has a finite amount of states, this leads to a finite amount of sequences over $\mathcal{B}$. □

Finally, according to Conjecture 3.14 the formal structure and the language properties of security patterns are not equivalent to general properties of regular expressions.

Theorem 3.15 assumes that the spanning alphabet is finite. Restricting the alphabet to be finite would force us to choose particular libraries and identifiers, but we like to have some structure that is independent of particular implementations, instead. Having defined the spanning alphabet, we are now able to give a definition of formal abstraction maps on formal patterns in the next section.

3.8 Abstract Pattern

Addressing the research question **"How can we define transformations between different abstractions of security patterns?"** we define a map $\mu : \text{ident} \to \text{IDENT}$ enabling us to define patterns independently from particular names of variables. Analogously, we define $\lambda : \text{lib} \to \text{LIB}$ representing an abstraction from concrete libraries.

Definition 3.16. We define an *abstract pattern* $\mathcal{P}$ to be a set of the form

$$\mathcal{P} = \bigcup_{\varphi \in \Phi} \varphi(p_L), \text{ where}$$

- $\omega \in \varphi(p_L) \Leftrightarrow \exists v \in p_L : \varphi(v) = w$ for all φ
- p_L is a formal pattern over $\mathcal{A}(L, \text{LIB}, \text{IDENT}, \text{INST})$
- there exists a spanning alphabet $\mathcal{A}(L, \text{lib}, \text{ident}, \text{inst})$ of p_L
- $\Phi \subset \Omega$ is a set of maps, where

$$\begin{aligned}
&\Omega := \{\varphi_{\lambda,\mu} : \mathcal{A}(L, \text{lib}, \text{ident}, \text{inst}) \to \\
&\mathcal{A}(L, \lambda[\text{lib}], \mu[\text{ident}], \nu) \mid \\
&\mu : \text{ident} \to \text{IDENT injective } \wedge \\
&\lambda : \text{type} \to \text{LIB} \cup \text{prim injective } \wedge \forall c \in \text{lib} : \\
&\lambda \text{ induces an injection meth}(c) \to \text{meth}(\lambda(c)) \wedge \\
&\nu : \mu[\text{ident}] \to \lambda[\text{lib}], \nu = \lambda \circ \text{inst} \circ \mu^{-1}\},
\end{aligned}$$

where $\lambda[\text{lib}]$ and $\mu[\text{ident}]$ denote the images of λ and μ.

In other words, this defines equivalence classes on variable names and libraries, and the pattern is defined over representatives of these equivalence classes. This enables us to characterize security patterns with concrete abstractions, in particular with a concrete set Φ of abstraction maps. This mathematical model can be used for a data structure of security patterns within model-driven software tools supporting designers, developers, and auditors in future.

4 CONCLUSIONS

We define security patterns to be formal languages, rather than using textual or graphical description forms. There is a dependence on a concrete choice of security libraries, but we identified this to be an attribute of the alphabet, rather than the pattern, which enables us to identify formal abstraction maps on the alphabet and to formalize the abstraction of a pattern (Definition 3.16). Security patterns found in literature can lead to constraints on system states, such that formal security patterns range over all Chomsky types and can be classified by their language properties (Theorem 3.9). Under some assumptions, formal security patterns of Chomsky type 3 can be reduced to a particular form, such that it is not necessary to use general FSMs to define security pattern instances in these cases (Theorem 3.15).

REFERENCES

[1] C. Alexander, S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction.* OUP USA.

[2] Aleem Alvi and Mohammad Zulkernine. 2017. In *IEEE International Conference on Software Security and Assurance (ICSSA).*

[3] Wolfram Bartussek and David L. Parnas. 1978. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology*, Giampio Bracchi and Peter Christian Lockemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–236.

[4] Craig Heath Bob Blakley and members of The Open Group Security Forum. [n. d.]. *Technical Guide, Security Design Patterns.* Technical Report.

[5] Michaela Bunke. 2014. On the Description of Software Security Patterns. In *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP '14).* ACM, New York, NY, USA, Article 34, 10 pages. https://doi.org/10.1145/2721956.2721990

[6] Michaela Bunke, Rainer Koschke, and Karsten Sohr. 2012. Organizing Security Patterns Related to Security and Pattern Recognition Requirements. *International Journal On Advances in Security* 5, 1&2 (July 2012), 46–67.

[7] Michaela Bunke and Karsten Sohr. 2011. An architecture-centric approach to detecting security patterns in software. In *International Symposium on Engineering Secure Software and Systems.* Springer, 156–166.

[8] Luis Sérgio da Silva Júnior, Yaan-Gael Guéhéneuc, and John Mullins. 2013. *An approach to formalise security patterns.* Technical Report. Citeseer.

[9] Chad R Dougherty, Kirk Sayre, Robert Seacord, David Svoboda, and Kazuya Togashi. 2009. Secure design patterns. (2009).

[10] Eduardo Fernandez-Buglioni. 2013. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns* (1st ed.). Wiley Publishing.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Stefan Küger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2017. CrySL: Validating Correct Usage of Cryptographic APIs. *arXiv preprint arXiv:1710.00564* (2017).

[13] Takao Okubo and Hidehiko Tanaka. 2008. Web Security Patterns for Analysis and Design. In *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP '08).* ACM, New York, NY, USA, Article 25, 13 pages. https://doi.org/10.1145/1753196.1753226

[14] Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. 2005. *Security Patterns: Integrating Security and Systems Engineering.* John Wiley & Sons.

[15] Johannes Späth, Karim Ali, and Eric Bodden. 2017. IDE al: efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 99.

[16] Christopher Steel, Ramesh Nagappan, and Ray Lai. 2006. *Core security patterns: Best practices and strategies for J2EE, Web services, and identity management.* Prentice-Hall. http://www.coresecuritypatterns.com/

[17] Joseph Yoder and Jeffrey Barcalow. 1998. Architectural patterns for enabling application security. *Urbana* 51 (1998), 61801.

[18] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2015. Do security patterns really help designers?. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 292–302.