Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Analysing and improving the crypto ecosystem of Rust

Philipp Keck

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Stefan Wagner

Supervisor: Kai Mindermann M.Sc.

Commenced: 2016-10-13

Completed: 2017-04-14

CR-Classification: D.2.2, D.2.11, D.2.13, E.3

Abstract

Context: Rust is an emerging systems programming language that suits security-critical applications because it guarantees memory safety without a garbage collector. Its growing ecosystem already encompasses several crypto libraries, though the competition is still open. Previous cryptography research found that vulnerabilities are often due to misunderstandings and misuse of cryptographic APIs rather than bugs in the libraries themselves. *Aim:* This thesis presents a holistic analysis of Rust's current crypto ecosystem and aims to improve its further development. A particular focus is on API design because all libraries are still open to change their APIs and it will become increasingly difficult to change them later. *Method:* All parts of the ecosystem are systematically analysed, guided by the general structure of a crypto ecosystem. Research methods include a systematic search for libraries, a survey among contributors, GitHub analyses as well as a self-experiment and a controlled experiment to test the usability. *Results:* The contributors are typical open source developers and they collaborate in typical ways on GitHub. Most libraries have a clear main developer and there is a general lack of contributors. While two of the major libraries focus on usability and are consequently easier to use and more resistant to misuse, the two most widespread libraries consciously neglect these topics and exhibit flaws known from crypto libraries in other languages. *Conclusion:* The misuse resistant Rust crypto libraries should be advertised more actively. In the medium term, an officially endorsed API could improve interoperability and foster competition. For such an API and for the improvement of existing APIs, the thesis discusses a number of design decisions and their usability implications.

Kurzfassung

Kontext: Rust ist eine junge Systemprogrammiersprache, die sich für sicherheitskritische Anwendungen eignet, weil sie Speichersicherheit ohne einen Garbage Collector garantiert. Das wachsende Ökosystem umfasst bereits einige Krypto-Bibliotheken, wobei der Wettbewerb noch offen ist. Die bisherige Forschung hat gezeigt, dass Schwachstellen oft durch Missverständnisse und Missbrauch der kryptographischen APIs verursacht werden anstatt durch Fehler in den Bibliotheken selbst. *Ziel:* Diese Thesis enthält eine ganzheitliche Analyse des Krypto-Ökosystems von Rust mit dem Ziel, die zukünftige Entwicklung zu verbessern. Ein besonderer Fokus liegt auf dem API-Design, weil alle Bibliotheken noch offen für API-Änderungen sind und solche Änderungen später schwieriger werden. *Vorgehen:* Alle Bestandteile des Ökosystems werden anhand der allgemeinen Struktur eines Krypto-Ökosystems systematisch analysiert. Zu den eingesetzten Forschungsmethoden gehören eine systematische Suche nach Bibliotheken, eine Entwicklerumfrage, GitHub-Analysen sowie ein Selbstversuch und ein kontrolliertes Experiment um die Benutzbarkeit zu testen. *Ergebnisse:* Die Entwickler sind typische Open-Source-Entwickler und sie arbeiten auf typische Weise auf GitHub zusammen. Die meisten Bibliotheken haben einen eindeutigen Hauptentwickler und es gibt einen generellen Mangel an weiteren Entwicklern. Während zwei der größeren Bibliotheken sich auf Benutzbarkeit konzentrieren und dementsprechend einfacher zu verwenden und missbrauchsresistenter sind, vernachlässigen die beiden am weitesten verbreiteten Bibliotheken diese Themen bewusst und weisen Schwächen auf, die von Krypto-Bibliotheken anderer Sprachen her bekannt sind. *Fazit:* Die missbrauchsresistenten Krypto-Bibliotheken in Rust sollten aktiver beworben werden. Mittelfristig könnte eine offiziell unterstützte API die Interoperabilität und den Wettbewerb fördern. Für eine solche API und für die Verbesserung der existierenden APIs werden in der Thesis diverse Designentscheidungen und ihre Auswirkungen auf die Benutzbarkeit erörtert.

Acknowledgements

I would like to thank my supervisor Kai Mindermann for coming up with this exciting topic and for his guidance, advice and collaboration. Without his support, this thesis would not have been possible. I would also like to thank all survey participants for their time and input as well as the active community members on the #rust-crypto IRC channel and @briansmith for their help, tips and for interesting discussions.

Publication

Parts of this thesis have been submitted as a paper at the Thirteenth Symposium on Usable Privacy and Security (SOUPS '17), in collaboration with Kai Mindermann M.Sc. and Prof. Dr. Stefan Wagner. A separate paper has been submitted for the controlled experiment that is only briefly reported on in this thesis.

Note on links

Many of the sources referenced in this thesis are rather volatile. Where necessary, links are annotated with the date on which they were stored in the Internet Archive (https://web.archive.org/), where the respective version can be retrieved.

License

Contents

List of Figures

List of Tables

1 Introduction

The systems programming language Rust, sponsored by Mozilla, was published in 2010 and reached its first stable version in 2015. Its main goals are safety, speed and concurrency [Moz16]. Rust is able to guarantee memory safety without needing a garbage collector and thus rules out entire error classes like segmentation faults and (certain) race conditions at compile time. While it promises performance on par with other low-level languages like C, the syntax and the ecosystem are designed to make it feel more like a high-level language such as Java. Mozilla is already using Rust productively to build parts of its Firefox browser [Her16a] and a number of other companies have begun to experiment with the new language. Even though the Rust language itself is declared stable, it is still under heavy development to make it more productive [Mat+16] and the ecosystem is not mature, yet. According to the 2017 Stack Overflow developer survey, Rust is the most loved programming language for the second year in a row [Sta17].

The first libraries providing cryptographic primitives in Rust started being developed in late 2011. To my knowledge, there are currently 12 crypto libraries which offer a range of cryptographic primitives and 80 crypto libraries in total. The overall crypto ecosystem of Rust is in its early stages and the competition among libraries is still open, as is the technical question how to best implement cryptography in Rust.

Cryptography research has shown that many applications are vulnerable because their developers knowingly or unknowingly misuse cryptographic libraries—and not only due to cryptographic bugs in those libraries, as one might expect [EBFK13; FHM+12; GIJ+12; LCWZ14]. This misuse results from developers misunderstanding the libraries' interfaces and concepts, combined with an understandable lack of detailed cryptographic knowledge. Tulach [Tul08] formulates the concept of *selective cluelessness*. The idea is that software engineers—despite being good programmers—are allowed to know only little about components and tools they use, such as cryptographic libraries, if those are not central to their product. And more often than not, cryptographically securing sensitive data is not the main feature of an application, but merely a cross-functional requirement that pretty much any application using the internet is faced with. The goal is to "maximize cluelessness, while getting reliable results" [Tul08], and properly designed application programming interfaces (APIs) at the right level of abstraction are the "perfect tool" to reach this goal.

Many researchers and developers have pointed out that usable and misuse resistant API designs are just as crucial for crypto libraries as bug-free implementations of the cryptographic algorithms [DK14; LCWZ14; Min16; NKMB16], and many make concrete API design recommendations [FHP+13; FLW12; GS16]. The established crypto libraries such as OpenSSL, however, cannot make significant changes to their APIs for backward compatibility reasons. Instead, new libraries with a strong focus on API usability and misuse resistance are being developed. Notable examples are NaCl [Ber09], its offspring libsodium [Den] and Keyczar [DW08].

This thesis systematically analyses all parts of the crypto ecosystem of Rust, that is, the libraries, the developers and development processes, the influence of the Rust platform and the way the libraries are currently used. Suggestions to improve various aspects of the crypto ecosystem are made, especially regarding usability and misuse resistance. As none of those libraries guarantees long-term API stability yet, but most of them already implement executable primitives and higher-level protocols such as the Transport Layer Security (TLS) protocol, it is the right moment to (re)consider the API design. Particularly for a language like Rust, which focuses on and advertises safety, the ecosystem should also provide secure, fast and usable crypto libraries. Just like the Rust language is designed to prevent mistakes like dereferencing null pointers, a good Rust crypto library should be designed to prevent mistakes like nonce reuse, hard-coded keys, insecure comparisons, etc. And with all of Rust—the language itself, the compiler, the standard library, the package manager and all crypto libraries—being open source, the emerging crypto ecosystem of Rust is also an interesting scientific subject to study how crypto libraries are developed and how design decisions are made.

After an overview of the basic principles of Rust and cryptography in chapter 2 and a summary of related work in chapter 3, chapter 4 derives a definition of the term *crypto ecosystem* and outlines the crypto ecosystems of other well-known programming languages. Chapter 5 uses the definition to derive research questions (section 5.1) and analyse every *population* of the crypto ecosystem—with a systematic search (section 5.2), a survey among Rust crypto contributors (section 5.4) and an analysis of publicly available data on GitHub (section 5.5)—and to derive recommendations for improvement in certain areas (section 5.7). The remainder of the thesis singles out API usability as a particularly important aspect at this stage of the ecosystem. Chapter 6 analyses current usages of cryptographic primitives in Rust to find out which particular features are used. Chapter 7 reports on several experiments with the existing primitive libraries and chapter 8 makes recommendations for the design of cryptographic libraries and their APIs in Rust based on insights from the experiments and the literature. Finally, chapter 9 summarizes the thesis and looks at the future of the Rust crypto ecosystem and possible research directions.

2 Foundations

This chapter introduces the knowledge required to understand the following chapters. The explanations in this chapter assume that the reader is familiar with general concepts in computer science and a major programming language such as C++ or Java. Section 2.1 gives an overview of Rust's syntax and important differences from other programming languages. Section 2.2 covers the cryptographic primitives and concepts used in this thesis.

2.1 The Rust programming language

This section only covers a few particular aspects of the Rust programming language, standard library and build system, which are relevant to this thesis. For more details, please refer to the excellent and free online book *The Rust Programming Language* [Moz16]. All code snippets from this section are available for download in a runnable sample application (see appendix A).

Like many other languages, Rust's syntax is loosely based on C. The most obvious differences are some keywords (**`fn`** to define a function and **`let`** to create a *variable binding*) and the parentheses requirements being the other way round: Blocks such as **`if`** and **`for`** do *not* require parentheses around their condition/head, but curly braces around the body are mandatory:

```
fn main() {
    let mut x: i32 = 0; // i32 is a 32-bit signed integer
    for i in 1..10 {
        if i % 2 == 1 { x += i; }
        else { x -= i; }
    }
    println!("Alternating sum: {}", x); // will output 5
}
```

Variable bindings are immutable by default and need to be declared mutable explicitly with the **`mut`** keyword. Unlike in C or Java, the type is added after a colon instead of in front, and the type annotation is optional because the type can usually be inferred.

2.1.1 Arrays, vectors and slices

Arrays are built-in types and have a fixed size, whereas vectors are dynamically sized and implemented in the standard library:

```
let array1: [i32; 4] = [1, -2, 3, -4];
let byte_array: [u8; 5] = [1, 2, 3, 4, 5]; // u8 is a 8-bit unsigned integer
let array_of_tens: [i32; 20] = [10; 20]; // contains the number 10 twenty times
let empty_array: [i32; 0] = [0; 0]; // empty array

let mut vector1 = Vec::new();
vector1.push(42);
vector1.push(43);
let vector2 = vec![42, 43]; // short-hand generation with macro
```

Because array sizes are fixed, they are usually not suitable as function parameters (or one would have to implement `fn sum(nums: [i32; 1])` and `fn sum(nums: [i32; 2])` separately, and so on). Instead, slices are typically used. The type of a slice referencing some elements of type `T` is expressed as `&[T]`. Even though slices are proper types, they cannot be allocated themselves. They only reference data that is stored elsewhere (in an array or a vector, for example); either the entire structure or a continuous subrange. Continuing the example above:

```
10 fn sum(nums: &[i32]) -> i32 {
11     let mut result = 0;
12     for num in nums {
13         result += *num; // num is of type &i32 and needs to be dereferenced
14     }
15     result
16 }
17 println!("Array sum: {}", sum(&array1)); // will output -2
18 println!("Sub-Array sum: {}", sum(&array1[1..4])); // will output -2+3-4=-3
19 println!("Vector sum: {}", sum(&vector1)); // will output 85
```

Note that line 15 omits the **`return`** keyword and the semicolon. This is idiomatic in Rust and it is possible because the syntax is expression-based and not statement-based.

2.1.2 Strings

Similar to arrays, the built-in `str` type is for fixed-length strings. Unlike arrays, however, the length is not part of the type and it cannot be expressed in the source code (like `[i32; 10]` for arrays). Strings are mostly used in the form of *string slices* denoted as `&str`. Every string literal is of type `&str`, for example. The dynamically sized, mutable counterpart is `String`. UTF-8 (Unicode Transformation Format) is used as the internal encoding and a `&str` or `String` can be converted to its UTF-8 byte representation with the `.as_bytes()` method:

```rust
let str1: &str = "Hello World";
let mut string1: String = str1.to_string();
string1.push_str(" and Mars");

let utf8_bytes: &[u8] = string1.as_bytes();
let utf8_vector: Vec<u8> = utf8_bytes.to_vec();
let converted_back = String::from_utf8(utf8_vector);
```

Note that `Vec<T>` is a generic type whose elements are of type `T`. Generics work similar to other programming languages.

2.1.3 Structs and enums

There are no classes in Rust and in particular there is no inheritance. Instead, **`impl`** blocks can be used to define static and instance methods for structs and enums. Structs are very similar to most other programming languages. Enums are union types that have multiple variants, each of which can have different fields associated with it.

```rust
struct PhoneNumber {
    country_code: u32,
    area_code: String,
    number: String,
}
enum TANMethod {
    iTAN,
    smsTAN { phone_number: PhoneNumber }, // alternatively unnamed: smsTAN(PhoneNumber)
}
impl PhoneNumber {
    pub fn parse(string_representation: &str) -> Result<PhoneNumber, InvalidNumberError> {
        // omitted
    }
    pub fn to_string(&self) -> String {
        format!("+{} {} {}", self.country_code, self.area_code, self.number)
    }
}
```

Note that the `parse()` method is static because it does not take a `self` parameter, whereas the `to_string()` method takes a `&self` parameter and is thus an instance method with read-only access (similar to `const` methods in C++). Hence, they are called as `PhoneNumber::parse(...)` and `instance_variable.to_string()`, respectively. If an instance method wants to modify its instance, it needs to declare a `&mut self` parameter. The `Result<..>` type is part of the error handling, see below.

2.1.4 Error handling

Rust does not have exceptions or any other language-level feature to throw and catch errors—the only built-in feature are *panics*, which are uncatchable and always terminate the current thread. Nevertheless, there are carefully crafted types and macros to aid error handling. The information that an operation failed needs to be passed back as part of the return value, and the `Result<T, E>` is the idiomatic way to do so. It can either be `Ok(T)` and contain the result of type `T`, or it can be an `Err(E)` with additional error information of type `E`. Upon receiving one of these enum values, the caller can use the **`match`** keyword to react to errors. For example, the `PhoneNumber` parser above would be used as follows:

```
let phone_number = match PhoneNumber::parse("+49 123 4567") {
    Ok(number) => number, // simply pass out the value
    Err(e) => { println!("Error {:?}", e); return } // abort
};
println!("Parsed number: {}", phone_number.to_string());
```

However, there are a few easier ways that require less boilerplate code. When any error must be caused by a bug in a program, or when the program is just a quick test rather than a productive application, the `unwrap()` method can be used, which essentially returns the nested result or panics if an error occurred:

```
let phone_number = PhoneNumber::parse("+49 123 4567").unwrap(); // panics if parsing failed
```

Alternatively, if the calling function returns a `Result` itself, it can automatically hand through the error to its caller (and possibly convert it) using the `try!` macro:

```
fn get_area_code(input: &str) -> Result<u32, InvalidNumberError> {
    Ok( try!(PhoneNumber::parse(input)).country_code )
}
```

The macro will **`return`** early with an error if its argument is an error, otherwise it passes out the unwrapped value.

In addition to `Result<T, E>`, there is the **`enum`** `Option<T>` which does not contain error information. More importantly, return values of this type can safely be ignored, whereas ignoring a `Result` triggers a compiler warning. Throughout this thesis, `unwrap()` is used when error handling does not play a role.

2.1.5 Macros and annotations

The `vec!`, `format!`, `println!` and `try!` statements in the code above call macros instead of functions—macro names end with an exclamation mark. Although these and other macros occur regularly, it suffices to know that these calls are evaluated and replaced with other code by the compiler, so the actually executed code looks different. Macros can be used to generate all kinds of Rust code including entire functions and modules.

Another frequently used Rust feature that does not play a significant role in this thesis are annotations. They use the `#` character and square brackets and can be placed on any language element. For example, a `Result` value cannot be ignored, as explained above, because the type is annotated with *`#[must_use]`*.

2.1.6 Ownership and borrowing

The key to Rust's memory safety guarantees is the way it handles ownership of resources. There is no garbage collector and while reference-counted pointer types (e.g. `Rc<T>`) with a runtime overhead do exist, it is preferable to use regular references and temporarily *borrow* them to functions that need them. This unique feature is at the same time responsible for the most annoying compiler warnings—especially Rust beginners initially struggle a lot with the *borrow checker* before getting acquainted with it.

Generally, the ownership system does not apply to copyable types, including all primitive types. For all other types, the **`let`** binding where an instance is assigned owns the data. From there it can be borrowed mutably (`&mut name`) or immutably (`&name`). At every point of the program execution, there can only be a single mutable reference *or* an arbitrary number of immutable references, conceptually similar to read-write locks.

```
let mut data: Vec<i32> = vec![1, 2, 3];
let immutable_ref: &Vec<i32> = &data;
let_me_borrow(&data);
let_me_borrow(immutable_ref);
fn let_me_borrow(data: &Vec<i32>) {
    // omitted
}

let_me_modify(&mut data); // error: cannot borrow because also borrowed as immutable
fn let_me_modify(data: &mut Vec<i32>) {}
```

Instead of borrowing a value, its ownership can also be passed away (into a function, to another binding, etc.) by using neither `&` nor `&mut`:

```
let mut data = vec![1, 2, 3];
fn let_me_take_ownership(data: Vec<i32>) {}

data.push(4); // works fine
let_me_take_ownership(data);
data.push(5); // error: use of moved value
```

Encapsulating types like vectors or strings offer methods like `.as_ref()`, `.as_mut()` or `.as_mut_slice()` to give access to their internal data structures in the form of (mutable) references or slices.

2.1.7 Traits

Traits are an integral but also very complex part of Rust's type system. A trait defines a set of functions that every implementation of the trait has to provide, similar to interfaces in other languages. Trait functions can be static or instance methods, and they can have a default implementation. Usually, an `impl TraitName for Type {...}` block is used to provide a trait implementation for a type, though there a couple of tricks to make implementing traits easier, which are not relevant to this thesis.

Generic functions and generic types can specify trait bounds on their generic arguments. As an example, consider this function which accepts any data as input that can be converted to a `&[u8]` slice:

```
pub trait AsRef<T> { // simplified definition from std::convert
    fn as_ref(&self) -> &T; // &T will be &[u8]
}
impl AsRef<[u8]> for PhoneNumber {
    fn as_ref(&self) -> &[u8] { self.number.as_ref() }
}
fn do_something_with_some_bytes<T: AsRef<[u8]>>(data: T) {
    let the_bytes: &[u8] = data.as_ref();
}
do_something_with_some_bytes(PhoneNumber::parse("+49 000 123456").unwrap());
```

This pattern is common to account for the lack of method overloading in Rust.

2.1.8 Build system

Modules are containers for types, implementations and functions. They are at roughly the same hierarchical level as namespaces in C++ or packages in Java and are simply declared with the **mod** keyword:

```
mod outer_module {
    pub struct SomeType { ... }
    pub mod inner_module1 {
        use super::SomeType;
        pub fn some_function(x: SomeType) { ... }
    }
    mod inner_module2 {} // empty module
}
mod some_other_module;

fn main() {
    use self::outer_module::inner_module1::some_function;
    let some_instance = outer_module::SomeType { ... };
    some_function(some_instance);
}
```

Note that `some_other_module` is only declared, which delegates the module definition to another (accordingly named) file and allows for modularization on the file level. **`pub`** makes members available outside the module and **`use`** statements import names to avoid repeating the fully qualified names.

Multiple modules form a `crate`, which roughly corresponds to a library in C++ or a JAR archive in Java. Dependencies have to be declared in the top-level file: **`extern`** `crate some_useful_crypto_library`. Rust's package management and build tool is called *cargo* and crates are made available to others through http://crates.io/, where cargo automatically downloads them when referenced from another crate. Meta information about a crate and its dependencies are declared in the `Cargo.toml` file.

Cargo uses semantic versioning,[1] which divides version numbers into three parts: major, minor and patch. The major version must be increased when the API changes in an incompatible way, that is, when a program written for the previous version could fail to compile or properly execute with the new version because of the changes. When designing an evolving API, it important to understand that not all API changes are breaking changes (adding new items is mostly unproblematic) and that breaking behaviour changes can occur without API changes if the contract of the interface changes.

2.1.9 Rustdoc

Rust code is documented with comments starting with a triple slash (`///`) in every line, which can be placed in front of modules, functions, structs, traits, and so on:

```
/// Returns the larger of two bytes.
///
/// # Example
/// `max_u8(42, 43)` will return `43`
fn max_u8(a: u8, b: u8) -> u8 { if a > b { a } else { b } }
```

In contrast to Javadoc, there is no special Rustdoc syntax because Markdown is used instead. The `rustdoc` tool is used to generate Hypertext Markup Language (HTML) files from the source files (similar to Javadoc and others), which can be viewed locally or hosted online for convenience. Docs.rs automatically provides the documentation for all crates published on crates.io.

2.1.10 Toolchains

Because Rust code compiles to native executables, it needs to be compiled for the right target architecture and operating system (OS). On Windows in particular, there

[1] http://semver.org/ (2016-11-10)

are two distinct toolchains that must not be mixed up: the one based on Microsoft's Visual C++ environment (MSVC, called `-msvc` in Rust) and the other based on the GNU Compiler Collection (GCC, called `-gnu` in Rust). Some libraries can only be compiled with the `-msvc` toolchain or the `-gnu` toolchain, though most libraries support both. Additionally, there are different relevant versions of the compiler and build tools (`stable` and `nightly`), and it can be important to be able to switch between those for experiments. Therefore, it is advisable not to install a single variant/version of the toolchain, but to use `rustup`[2] instead.

2.2 Cryptographic foundations

This section explains the concepts in cryptography which are relevant to this thesis. Please refer to a cryptography textbook (e.g. the one by Stallings [Sta06] or Ferguson, Schneier and Kohno [FSK10]) or the respective papers for details, implementations and further concepts not discussed here.

2.2.1 Pseudorandom number generation

Random numbers play a central role in cryptography: any secret like a private or shared key would be compromised if an attacker could compute it deterministically. However, truly random numbers are expensive to obtain in a computer, as they usually originate from a non-deterministic physical process like a coin flip, radioactive decay or atmospheric noise.[3] Pseudorandom number generators (PRNGs) overcome this difficulty by computing sequences of pseudorandom numbers starting from a given *seed* value.

Albeit deterministic, PRNG computations are scrambled enough to guarantee certain quality criteria. The generated numbers should be distributed as evenly (or as closely to the desired distribution, e.g. the normal distribution) as possible, that is, all possible values should occur with (almost) the same probability. For cryptographic applications, this property alone does not suffice: it must additionally be impossible for an attacker to correctly guess the internal state of the generator or the subsequent numbers after having observed many produced numbers. Such a generator is called *cryptographically secure* (CSPRNG).

Even a CSPRNG can be attacked if its seed value can be guessed. To prevent this, most OSs provide facilities for secure random number generation seeded with various low-level values like the exact time, temperature sensor values and hardware identifiers,

[2] https://www.rustup.rs/ (2016-12-16)
[3] https://www.random.org/ (2017-01-12)

which are not normally available outside the machine. Rust has an official `rand` crate[4] for its random number generators (RNGs) and offers, among others, an `OsRng` to read from the operating system's secure generator.

2.2.2 Nonces and IVs

A *nonce* is a "**n**umber used **once**."[5] That is, a new nonce value must be generated for every encryption/hashing/authentication operation. While not always strictly required, random numbers can be used as nonces. Another approach is a counter: use some start value x for the first operation, then $x + 1$, then $x + 2$, ...

Initialization vectors (IVs) are inputs to cryptographic algorithms (typically symmetric ciphers, see section 2.2.6) which must always be nonces and sometimes unpredictable (i.e. not a counter). In cryptographic APIs, the terms *nonce* and *IV* are often used interchangeably.

2.2.3 Hashing

A *hash function* deterministically maps an input of arbitrary length to a fixed-length output called the *hash value* or *digest*. Although hash functions formally have a single input, multiple pieces of input data can be concatenated and hashed together. A good hash function produces evenly distributed hash values that fill the entire output space and are seemingly random (small changes in the input lead to large, unpredictable changes in the output).

As with random numbers, hash functions need to fulfil special requirements to be suitable for cryptography. For a *cryptographically secure* hash function, it must be infeasible[6] to compute the original input from a hash value, or another input that maps to the given hash value, or two inputs that lead to an arbitrary but common hash value.

Well-known cryptographic hash functions include the Secure Hash Algorithm (SHA) family (SHA-1 is deprecated, SHA-2 and SHA-3 are unbroken) and MD5 (Message-Digest Algorithm 5), which is widespread despite being broken for decades.[7]

[4] https://doc.rust-lang.org/rand/rand/ (2017-03-23)

[5] This mnemonic is not the etymologic root of the term.

[6] A computational problem is *infeasible* if a solution theoretically exists but is provably impossible to compute within a reasonable time.

[7] In this thesis, an algorithm is said to be *broken* when a *computationally feasible*[6] attack has been published that circumvents the security guarantees of the respective primitive (e.g. it computes the original input from a hash value). Other definitions of the term *broken* only require a *theoretical* attack faster than a brute force attack, which leads to *deprecation* in the sense of this thesis.

2.2.4 MAC

A message authentication code (MAC) has similar properties as a hash value in that small input changes lead to unpredictable output changes, making it computationally infeasible to find an input for a given output. Unlike hashes, MACs can only be computed with a secret key, i.e., to generate and verify a MAC value, the sender and receiver need to share a key.

This makes MACs suitable for authentication: the receiver computes the MAC for the received data and compares it with the received MAC value (sometimes called *authenticator* or *tag*). If the two values match, the sender must have known the secret key to generate the MAC, which proves the sender's identity assuming that the key was not compromised.

The most popular MAC algorithms are keyed-hash message authentication codes (HMACs), which apply a secure hash function in a special way. The resulting algorithms are named after the used hash function, e.g., HMAC-SHA-256 if SHA-256 is used as the hash function. Another popular MAC is Poly1305.

2.2.5 Secure password hashing and key derivation

Storing passwords for user authentication bears the risk of an attacker stealing the database with all the passwords (if they are encrypted, the attacker additionally has to steal the key). It is therefore common practice to store a hash value of the password rather than the password itself. The hash function guarantees that collisions are unlikely, but if two users have the same password, their hash values are also identical, which gives the attacker additional information: the most common passwords are probably trivial ones and can be guessed. As a countermeasure, some random data called *salt* is provided as additional input and stored in the database next to the hash value. A different random salt must be used for every hashed password.

Even though the salt prevents the use of pre-computed rainbow tables [Oec03], this scheme is still vulnerable to brute-force attacks, as hardware is getting cheaper and more powerful. Thus, it is essential to use a deliberately slow hash function, which is one of many usage scenarios for key derivation functions (KDFs) like bcrypt, scrypt or the Password-Based Key Derivation Function 2 (PBKDF2), for which fast implementations are impossible. The HMAC-based key derivation function (HKDF), on the other hand, is not suitable for password hashing despite being a KDF, but it can be useful for other KDF use cases like key expansion, which are not relevant to this thesis.

Password hashing occurs in many applications and it is not trivial to get right. To keep developers from having to reinvent the wheel, many platforms and libraries, especially in

web development, offer functions that use an appropriate algorithm and take care of the salting. They only produce one output which contains the salt and the hash value in a special format. Some even include information about the used algorithm to solve another engineering problem with password hashing: when an algorithm becomes deprecated, applications have to switch to another one, but they cannot switch immediately because old passwords are still hashed with the old algorithm. So the application needs to keep track of the used hash algorithm for every hash value.

2.2.6 Symmetric encryption

Encryption encodes a given message (called *plaintext* or *cleartext*) such that the produced ciphertext can only be decrypted by someone in possession of the right key. With symmetric encryption, this key is the same as the one used for encrypting the original message and is called *secret key*. There are various applications for symmetric encryption. A single user can encrypt data to store it securely and read it again later (encrypted file systems, password managers, etc.). A sender can encrypt a message before sending it over an insecure connection to a receiver who possesses the right key, denying any eavesdropper access to the message contents (see figure 2.1). In this scenario, the sender and the receiver need to share the secret key beforehand by meeting in person, using a trusted channel like the telephone or another encrypted connection, deriving it from another secret value or using a cryptographic key exchange technique.

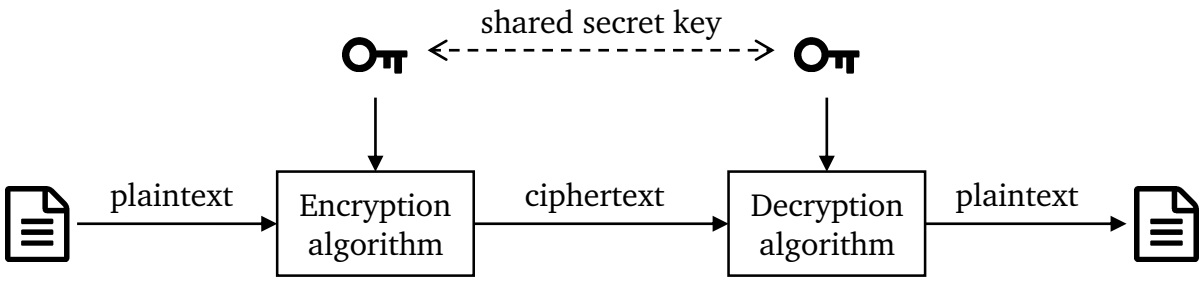

Figure 2.1: Data flows for symmetric encryption

The algorithms that implement symmetric encryption are called *symmetric ciphers* and can be categorized as *block ciphers* and *stream ciphers*. These two kinds of ciphers are operated in different ways (and hence need different APIs). A stream cipher produces a *keystream* from the given key, which can then be combined with the plaintext/ciphertext (using the bitwise `xor` operation) to obtain the ciphertext/plaintext, that is, the same algorithm and the same keystream can be used for both encryption and decryption. Because the keystream never ends naturally, arbitrarily long messages can be encrypted.

A block cipher, on the other hand, can only encrypt/decrypt a single, fixed-size block. In order to process longer texts, there are various *modes of operation* that combine multiple block encryptions in different ways. The Electronic Codebook (ECB) and the Cipher Block Chaining (CBC) modes encrypt one block at a time, though the former encrypts them independently and thereby leaks information about the plaintext.[8] Because the ECB and CBC modes can only encrypt messages whose length is a multiple of the block length, most messages need to be padded to reach an appropriate length. There are multiple padding schemes; the most commonly implemented one for symmetric encryption is PKCS#7 from the Public-Key Cryptography Standards (PKCS). The Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) modes turn the block cipher into a stream cipher, that is, they generate a keystream. In the case of OFB and CTR, the keystream does not depend on the input (plaintext or ciphertext) and can be pre-computed. The CBC, CFB, OFB and CTR modes require a nonce as an IV, and CBC becomes insecure if the IV is predictable.

Well-known block ciphers include the deprecated Data Encryption Standard (DES) and its secure successor, the Advanced Encryption Standard (AES). Well-known stream ciphers are the insecure RC4 cipher, Salsa20 and its newer variant ChaCha20.

2.2.7 Authenticated encryption and AEAD

Unauthenticated encryption as described in the previous section is considered harmful. One reason is a popular misunderstanding of the security guarantees made by the *encryption* primitive: only someone in possession of the right key can read the message, but encryption alone does *not* prevent the message from being changed (blindly) by an attacker [Par15]. This mistake is not limited to layman cryptographers but has been made in the design of Kerberos version 4 [YHR04], for instance. Another reason are side-channel attacks to break the encryption itself (i.e. allow the attacker to recover the plaintext), which are possible when the receiver reacts differently to valid and invalid messages [BU02].

Adding authentication solves both problems: tampered messages are rejected before decryption, preventing any side-channel attacks on the decryption process and guaranteeing that the message was created by someone who has the secret key. In 2002 already, Black and Urtubia [BU02] argued that authenticated encryption (AE) should generally be used, especially since the cost is "quite small," and Rogaway [Rog02] introduced the concept of additional "associated data" (AAD or associated data (AD) for short) that is

[8] The weakness of the ECB mode is nicely demonstrated by the three Tux images on Wikipedia: https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=758793932#Electronic_Codebook_.28ECB.29

sent unencrypted but authenticated along with the main message. The resulting kind of encryption is called authenticated encryption with associated data (AEAD). When using AE(AD), the user has to transmit the authentication tag along with the ciphertext and provide it to the decryption function for verification (see figure 2.2). A nonce prevents replay attacks and has to be supplied to both encryption and decryption.

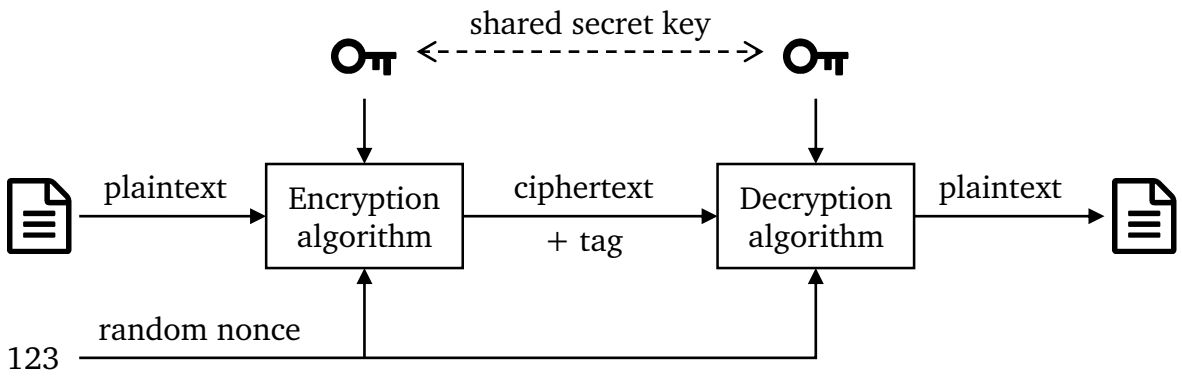

Figure 2.2: Data flows for authenticated symmetric encryption

AEAD can be implemented by combining a symmetric cipher (see the previous section) with a MAC, e.g., ChaCha20-Poly1305.[9] As an alternative, there are various block cipher modes that provide authenticated encryption (sometimes with associated data), like the popular Galois/Counter Mode (GCM). Note that AEAD is still not perfectly secure, as many implementations including AES-GCM become totally insecure when a nonce is reused—with random nonces, this is unlikely but possible. Nonce-misuse resistant AEAD ciphers exist but are not yet widespread.

2.2.8 Digest comparisons

Digests, hash values, MAC values and tags are usually verified by computing the expected *correct* value and comparing it against the received one. A straightforward implementation is prone to use comparison functions provided by the language. In Rust in particular, there is the `==` operator which internally calls the `PartialEq.eq()` function. Most implementations optimize for performance. Comparing byte by byte, they abort with a negative result as soon as the first mismatch occurs. An attacker can measure the response time (many times to gain statistically significant results), infer how many bytes from the beginning already match and progressively guess the entire digest.

[9] https://tools.ietf.org/html/rfc7539 (2017-01-20)

To prevent these timing attacks, digest comparisons need to be done in constant time, that is, the comparison function always compares all bytes before returning a result. While this is crucial for scenarios where the digest is transmitted itself (MAC and AEAD scenarios), it is considered a best practice in general and applied to other digest comparisons, as well.

2.2.9 Primitives vs. protocols

Different authors use the terms *cryptographic primitive*, *cryptosystem* and *cryptographic protocol* to mean slightly different things. Hashing and encryption are clearly primitives, ElGamal encryption is clearly a cryptosystem and TLS is clearly a protocol. On the other hand, the RSA cryptosystem (named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman) can also be viewed and used as a primitive.

The term *primitive* is particularly confusing. Take authenticated encryption as an example. As mentioned in the previous section, it can be implemented with a certain block cipher mode for AES, which is certainly a (single) primitive. However, one can also combine two primitives (encryption and authentication) to implement AEAD. The term *primitive* suggests atomicity, i.e., a primitive would be the smallest building block and not further decomposable. So AEAD could both be a primitive and not be a primitive.

As this thesis is about API design and how to deliver primitives to users, it is practical to ignore the atomicity aspect (and in fact all implementation aspects) and focus on the *building block* aspect instead: A primitive is a function which takes one or a few inputs to compute usually one output and makes certain provable security guarantees. Mere utility functions do not make such guarantees and higher-level cryptosystems and protocols are more complex than a function. Note that a primitive can also consist of a small set of functions, e.g., reverse or key generation functions. In particular, AEAD, HMAC and salted password hashing are cryptographic primitives in this sense, even though they are each built on another primitive, as explained in the previous sections.

Cryptosystems and cryptographic protocols are on higher levels of abstraction than the one covered in this thesis. The term *protocol* is consistently used for any higher-level cryptographic construction that uses primitives but is not a primitive itself, including what other authors distinguish as cryptosystems.

3 Related work

This thesis touches a number of research areas. First, the thesis analyses the crypto ecosystem of Rust using various research methods. To the best of my knowledge, there is currently no research that analyses the crypto ecosystem of a programming language or its platform. In non-academic writing, the term *crypto ecosystem* is predominantly used in the context of crypto currencies, though a few already use it in the same sense as this thesis. Please see the following chapter for related usages of the term *ecosystem*.

The social components in the crypto ecosystem of Rust, namely the developers and their interactions, are not necessarily specific to Rust or cryptography, but resemble open source software development in general. Section 3.1 briefly introduces existing surveys and other studies of open source projects. Regarding the research methods themselves (survey, GitHub mining, experiments), there is obviously an extensive body of research that uses them and several works that deal with the methods themselves and how to apply them correctly. I only reference these works when I adopt their suggestions and in the discussion of the threats to validity (e.g. in section 5.5.6).

The second half of this thesis is concerned with the usability and misuse resistance of the cryptographic libraries in Rust. Section 3.2 briefly introduces a few works about API design that contain general concepts applicable to Rust. There are even some blog posts and guidelines for API design in Rust referenced in section 3.2.1. More importantly, however, this thesis concerns *cryptographic* APIs, which hide relatively complex algorithms and must be used correctly to avoid vulnerabilities. This characteristic makes them similar to other cryptographic (user) interfaces, which have been extensively studied—section 3.3.1 gives a brief overview. Section 3.3.2 reviews studies about crypto API usability in detail and section 3.3.3 introduces two well-known libraries which implement usable crypto APIs.

3.1 Open-source projects

Several researchers have surveyed contributors to open-source projects to find out about their motivation, invested efforts and backgrounds. For example, Hertel, Niedner and Herrmann [HNH03] survey Linux kernel contributors, Lakhani and Wolf [LW03] survey contributors on SourceForge.net, and David and Shapiro [DS08] and Ghosh et al.

[GGKR02] each got responses from thousands of diverse open source developers worldwide. Crowston et al. [CWHW12] review the literature on open source development and, among other insights, distill the following points: Developers are primarily European and North American, motivated by reputation and rewards (like career development), joy of programming, joy of learning and user needs. Stack Overflow has started a yearly survey among its users in 2011, which also covers backgrounds (including age and gender) and motivation [Sta16; Sta17].

Besides the contributors themselves, the most interesting data points are their contributions and other interactions online. Quantitative analyses by Lima, Rossi and Musolesi [LRM14] and Chełkowski, Gloor and Jemielniak [CGJ16] find that they resemble power-law distributions, which are also described with the "1-9-90" rule [CGJ16] and the Pareto principle, though the applicability of the latter could *not* be confirmed for many projects [YMK+15].

Today and particularly in the Rust crypto ecosystem, most interactions happen on GitHub, the leading social repository hosting platform, which provides convenient APIs for automated, large-scale analyses like the ones mentioned above [LRM14; YMK+15]. When analysing the mechanics of GitHub's pull requests, Gousios, Storey and Bacchelli [GSB16] find that many developers use additional communication channels like e-mail and Internet Relay Chat (IRC). Therefore, when studying discussions on GitHub, it needs to be kept in mind that discussions might also take place elsewhere. Kalliamvakou et al. [KGB+16] identify pitfalls and perils that arise in analyses of GitHub data, including the one mentioned before, and discuss avoidance strategies.

3.2 API design and usability

There is a large body of literature about the usability of general APIs. Every book about API design also covers API usability. Those works are not specific to cryptographic APIs and they usually focus on a particular, widespread programming language. This section only references a small selection, which contains concepts, solutions and recommendations that can be transferred to the crypto ecosystem of Rust.

Tulach [Tul08] advocates the use of APIs to achieve more *cluelessness*, which means that it should be possible for developers to understand only very little about the inner workings of the libraries and tools they use. Just like a smartphone user does not need to know programming, electrical engineering or physics, the user of a cryptographic library should only need a minimal knowledge of cryptographic primitives and of the library's implementation. Tulach uses the term *empirical programming* for an approach to software implementation where the programmer does not study the documentation of an API and does not try to actually understand it, but rather just calls a promising

method and observes what happens. The programmer then iterates such experiments until the program works. Instead of condemning this programming practice, Tulach recommends to support it by making APIs *self-documenting*.

Wurster and van Oorschot [WO08] take cluelessness a step further: Any library, platform or tool that a developer uses should be built as if the developer was an *enemy* similar to an attacker instead of relying on the developer to be informed and make the right choices. The paper covers secure programming in general, but interestingly they use a cryptographic misuse [Wu05] to illustrate that usable and learnable APIs are important. "[P]roviding cryptographic algorithms alone does not ensure security." [WO08, sec. 3.4] They also point out that usable APIs do not only prevent security bugs but also motivate developers to use a security library in the first place. One of their two main proposals, namely *unsuppressible warnings*, could also be beneficial for crypto libraries.

Robillard [Rob09] reports on a survey about API learnability among Microsoft developers and finds that a *well-structured documentation* and *suitable code examples* are important because they help developers understand the *high-level design* of the API.

3.2.1 Rust API design

APIs are code themselves and should therefore follow general coding guidelines. The `rust-guidelines` repository[1] by Rust core team member Aaron Turon is discontinued, and after the style guidelines were temporarily part of the core documentation,[2] there is now a separate process for code formatting requests for comments (RFCs), the results of which are available from the `fmt-rfcs` repository.[3] This repository naturally focuses on code formatting and code style, whereas the previous styleguide also covered design topics including some API design aspects. Albeit unfinished, already deprecated and possibly outdated soon, the latest version[2] contains relevant considerations on constructor design and the builder pattern, placement of functions/methods, function signature design, strong typing, type conversions, module organization and error handling.

An unofficial repository collects Rust design patterns,[4] some of which apply to APIs. Hertleif [Her16b] covers several design techniques for "elegant" APIs. Martins [Mar16] writes about conversions between different data types, which allows APIs to accept data in various formats without declaring redundant variants of the same function.

[1] https://github.com/rust-lang/rust-guidelines (2017-03-23)
[2] https://doc.rust-lang.org/1.12.0/style/ (2017-03-23)
[3] https://github.com/rust-lang-nursery/fmt-rfcs/blob/master/guide/guide.md (2017-03-23)
[4] https://github.com/rust-unofficial/patterns (2017-03-16)

3.3 Crypto usability

3.3.1 End-user crypto usability

Starting with the widely known paper "Why Johnny Can't Encrypt" by Whitten and Tygar [WT99] in 1999, there has been plenty of research about the usability of cryptographic applications. For example, two follow-up studies by other researchers in 2006 and 2015 find that modern Pretty Good Privacy (PGP) tools are still too difficult to use. The field covers topics such as e-mail and messaging clients, security indicators in browsers, multi-factor authentication and so on.

This research is usually concerned with the user interface and is therefore not immediately useful for design considerations for cryptographic libraries, as those are used by other programmers through an API. Nevertheless, there are some interesting parallels, especially when it comes to research methods and high-level ideas. For instance, user tests like the one by Whitten and Tygar [WT99] to evaluate PGP can analogously be conducted with Rust programmers who are tasked to perform certain tasks with a given Rust crypto library. Section 7.2 reports on such an experiment conducted with Rust beginners at the University of Osnabrück.

As another example, Troutman [Tro14] questions the idea of "usable cryptography" itself and points out that end users are not interested in encrypting messages for the sake of encrypting, but they want to communicate securely: "Johnny can't encrypt because Johnny never wanted to encrypt" [Tro14]. Zurko and Patrick [ZP08, sec. 4] formulate the same idea and also conclude that a product design perspective is essential to solve this problem. When applied to crypto API usability, which is discussed in the next section, the general idea still holds: An application developer's goal is not to encrypt a data packet or to validate a certificate, but to communicate securely with the server, and cryptographic libraries should be designed with such goals in mind.

3.3.2 Crypto API usability

Georgiev et al. [GIJ+12] analyse Secure Sockets Layer (SSL) certificate validation code in a range of applications and, if vulnerable, investigate the cause of the bug. They find that the libraries themselves are correct "for the most part," but developers often misunderstand the APIs, which is the "primary cause" for vulnerabilities. Consequently, they call for more high-level and more formalized APIs, consistent error reporting, safe defaults and better documentation. They also find that many application developers disable certificate validation altogether, which "appears to be the developers' preferred solution to any problem with SSL libraries" [GIJ+12, sec. 10].

In line with Wurster's idea of treating the application developer as an enemy, Fahl et al. [FHP+13] suggest that the certificate validation should be implemented by the OS, after having found similar problems with custom validation code or disabled validation in previous work [FHM+12]. The app itself could merely configure the validation by providing a trusted root certificate, for example. If in doubt, the OS could directly notify the user about possibly insecure connections and thus make it impossible for apps to silently skip certificate validation. This approach only works on "appified" platforms, however, where the OS can actually impose such restrictions. The low-level nature of Rust does not allow that, but the work of Fahl et al. [FHP+13] nevertheless contains some valuable insights for Rust TLS libraries about the usage of certificate validation.

Forler, Lucks and Wenzel [FLW12] demonstrate how the Ada language and compiler can be used to design an API that prevents nonce reuse and plaintext leaks. More specifically, they ensure that every generated nonce can only be used once and that the state of the nonce generator cannot be copied. Similar constructions could be possible in Rust (see section 8.8). Their API design also prevents another kind of misuse where the caller reads a (partially) decrypted plaintext even though the authentication has failed or has not even been performed yet.

Egele et al. [EBFK13] implement a static analysis tool called CRYPTOLINT to automatically discover common security flaws such as ECB mode, constant keys, constant salts and constant PRNG seeds. CRYPTOLINT is implemented for the Android platform and only analyses the Dalvik bytecode. In addition to the tool, which detects mistakes after they have already been made and which needs to be incorporated into the build process explicitly, they also discuss measures to prevent security flaws on the API or compiler level. Like many others, they propose getting rid of insecure defaults (the ECB mode in Java's `Cipher.getInstance()`, for example), either by replacing them with more secure values or by removing the defaults and forcing every caller to specify the respective parameter explicitly. While the former solution breaks backward compatibility, the latter is in conflict with Tulach's cluelessness concept. To mitigate that, Egele et al. recommend improving the API documentation: explicitly state all defaults and suggest sane choices when there is no default. Finally, they propose to build APIs which enforce their semantic contracts by the means of the API itself. For example, the criteria for a proper initialization vector (being unique and non-predictable) should not only be documented in the description of the function which accepts them, but their violation should also lead to a compile error. At least in Java, this requires compiler support or a tool like CRYPTOLINT to discover violations. [EBFK13, sec. 7]

Lazar et al. [LCWZ14] investigate 269 vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database and find that the majority is caused by application code misusing the properly implemented crypto libraries. Among the causes are insecure

defaults, disabled or insufficient certificate validation, weak/obsolete ciphers, hard-coded keys and weak random numbers. They discuss a range of prevention techniques for such vulnerabilities, only one of which is aimed at the usability of crypto libraries: APIs should be more high-level and, for example, provide authenticated encryption as a single function. Keyczar is named as a positive example. [LCWZ14, sec. 3.1]

Das and King [DK14] define 7 properties to determine how safe a cryptographic library is and apply them to 6 libraries for the most popular programming languages. On the one hand, these 7 properties can also be evaluated on Rust crypto libraries, making them comparable to other languages' libraries. On the other hand, the discovered problems with the non-Rust libraries can be used to check Rust libraries for similar issues (see section 7.3 for a partial execution of this idea).

Nadi et al. [NKMB16] empirically investigate the obstacles that developers face when using Java's standard crypto API. They analyse Stack Overflow posts, GitHub repositories and conduct developer surveys. For API designers, they recommend improving the documentation, building more high-level, *task-based* APIs and name NaCl and Keyczar as positive examples.

Green and Smith [GS16] develop 10 "principles for creating usable and secure crypto APIs," which are more or less directly applicable to Rust crypto libraries (see section 7.3). Besides several recommendations that have already been made in earlier works, they suggest adding a "testing mode" feature that restricts weakened cryptography for testing purposes (e.g. hard-coded keys, weak random numbers or disabled certificate validation) to the developer's machine using a specific machine ID (see section 5.7.2). Their main statement "developers are not the enemy either" is derived from the well-known statement "users are not the enemy." Just like users cannot be blamed for not understanding the difficult concepts that applications confront them with, Green and Smith argue that application developers cannot be blamed for misunderstanding the APIs of crypto libraries. While this reasoning is in accordance with all previously mentioned authors, the formulation should not be confused with the grammatically opposite formulation "the developer is the enemy" by Wurster and van Oorschot [WO08]. Despite the seemingly contrary meaning, Wurster and van Oorschot actually take the same line as Green and Smith, because they recommend library developers to view application developers *as if they were* enemies, thus anticipating they will (deliberately) make every imaginable mistake. Both lines of thought lead to the conclusion that library developers are, to some degree, accountable for misuse of their libraries by application developers.

Besides improving the APIs themselves, there are also approaches to deal with the misuse of APIs that cannot be changed anymore due to backward compatibility. One approach is to improve the documentation and provide proper code samples. Another, more obtrusive solution are automated misuse detectors (also known as *linters*), which are usually based on static code analysis. The tool of Egele et al. [EBFK13] is called

CRYPTOLINT and was already described above. The authors point out their large-scale analysis of Android apps is not the only purpose of the tool: it can also be used by developers to check their code, by app store operators to check submitted apps and by "security-conscious" users to check their downloaded apps. Another example is the PYCRYPTO LINTER by Das and King [DK14], which is a proof of concept they developed to discover nonce reuse, counter reuse and use of the ECB mode. Amani et al. [ANN+16] provide a set of API misuses collected from various sources called MUBENCH, some of which are crypto API misuses, in order to benchmark such automated misuse detectors. For Rust at its current stage of development, misuse detectors are not yet relevant, as the goal must be to design APIs that prevent misuses from happening in the first place.

3.3.3 Crypto libraries designed for usability

Few crypto libraries are designed specifically with API usability in mind. The most well-known ones are NaCl, libsodium and Keyczar, all of which have repeatedly been highlighted by researchers for their good usability.

3.3.3.1 NaCl and libsodium

The Networking and Cryptography library (NaCl) by Daniel Bernstein [Ber09] has three main goals, one of which is to improve usability. It does so by focusing on the most important use cases and supporting those with an extremely simple and high-level API. For example, the "most fundamental" operation, according to Bernstein, is public-key authenticated encryption, and it is provided through a single function called `crypto_box`, though lower-level functions with more flexibility are available, too. These high-level functions are named after the cryptographic services they provide for their user—and not after the primitives they are implemented with. Every high-level function is backed by a default choice of primitives, though there is usually one alternative. To avoid accidental misuse, NaCl excludes any insecure or deprecated algorithms and conservatively chooses a few primitives only. The inherent disadvantage is that the library cannot be used in every scenario because some protocols or other interfaces specifically require implementations of AES-CBC, RSA or other missing, yet secure and popular algorithms.

Because NaCl is not easily portable and packageable, the fork libsodium was created [Den]. Unlike NaCl, which was last updated six years ago (early 2011), libsodium is actively maintained and continually improved. There are bindings to libsodium for numerous languages including Rust (see section 5.3.3).

3.3.3.2 Keyczar

Keyczar differs from other crypto libraries in that it does not implement primitive algorithms itself and it is not a plain wrapper for another existing library. Instead, it comprises a file-based key management system and a rather small set of very high-level primitives.[5] The definition of these primitives includes a custom format for ciphertexts and signatures, which includes "technical" details like the used algorithms, modes, initialization vectors and authentication tags. By including all this information in every ciphertext/signature, Keyczar is able to remove the corresponding parameters and options from its developer-facing API, to make all decisions internally and to read data encrypted/signed with another Keyczar version. Like NaCl/libsodium, Keyczar does not solve all use cases but tries to solve the common ones well.

There are official and mostly equivalent implementations in C++, Java and Python, each of which uses a different lower-level crypto library for the primitive implementations on the respective platform (namely OpenSSL, the Java Cryptography Architecture (JCA) and PyCrypto). Implementations in other languages like Go, .NET, Perl and JavaScript are available unofficially. Unfortunately, Keyczar is not actively developed anymore and has some security issues.[6] It is nevertheless an interesting design concept.

[5] https://github.com/google/keyczar/wiki/KeyczarSummary (2017-03-23)
[6] https://github.com/google/keyczar#known-security-issues (2017-03-23)

4 Ecosystems

This chapter defines the term *crypto ecosystem*, which is used to structure the analysis of Rust's crypto ecosystem in chapter 5, and outlines the crypto ecosystems of other popular programming languages. The term *ecosystem* is widely used to mean many different things, and yet it is usually intuitively understood. Section 4.1 draws from various areas where the term is already used in order to characterize two particular kinds of ecosystems—namely *programming ecosystems* and *crypto ecosystems*—and section 4.2 then defines these terms as well as the populations and interactions within such ecosystems. Section 4.2.3 clarifies some alternative meanings which have to be distinguished from the meanings in this thesis. To put the crypto ecosystem of Rust into perspective, which is later described in chapter 5, the sections 4.3 through 4.6 describe the crypto ecosystems of C++, Java, .NET and Python and briefly summarize a few relevant design decisions made in these ecosystems and their usability implications.

4.1 Deriving a definition

The term *ecosystem* originates in ecology and hence describes natural ecosystems as communities of "living organisms in conjunction with the nonliving components of their environment" [Wik16]. In other fields, the term can have vastly different meanings, though certain characteristics of an ecosystem recur consistently: An ecosystem is made up of *populations* (originally the living organisms, e.g. foxes and rabbits), who are connected through certain *interactions* (e.g. foxes eat rabbits, their roles would be called predator and prey in this case). Moreover, ecosystems form a hierarchy: they can have *neighbours* (e.g. a lake next to a forest), with which they form *higher-level* ecosystems (e.g. all ecosystems on Earth together form the biosphere), or conversely they can have *nested* ecosystems inside them (e.g. a rabbit's gut flora).

These characteristics are also found in technical and digital ecosystems, which is illustrated by the Stack Exchange question "What is ecosystem in IT world?"[1] [sic]. The accepted answer discusses various interconnected ecosystems on different levels:

[1] http://superuser.com/a/553790 (2017-03-23)

hardware, operating systems, programming ecosystems and app platforms. For "programming ecosystems" specifically, the answer names examples of possible populations such as integrated development environments (IDEs), libraries and documentation.

The definition of a *programming ecosystem* used in this thesis follows the same notion. To delimit a programming ecosystem from its neighbours, it is reasonable to use the technical boundaries of its programming language (e.g. Rust) or programming platform (e.g. .NET). Like the populations in a natural forest ecosystem interact more with other individuals *inside* the forest than with the outside, interactions among code written in the same language or for the same platform are much more frequent than with other code. The platform itself is also part of the ecosystem because the libraries interact with it a lot and can even become part of the platform.

German, Adams and Hassan analyse the "R software ecosystem" and use a very similar definition where the *core platform* is surrounded by "a halo of user contributions" [GAH13]. Another definition by Bosch and Bosch-Sijtsema [BB10] extends the ecosystem and includes the human developers as a separate population, in addition to the technical entities (platform, libraries, code, documentation). I follow this decision because the interactions among developers and between developers and libraries are particularly interesting to study and relevant to usability. Loyola and Ko [LK12; LK14] even quantify these interactions and apply the Lotka-Volterra equations, which originally model biological mutualism and competition, to open source ecosystems.

Based on the *programming ecosystem* definition, the *crypto ecosystem* can simply be defined as the subset concerned with cryptography. Because the *users* of (cryptographic) APIs are software developers themselves, I use the term *contributors* to distinguish developers who work on the platform/libraries in the ecosystem itself.

4.2 Definition

4.2.1 Programming ecosystem

A *programming ecosystem* consists of a core *platform* with at least one programming language, the standard library and development tools, all compatible *libraries*, their *code* and *documentation* as well as all *contributors* and *users*.

There are (at least) the following types of interactions in a programming ecosystem (and therefore in its crypto ecosystem):

- Library A can be a *dependency* of library B. Then B is said to *depend on* A, and B is a *reverse dependency* of A. All libraries depend on the core platform.

- A library can be a *fork* of another library.[2]
- A library always *contains* a number of code files and pieces of documentation.
- A library's code can *reference* other parts of the code, e.g., by using functions or types declared there.
- A library's documentation can *reference* another part of the documentation or the corresponding code.
- A library's documentation can be *generated* from the corresponding code.
- A contributor can *contribute* to a library or the core platform. Possible contributions are code in the form of pull requests, code reviews, documentation and bug reports.
- A *user* can *use* a library by making it a dependency of his/her own library or application.

Note that the *contributors* and *users* populations both consist of (human) software developers. Thus, the term *users* and the *use* interaction always refer to using an API, its documentation or other parts of a library rather than an end-user using an application. The developers of such end-user applications are *users* in the programming ecosystem because they use its libraries. The *documentation* population also includes code samples, tutorials, blog posts, etc. besides the generated Rustdoc documentation. See figure 4.1 for an illustration of the main components of a programming ecosystem.

4.2.2 Crypto ecosystem

The *crypto ecosystem* of a programming language or platform is the part of its programming ecosystem concerned with cryptography, that is, all cryptographic libraries, their code and documentation as well as associated contributors and users.

It can sometimes be difficult to determine the degree to which a library is "concerned with" cryptography, so the boundary of the crypto ecosystem is rather blurry. Libraries which primarily implement cryptographic primitives or protocols are clearly in the scope of the crypto ecosystem. The same holds for libraries which primarily offer primitives or protocols by wrapping other libraries' implementations (possibly written in another language).

At the edge of the crypto ecosystem there are libraries which have a different primary focus but still offer APIs that control cryptographic behaviour. A good example are application-layer protocol implementations such as the Hypertext Transfer Protocol (HTTP), the Internet Message Access Protocol (IMAP) and so on, which are not cryptographic libraries themselves but use some TLS implementation to offer the secure

[2] A library is called a *fork* if its development started by copying the source code of the original *upstream* library but continues rather independently and is driven by different developers.

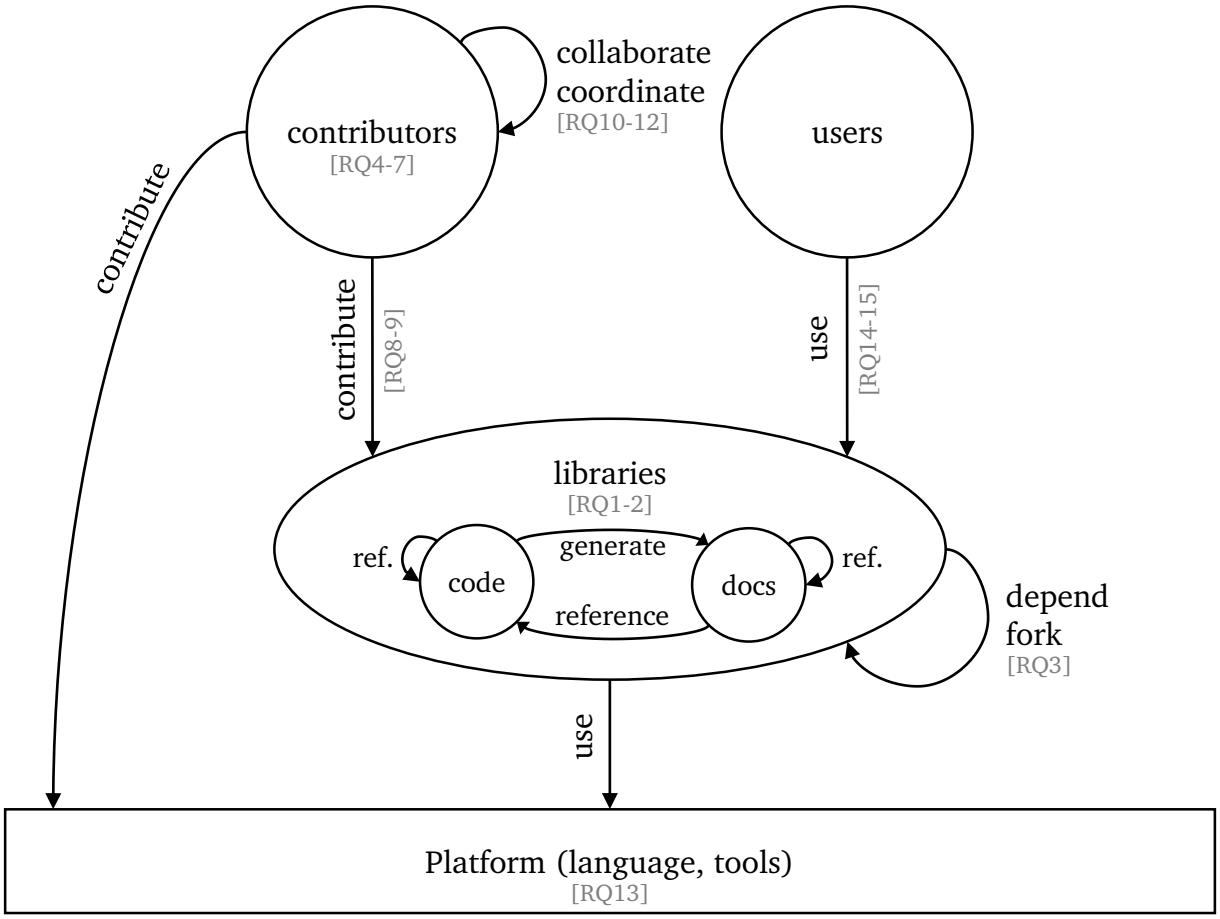

Figure 4.1: Populations and interactions in a programming/crypto ecosystem. The annotated research questions (RQs) can be found in section 5.1 on page 45.

variants HTTPS, IMAPS, etc. Most of these libraries give the library user the opportunity to customize the signature validation process. For example, `curl-rust`[3] is a wrapper for the cURL library and is therefore used to transfer data over various protocols, so its primary purpose is not cryptography. Yet, it offers methods like `ssl_cert`[4] to provide certificates and keys for secure connections.

This thesis intentionally focuses on libraries that implement cryptographic primitives, and all other cryptographic or higher-level protocols are outside the scope of this thesis. Nevertheless, when considering API usability in the Rust crypto ecosystem, it is important to keep in mind that libraries like `curl-rust` are just as well part of the ecosystem, albeit

[3]https://github.com/alexcrichton/curl-rust (2017-03-23)
[4]http://alexcrichton.com/curl-rust/curl/easy/struct.Easy.html#method.ssl_cert (2017-03-23)

at the outer edge. Weaknesses in the design of such APIs have led to severe vulnerabilities [e.g. GIJ+12, sec. 7.1].

4.2.3 Disambiguation

A Google search for "crypto ecosystem" and "cryptographic ecosystem" yields 2.080 and 161 results, respectively (executed in November 2016). Manual investigation of the first 50 results, after which the results become increasingly irrelevant, shows that the term is used to mean different things.

The understanding of a crypto ecosystem in this thesis, as defined above, is always with regard to a certain programming language or platform. The term has been used in this sense before, though very rarely—there are only two occurrences that I am aware of: Mergen [Mer15] writes "Haskell's entire cryptography ecosystem" and means all cryptographic libraries for Haskell. And Arcieri [Arc13] writes about the "WebCrypto ecosystem," which is a set of cryptographic APIs for JavaScript.

In the same article and referencing the same ecosystem, Arcieri writes about "an interoperable cryptographic ecosystem for the web," which highlights a different aspect: this kind of cryptographic ecosystem is defined by compatibility rather than the platform. Compatibility means that the populations in the ecosystem use the same data formats or protocols. For example, all libraries, applications and even devices supporting TLS or Kerberos can be considered an ecosystem. Another example is libsodium [Den], which uses the same custom data format on all supported platforms for interoperability. On the one hand, the many implementations and bindings of libsodium for various programming languages form an ecosystem. On the other hand, each of these implementations is also part of the crypto ecosystem of the corresponding language or platform. So these different kinds of ecosystems are neighbours and even overlap.

Predominantly, however, the term *crypto ecosystem* is used for something else, namely the economical and technical ecosystems that evolve around crypto currencies like Bitcoin. While the "Blake cryptographic ecosystem"[5] only refers to the ecosystem related to BlakeCoin, most others use the term *the crypto ecosystem* to mean the single global ecosystem encompassing all crypto currencies and sometimes even related blockchain applications.

[5] https://blakecrypto.com/ecosystem/ (2016-12-22)

4.3 C and C++

The C crypto ecosystem is the biggest of all crypto ecosystems with quite many libraries for primitives (libgcrypt, libsodium, LibTomCrypt, NaCl, nettle, wolfCrypt), TLS and other protocols (BoringSSL, cryptlib, LibreSSL, GnuTLS, mbed TLS, OpenSSL, wolfSSL; some of these provide access to the underlying primitives, too) as well as a large number of smaller libraries with a few algorithms only. Most reference implementations for cryptographic algorithms and most mature and well-regarded crypto libraries are written in C. Therefore, many wrappers and bindings written in other languages reference libraries from the C crypto ecosystem.

C++ projects can easily compile, link against and call C functions directly, effectively making the C++ (crypto) ecosystem a superset of of the C (crypto) ecosystem. There are three notable crypto libraries implemented in C++ itself (Botan, Crypto++ and Keyczar), though the development of the latter two seems to have stalled.

While libsodium, NaCl and Keyczar are generally praised for their high-level, usable interfaces (see section 3.3.3), the usability analysis by Das and King [DK14, table 3] finds few weak points in Crypto++ and criticizes most aspects of OpenSSL. To my knowledge, the usability of the other libraries has not been systematically analysed.

4.4 Java

Java is not only a programming language but also the basis of various platforms. Therefore, its crypto libraries can be used in different contexts like Android, Java EE server or desktop applications. Since version 1.1 (released in February 1997), the Java platform contains the Java Cryptography Architecture (JCA), which features a service provider interface (SPI). That is, there is a set of interfaces to be implemented by a *cryptographic provider*, whose capabilities are then made available to other programmers through a single common API. The JCA was designed and implemented by then-Sun employee Benjamin Renaud with feedback from many others including Java and API design expert Joshua Bloch.[6] It can be found in the packages `java.security` and `javax.crypto`. Its documentation explicitly lists the design principles of the API—and API usability or misuse resistance are not among them. Instead, the documentation contains a clear and highlighted warning: “this document does not cover the theory of security/cryptography” and “also does not cover the strengths/weaknesses of specific algorithms, not does it cover protocol design.” [Ora16]

[6] https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/guide/security/CryptoSpec.html#Ack (2016-08-14)

The SPI proved to be a practical solution to the U.S. export restrictions: while the API itself was part of the Java Development Kit (JDK) so that application programmers could compile against it, the possibly restricted implementations could be shipped separately. For the same reason, the Java Cryptography Extension (JCE) was not part of the core platform at the time but available as an extension. With version 1.4 (released in February 2002), the JCE's interfaces were integrated and are now considered part of the JCA.

A cryptographic provider is, of course, not part of the JDK itself and it is up to the vendor of a Java Runtime Environment (JRE) to include one. Oracle's official JRE distribution ships with the official JCE provider implemented by Sun (hence called the *Sun-JCE*), but it is disabled by default due to export restrictions and needs to be enabled by installing the "Unlimited Strength Jurisdiction Policy Files." Android, on the other hand, uses the other popular JCE implementation called *BouncyCastle* and modifies it for the Android platform. These modifications have been subject to a lot of criticism because features are missing, the implementation is outdated on older devices and it is difficult for app developers to supply the latest version of BouncyCastle due to class name conflicts with the built-in version. One solution to this problem is the library *SpongyCastle*, which simply renames all *BouncyCastle* packages to resolve the conflicts. The other solution is to use an entirely different library which is not based on the JCA. Both solutions require additional effort from the developer and thus nullify the advantage of having a built-in cryptography architecture.

The usability of the JCA APIs has been criticized occasionally, especially after the Android platform caught security researchers' attention [ANA+15; EBFK13]. In the comparison by Das and King [DK14], it ranks about average. Nadi et al. [NKMB16] find that its APIs are too low-level.

Keyczar, as one of the many alternative crypto libraries in Java, addresses this by building on the JCA and providing a higher-level API (see 3.3.3.2), but it is not actively maintained anymore. Jasypt also advertises[7] easy-to-use, crypto functions based the JCE, but it has not been updated in three years and has been criticized for using non-standard constructions without peer review.[8] There are several bindings for libsodium (NaCl; see section 3.3.3.1), e.g. Kalium, and another high-level crypto library is AeroGear Crypto. Apache Commons Crypto wraps OpenSSL or JCE, alternatively, and exposes them through a common API, though only RNG and symmetric encryption are supported. Besides the named libraries, there area few commercial libraries and many smaller, mostly discontinued libraries for specific purposes.

[7] http://www.jasypt.org/ (2016-09-08)
[8] http://security.stackexchange.com/a/65240 (2017-03-23)

4.5 .NET

The .NET Framework ecosystem encompasses the runtime environment called Common Language Runtime (CLR), an extensive standard library and several programming languages including C#, managed C++, F# and Visual Basic .NET as well as the web framework Active Server Pages .NET (ASP.NET). Since the .NET Framework's first release in 2002, its predominant crypto library is officially developed by Microsoft and shipped in the `System.Security.Cryptography` namespace. It integrates well with the surrounding .NET ecosystem, for example by using the `System.IO.Stream` API for most crypto operations, or by allowing new algorithm implementations to be specified through a configuration file without any code changes [Pil04]. Internally, `System.Security.Cryptography` partially implements cryptographic algorithms in the managed .NET environment and partially wraps the Microsoft Cryptography API (CryptoAPI) or the newer Cryptography API: Next Generation (CNG).

The documentation is extensive and contains many code samples. The API usability has not been analysed scientifically, though Duong and Rizzo [DR11] find problems with key management and unauthenticated encryption in ASP.NET and criticize that "developers still have to figure out on their own how to use cryptographic primitives correctly." This criticism is also applicable to the `System.Security.Cryptography` namespace: the .NET Framework itself lacks authenticated encryption primitives entirely, requiring users to combine authentication and encryption manually, which would be highly error-prone. The official but separate CLR Security project provides extensions for the .NET Framework, and its `Security.Cryptography.dll` provides authenticated encryption in the same API style and with comparable documentation and code samples. The project aims to have its developments integrated in the official .NET platform, eventually, and has influenced the current crypto APIs of .NET already.[9] By open-sourcing the development and letting new features mature before integrating them into the core library, Microsoft benefits from community feedback and peer review and avoids getting stuck with a suboptimal API due to backward compatibility.

Alternative crypto libraries include the C# version of BouncyCastle, bindings to libsodium and a few commercial ones. The inferno crypto library is advertised[10] as "developer-friendly" and "misuse-resistant." It deliberately excludes unauthenticated encryption and, similar to NaCl, only provides a few primitives with an opinionated choice of algorithms to implement them—though the chosen algorithms are different from NaCl. Yet, the documentation is scarce and contains hard-coded keys. Like the other presented crypto ecosystems, the .NET crypto ecosystem comprises a large number of smaller libraries for specific tasks.

[9] http://clrsecurity.codeplex.com/discussions/651423 (2017-03-23)
[10] http://securitydriven.net/inferno/ (2016-10-22)

4.6 Python

Python's standard library contains modules for cryptographic hash functions, HMACs and CSPRNGs.[11] Besides those, there is no officially endorsed solution like in Java or .NET, but one of the oldest Python crypto libraries, PyCrypto by Dwayne Litzenberger, has been the de-facto standard for a long time and reaches more downloads than all other crypto libraries combined, according to the PyPI Ranking.[12] It has not been updated in years, however, and Das and King [DK14, table 3] find many critical points regarding misuse resistance in their comparison with other libraries. For instance, PyCrypto lacks authenticated encryption as a single primitive. The first Google search result[13] for "PyCrypto authenticated encryption" manually combines authentication (HMAC-SHA-256) with encryption (AES-CBC-256) but fails to use a constant-time comparison function for verification. PyCryptodome is an actively maintained fork, provides AE and promises improved APIs.

The second most popular Python crypto library is simply called `cryptography` [KG+] and pioneers a structural concept that divides the entire interface into two levels with different target users: the "recipes" layer contains high-level primitives that are easy to use and require minimal understanding of cryptography, whereas the "hazardous material" layer provides access to the underlying low-level primitives.

Other crypto libraries include the OpenSSL wrappers PyOpenSSL and M2Crypto, the Python version of Keyczar (see section 3.3.3.2), the NaCl-based libraries PySodium and PyNaCl (see section 3.3.3.1), Python bindings to the Botan library from the C++ crypto ecosystem and `oscrypto`, which leverages the crypto implementation of the respective operating system.

4.7 Conclusion

Programming ecosystems consist of a platform, libraries consisting of code and documentation as well as contributors and users. Their crypto ecosystem is the subset concerned with cryptography. Those ecosystems vary in size; C and C++ have the biggest crypto ecosystem because reference implementations and other algorithm implementations are mostly written in C. A few libraries span across multiple ecosystems. Most notably, OpenSSL and NaCl/libsodium have countless wrappers in other languages and Keyczar is implemented in at least seven languages.

[11] https://docs.python.org/3.6/library/crypto.html (2016-04-19)
[12] http://pypi-ranking.info/search/crypto/ (2017-03-23)
[13] http://code.activestate.com/recipes/576980/ (2017-03-23)

Some ecosystems are dominated by an official *standard* crypto solution built into the standard library. This approach seems to decelerate the entire crypto ecosystem. The standard solution reasonably attracts more users because it is easy to discover and requires no extra dependencies. Because the remaining libraries have fewer users, their development slows down and they take longer to mature. On the other hand, the standard solution is less agile than the remaining ecosystem, which results in problems: Java's JCA has API issues that cannot be resolved due to backward compatibility, and the .NET Framework's solution does not have authenticated encryption, yet.

.NET presents a solution at the same time: the CLR Security project extends the .NET Framework and offers authenticated encryption among other improvements. Its development is more dynamic because it is not part of the standard library, but the library is still officially developed by Microsoft and therefore credible, making this form of development a good compromise. The equivalent in Rust are so-called "rust-lang crates" (see section 5.6). Albeit official, such libraries compete with other libraries (more so than a standard solution) and, as they are being developed openly, they benefit from mutual ecosystem dynamics (forks, wrappers and direct contributions).

5 The Rust crypto ecosystem

The general ecosystem structure introduced in the previous chapter (see figure 4.1 on page 38) guides the further research regarding the crypto ecosystem of Rust. Section 5.1 derives research questions from every population and from every interaction, the following sections 5.2 to 5.5 contain the corresponding analyses, section 5.6 discusses the role of the standard library and section 5.7 proposes improvements.

5.1 Research questions

To get an overview of the ecosystem's scope, the first three research questions (RQs) pertain to its core population—the libraries:

RQ 1: *Which libraries does the ecosystem encompass?* (sections 5.2.1 and 5.3)
RQ 2: *How can the many libraries in the ecosystem be subdivided?* (section 5.2.2)
RQ 3: *What are the forking and dependency relations between the libraries?* (section 5.2.3)

The second most important population are the contributors, whose individual backgrounds are studied by the survey in section 5.4:

RQ 4: *Who are the contributors demographically, especially in comparison to average open source developers?* (section 5.4.4)
RQ 5: *What relevant qualifications do the contributors have?* (section 5.4.5)
RQ 6: *What motivates the contributors to work on crypto in Rust?* (section 5.4.6)
RQ 7: *How much time do the contributors invest in the crypto ecosystem?* (section 5.4.7)

The GitHub analysis in section 5.5 investigates the interactions among contributors and their contributions to the libraries with respect to these questions:

RQ 8: *Do the contributors work on multiple libraries?* (section 5.5.1)
RQ 9: *How is the work on a library distributed among its contributors?* (section 5.5.1)
RQ 10: *How do the contributors collaborate, cooperate and coordinate?* (section 5.5.3)

Because the API design processes strongly affect the libraries' usability and misuse resistance, the following questions target them specifically and the GitHub data is combined with survey responses and statements from the web to answer them:

RQ 11: *Do discussions about API design take place and, if so, where?* (section 5.5.4)
RQ 12: *How important is usability in relation to other goals?* (section 5.5.5)

Unlike with other programming languages, the Rust platform does not itself offer any crypto functionality, though there is ongoing discussion about including crypto in the standard library (see section 5.6). Nevertheless, the platform influences the crypto libraries, which leads to the following question:

RQ 13: *Which crypto-relevant features are offered by the Rust platform and which are missing?* (section 5.7.2)

Last but not least, the entire ecosystem's purpose is to serve its users. To be able to improve the ecosystem for them, it is vital to look at their usage of the ecosystem. These questions are addressed by a systematic usage analysis and a series of experiments:

RQ 14: *Which features are the most widely used?* (chapter 6)
RQ 15: *How usable and misuse resistant are the current APIs?* (chapter 7)

The raw data and processing scripts for all analyses presented in this thesis are available for download (see appendix A). All major analyses used to answer the research questions above have individual threats to their validity, which are discussed in the respective sections. However, there is a general threat to the coherence of those results: the analyses were spread out over the course of a semester and the Rust crypto ecosystem is continuously evolving. Minor problems uncovered in one analysis could be fixed before the next analysis already, and my presence in the ecosystem (asking questions on the #rust-crypto IRC channel, etc.) could have influenced the development and therefore the further analyses. To mitigate this issue, I report the date of every analysis, use absolute links to stable versions wherever possible and attempted to conduct every individual analysis within a few days or automate it for flexible reproduction so that each individual data set is consistent.

5.2 Library search and categorization

A systematic search approach was used to find all the available libraries in the Rust ecosystem (section 5.2.1). The libraries were then manually categorized (section 5.2.2), the forking and dependency relations were analysed semi-automatically (section 5.2.3), and the major libraries were selected for all further analyses (section 5.2.4).

5.2.1 Systematic search (RQ 1)

The search was started from a few different places (in October 2016 and refreshed on 19 February 2017):

- the *crypto* and *cryptography* keywords on crates.io
- the *crypto* and *cryptography* topics on GitHub

- Reddit thread "What crypto library do y'all use?"[1]
- the "Cryptography" collection on libs.rs[2]
- an overview on arewewebyet.org[3]
- asking on the #rust-crypto IRC channel on the Mozilla IRC network.

These sources alone turned up a vast number of almost a hundred crypto crates belonging to about 75 libraries or projects (a single project can deploy multiple crates). For the purposes of this thesis, all the crates that originate from a single GitHub repository or are otherwise developed under a single umbrella (like a common "brand" name) are treated as a single library. Only libraries whose main purpose is to provide cryptographic functionality at any level of abstraction were considered (cf. section 4.2.2). Libraries whose crate was retracted ("yanked") from crates.io were excluded. A few additional libraries were identified by considering the main contributors of the main crypto libraries and looking at other GitHub repositories they work on, **leading to a total of 80 libraries**. This answers RQ 1, though the full list is not reproduced in this thesis but provided as supplementary material (see appendix A item 3).

5.2.2 Categorization (RQ 2)

There are several possible ways to categorize the libraries. As this thesis focuses on usability, a natural categorization groups the libraries by functionality. The presented categorization first organizes the libraries **by the level of the provided algorithms**, such that higher-level libraries normally depend on lower-level libraries:

- 10 crypto-specific utility libraries for constant-time operations, secret memory and so on. Examples: `nadeko`, `secrets`[4]
- Libraries on the primitive level.
  - 12 *larger* libraries which offer *multiple primitives* and usually multiple algorithms per primitive. The implementations are either written in Rust or attached through wrappers to code in another language. All libraries in this category are introduced in section 5.3.
  - 35 libraries which implement a single primitive, algorithm or a small family. Example: `ed25519-dalek`[5]
  - 5 libraries which offer a simpler interface to other implementations for specific application scenarios. Example: `crypto_vault`[5]
- 18 libraries that implement *cryptosystems or protocols*. Example: `rustls`[6]

[1] https://reddit.com/4d8hxm (2016-04-06)
[2] http://libs.rs/cryptography/ (2016-04-12)
[3] http://www.arewewebyet.org/topics/crypto/ (2016-10-05)
[4] https://github.com/klutzy/nadeko, https://github.com/stouset/secrets (2017-04-07)
[5] https://code.ciph.re/isis/ed25519-dalek, https://github.com/zmbush/crypto_vault (2017-03-23)
[6] https://github.com/ctz/rustls (2017-03-16)

Most parts of this categorization can be further refined. For instance, there are 10 TLS libraries, which form a significant sub-population of the 18 protocol-level libraries. Note again that only libraries which expose a (mostly) crypto-related API to other developers are included in the ecosystem. This excludes command-line tools and applications that simply use cryptography somewhere inside, and it also excludes libraries whose API is not crypto-related anymore, though it might still be security critical.

A relevant **alternative** way to subdivide the libraries (RQ 2) is to group them **by dependencies**, i.e., to consider the visible clusters in figure 5.1. Many alternative categorizations are possible, of course, depending on the purpose.

5.2.3 Forks and dependencies (RQ 3)

To analyse the dependency relations between the libraries, I used a script that downloaded every library's `Cargo.toml` file from the repository (on 9 March 2017) and parsed the respective dependency sections. Forking relations were noted manually when inspecting the repository (GitHub displays the upstream project, if any, under the name of the current repository).

The dependency graph in figure 5.1 shows 54 of the 80 Rust crypto libraries and their dependencies—the remaining ones do not have any dependency relations with other *crypto* libraries. Note that all these libraries have many more dependencies to libraries *outside* the crypto ecosystem that are not depicted. Two kinds of "centres" can be seen in the graph: `openssl`, `rust-crypto` and `ring` are surrounded by other libraries depending on them. On the other hand, `octavo` and `sarkara` consist of multiple smaller crates aggregated in a meta crate, which thus depends on all the others. The visualization shows multiple clusters that can be interpreted as sub-ecosystems since there are more interactions within the clusters than with the outside.

Almost all projects have a couple of forks because the standard workflow on GitHub requires forking a project before making changes and then submitting them as a pull request. Besides these forks for technical reasons, very few projects have been forked with the intention of developing the fork separately—such an intention becomes apparent when a separate crate is published. All cases of such forks currently concern the major libraries for primitives introduced in the next section and are thus depicted in figure 5.2 on page 50. Note that this analysis cannot find forks that happen by copying (part of) a library's code into a new project manually without creating a fork on GitHub.

To answer RQ 3, regarding *dependency* relations between libraries, it can be concluded that **the expected dependencies** from protocol-level libraries (like `rustls`) on primitive-level libraries (like `ring`) and on utility libraries (like `untrusted`) **do indeed exist**, but **surprisingly few libraries participate in this network at all**—one third does not and

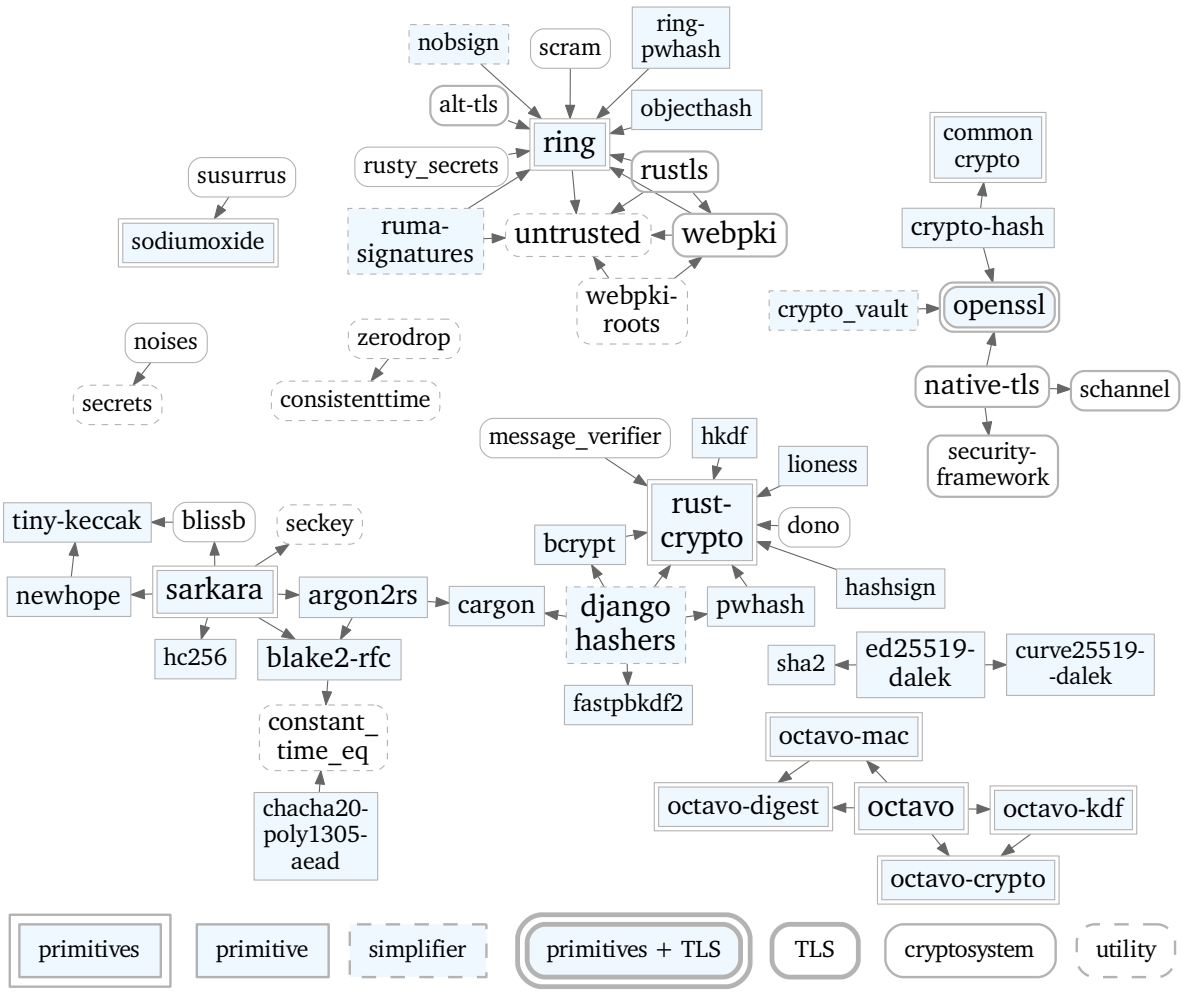

Figure 5.1: Rust crypto libraries and their dependencies (only crates with at least one dependency are shown)

only depends on other non-crypto libraries. Regarding *forking* relations, it can be concluded that **forking mostly happens due to GitHub's workflow and only few *real* forks exist**.

5.2.4 Major libraries

Considering the large number of libraries, it is necessary to focus on a subset for analyses which cannot be carried out automatically. Two metrics lend themselves to quantify the importance of a library: the number of downloads (from `crates.io`) and the number of dependent crates (the number of higher libraries and applications published on `crates.io` which use the crate in question). The main difference between the two is that transitive dependencies only count towards the downloads. For instance, `rustls`

depends on `ring`. A single user of `rustls` only increases the dependent crates counter of `rustls` but leads to equally many downloads of both `rustls` and `ring`. A bug in `ring` can affect this user as much as any other, direct user of `ring`—thus the number of downloads measures the impact of a library. However, this user probably does not use `ring`'s API directly and hence the usability and misuse resistance of `ring` itself do not affect the transitive user's code. For this thesis, the number of dependent crates is therefore the more suitable metric.

The libraries with more than 20 dependent crates (as of 1 March 2017) are: **`rust-crypto`, `rust-openssl`, `sodiumoxide` and `ring`**, in decreasing order. All of them fall in the category that provide *multiple primitives* and are introduced in detail in the following section. In the remainder of this thesis, these libraries are referred to as the *major libraries*. **For some analyses, the `octavo` library is also included** because it falls in the same category and is interesting to compare with because it implements all primitives purely in Rust. In addition, **`RustCrypto` and `rust_sodium` are sometimes included** alongside or instead of `rust-crypto` and `sodiumoxide`, respectively. The former two are forks of the latter two and have advantages in certain areas, whereas they are identical in others.

5.3 Libraries providing primitives

This section introduces libraries which provide multiple cryptographic primitives in Rust. These libraries are the centre of all further analyses and improvements. See section 2.2.9 for the characterisation of *primitives* used in this thesis. Table 5.1 provides an overview of the key data and figure 5.2 visualizes the libraries and their ancestors on a timeline.

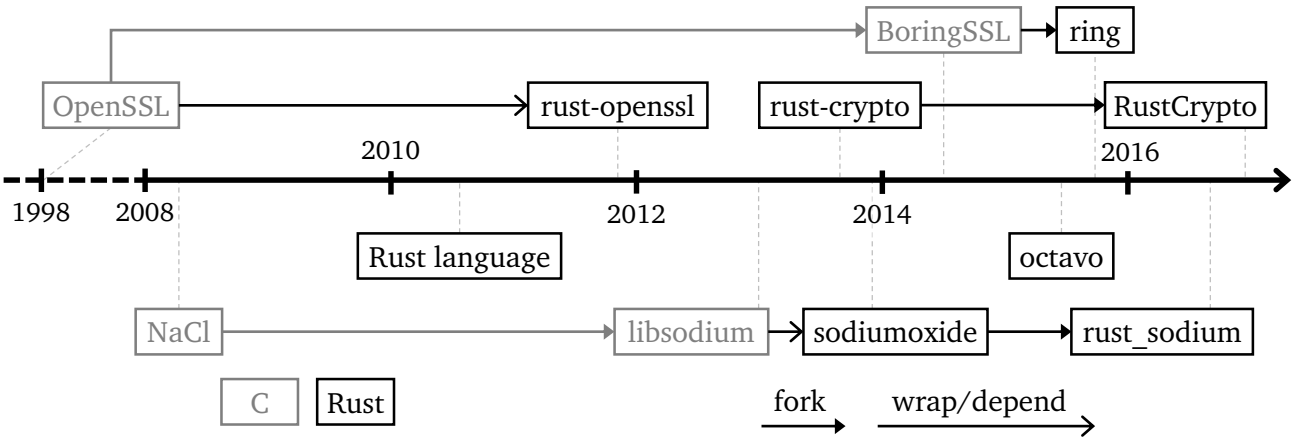

Figure 5.2: Timeline of the major Rust primitive libraries' start dates and ancestors

Name	rust-openssl	rust-crypto	sodiumoxide	octavo	ring	rust_sodium	RustCrypto
Based on	OpenSSL	(plain Rust)	libsodium	(plain Rust)	BoringSSL	sodiumoxide	rust-crypto
Started	Dec 2011	Sep 2013	Dec 2013	Jul 2015	Sep 2015	Aug 2016	Oct 2016
Last release	ongoing	May 2016	ongoing	Apr 2016	ongoing	Sep 2016	ongoing
Last commit	ongoing	Sep 2016	ongoing	Sep 2016	ongoing	Nov 2016	ongoing
Open issues/PRs	14 / 2	48 / 26	8 / 4	7 / 1	177 / 11	1 / 1	7 / 1
Commits	1,690	753	435	288	6,027 (2,299)[a]	46	72
Contributors	123	49	27	10	83 (33)[a]	8	5
Downloads	702,827	248,756	56,352	753	36,689	8,940	<20,000
Downloads (60d)	95,132	49,450	5,159	130	16,599	2,861	(unclear)
GitHub stars	251	538	201	117	448	10	21
Dependent crates	92	118	30	0	25	8	9+
API level	medium	low	high	medium-low	medium-high	high	low
FIPS coverage[b]	67%	50%	8%	31%	34%	8%	19%
CNSA coverage[c]	92%	67%	8%	17%	67%	8%	17%
Other algorithms[d]	9+[e]	20	11	7	6	11	16

Table 5.1: Key data of major crypto libraries (as of 2 March 2017)

[a] `ring` is technically a fork of BoringSSL and therefore contains all of its contributions. The numbers in parentheses only count contributions which are not present in the upstream BoringSSL project.

[b] Algorithms defined in the FIPS 140-2 Annex A [NIS16]: AES (CBC, CCM, GCM, XTS), GMAC, CMAC, HMAC, Triple-DES, SHA-1, several SHA-2, SHAKE and SHA-3 variants, DSA, RSA and ECDSA

[c] Algorithms defined in the Commercial National Security Algorithm (CNSA) Suite [Inf15] (former "Suite B"): AES (CBC, GCM), SHA2-256, SHA2-384, ECDH, ECDSA

[d] For example: ChaCha20, Poly1305, bcrypt, scrypt, Curve25519

[e] OpenSSL provides a plethora of algorithms, but `rust-openssl` currently only exposes a selection.

5.3.1 rust-openssl

The library with bindings to OpenSSL is the oldest major crypto library in Rust and is actively developed and maintained by a member of the official Rust library team (@sfackler). It is also the most used library according to the downloads statistics, though it is not directly comparable because `rust-openssl` offers both crypto primitives *and* a TLS implementation in the same crate. Following some criticism regarding the code quality in February 2016, which the developer confirmed himself ("some of rust-openssl is a mess, particularly on the crypto side") [Cua+16, posts 12 and 17], there has been a lot of activity over the rest of the year with about as many contributions as in all previous years combined. As a result, the code is in much better shape today.

Because it is based on the mature OpenSSL library, the obvious advantage of `rust-openssl` is its high standard of security and stability, although the wrapping Rust code has not been audited itself. The underlying OpenSSL library is also the major obstacle when using `rust-openssl` today: the OpenSSL binaries need to be downloaded, compiled and set up manually in the build environment. While this is a fair requirement, it may be a significant barrier for beginners in systems programming.

According to the developer, the API is designed to be higher-level than OpenSSL's API. For example, the Rust API offers a single function to hash a value, whereas OpenSSL requires at least three calls (init, update and final). The wrapping layer also takes care of error handling using Rust's `Result<T,E>` type, which fixes a major source of API misuses that OpenSSL has been criticized for [GIJ+12, sec. 4.1]. Apart from these improvements on a fine-grained level, `rust-openssl` explicitly does not attempt to create high-level APIs to improve usability and misuse resistance, as it intends to mirror OpenSSL's functionality. It also only exposes part of OpenSSL because "OpenSSL has approximately an infinite amount of features, so stuff gets supported as-needed in rust-openssl,"[7] as the developer puts it.

5.3.2 rust-crypto

The oldest library that implements primitives in Rust itself is `rust-crypto`. After its main development period from early 2014 to mid-2015, the number of commits has decreased and issues and PRs started to accumulate, though the developer (@DaGenix[8]) declined rumors that it might be abandoned.[9] There is a recent effort to split `rust-crypto`'s

[7] Source: #rust-crypto IRC chat (2017-02-21)

[8] The @-notation is used throughout this thesis to refer to GitHub usernames. The profile of @username can be found at https://github.com/username.

[9] https://reddit.com/46s75m, https://github.com/Keats/rust-jwt/issues/7 (2017-03-23)

functionality into smaller crates under the `RustCrypto`[10] GitHub organization. Because of this modularization, the statistics of `RustCrypto` cannot be determined exactly (see table 5.1). Despite the developer's warning that `rust-crypto` "has not been thoroughly audited for correctness," it is currently the second most downloaded crypto library and the one with the most dependent crates, among which there are OAuth implementations, a Bitcoin library and a Simple Mail Transfer Protocol (SMTP) implementation.

The focus of `rust-crypto` is on the implementation of primitives, so it does not wrap another implementation. While the majority of its code is written in Rust, it requires some C code for the AES New Instructions (NI),[11] which may cause build problems especially on Windows. As can be seen from table 5.1, the library already supports many algorithms and covers most users' needs. In relation to the other primitive libraries, `rust-crypto` is a rather low-level set of primitive implementations and its API reflects that and serves to make the implementations interoperable and composable.

5.3.3 sodiumoxide and rust_sodium

NaCl by Daniel Bernstein [Ber09] was designed with usability as one of its three main goals. The fork called libsodium [Den] was created to make it more easily portable and packageable, and unlike NaCl itself it is actively maintained and continually improved today. Please see section 3.3.3.1 for more details.

The bindings for Rust are in the `libsodium-sys` crate and can be used through proper Rust APIs in the `sodiumoxide` crate. Unfortunately, `sodiumoxide` is difficult to build (especially on Windows) and there are no build instructions. That is one of the issues that its fork `rust_sodium` solves (at least for the `-gnu` toolchain), which was created very recently (August 2016) and is currently almost identical to `sodiumoxide` apart from the build and continuous integration system. For this reason, the experiments in this thesis evaluate `rust_sodium` only, though it is worth pointing out that the API, documentation and most other parts were created by `sodiumoxide`'s developer (@dnaq).

`sodiumoxide` has the same concepts and characteristics as NaCl/libsodium and provides roughly the same set of high-level functions and underlying primitives. It also deliberately excludes any insecure or deprecated algorithms and only offers few, opinionated choices for every primitive. Thus, it cannot be used in certain scenarios where specific algorithms are required by a cryptographic protocol. `sodiumoxide` is more than just a simple wrapper: it uses Rust's type system and other language features to make the

[10] https://github.com/RustCrypto (2017-03-16)

[11] The AES-NI is an instruction set extension for the x86 microprocessor architecture and allows applications to use hardware acceleration for AES encryption and decryption operations.

API more usable and it adds Rust code samples to libsodium's extensive documentation. `sodiumoxide` and `rust_sodium` combined have a lot fewer downloads and dependent crates than `rust-crypto` or `ring` despite similar age.

5.3.4 octavo

`octavo` is another take at pure-Rust cryptography. It is split into multiple crates so that, for example, hash functions can be used without including the entire crypto stack. The main developer (@hauleth) points out in various places that the code is not production-ready and probably insecure. In particular, it does not defend against timing attacks at all, though a new constant-time helper library[12] is being developed. In addition to not being reviewed, the library is also incomplete. Of the more popular algorithms, some significant ones are missing, e.g., AES, Poly1305 and the Diffie-Hellman key exchange.

Although `octavo` is not safe to use in the short term, its long-term goal is to provide a solid, "hard tested"[13] implementation that is more readable than `rust-crypto`'s code.[14] The project clearly aims at making this implementation available in other programming ecosystems,[15] supporting the idea that Rust is a good language to implement crypto primitives in because of its safety guarantees.

5.3.5 ring

BoringSSL is Google's fork of OpenSSL, which cleaned out some old algorithms among other changes. `ring` provides a subset of Bo*ring*SSL's features (hence the name) in Rust and, in particular, it excludes the entire TLS stack and most deprecated algorithms (for example, there is no MD5 in `ring` on purpose). A notable difference to other Rust crypto libraries based on implementations in other languages is the way in which `ring` incorporates BoringSSL: it is technically a fork. All other wrapping libraries discussed in this chapter, on the other hand, are Rust crates with a dependency on a `sys` crate, which in turn is built against the original C library. So the original library is a dependency, whereas in `ring` it is the upstream. BoringSSL updates are regularly merged into `ring` and there are also contributions the other way round.

Despite this close relationship, the two libraries share mostly the algorithmic core and `ring` comes with its own, independent Rust API and documentation. It is the developer's

[12] https://github.com/libOctavo/ct (2017-03-23)
[13] https://reddit.com/comments/3hci07/-/cu6apht/ (2017-03-23)
[14] https://reddit.com/comments/3hci07/-/cu67q0c/ (2017-03-23)
[15] https://github.com/libOctavo/octavo/blob/d94d92/src/lib.rs#L9 (2017-03-23)

declared goal to make `ring` as usable[16] and "foolproof"[17] as possible and it is the only library that has a (frequently used) "usability" label in its issue tracker.[18] The documentation is extensive, virtually no method or module remains undocumented, the only deprecated algorithm (SHA-1) is marked as such, and there are code samples for most major functions. Because `ring` is mostly non-Rust code, it requires a C/C++ compiler and on Windows it currently depends on Microsoft's MSBuild tool and therefore only works with the `-msvc` toolchain, not with the `-gnu` toolchain.

Compared to the other libraries, `ring` is relatively new and had its first release on `crates.io` in August 2016. At the time of writing, it is by far the most actively developed library for crypto primitives in Rust, with the highest number of commits and quite many contributors considering its young age. Moreover, it is most frequently recommended library (see section 5.4.8).

5.3.6 Other primitive libraries

`sarkara` is an experimental post-quantum cryptography library. `sodalite` and `microsalt` wrap the TweetNaCl library, which implements the main NaCl primitives with as little code as possible to be easily auditable. `grypt` and `commoncrypto` are relatively straightforward bindings to libgcrypt and the Common Crypto library on Mac OS X, respectively. These five libraries also provide a range of primitives and algorithms, but they are neither introduced nor analysed in detail in this thesis because they are less developed and less popular. In addition, there are numerous libraries which only implement a particular primitive and mostly only a single algorithm.

5.4 Contributors survey

To find out about the backgrounds and the opinions of the developers in the Rust crypto ecosystem, I conducted an online survey. Sections 5.4.1 to 5.4.3 describe the setup of the survey. Sections 5.4.4 to 5.4.7 report on those results that pertain to the contributors themselves to address RQs 4 to 7. Section 5.4.8 briefly reports on the crypto libraries recommended by the survey respondents for basic tasks. Further results from the survey can be found in section 5.5 as they concern the collaboration of contributors (RQs 8 to 12) and in section 5.7.2 regarding crypto-relevant features missing from the Rust platform (RQ 13). Finally, section 5.4.9 discusses threats to the validity of this survey.

[16] https://reddit.com/comments/3jdlux/-/cuool0h/ (2017-03-23)
[17] https://github.com/briansmith/ring/issues/359#issuecomment-263207373 (2017-03-23)
[18] https://github.com/briansmith/ring/labels/usability (2017-03-23)

The survey structure, anonymous parts of the responses, the R scripts used to analyse the data and more plots and tables than the ones included in this chapter are available for download (see appendix A).

5.4.1 Survey design

The 37 questions were chosen in collaboration with my supervisor based on their relevance regarding the research questions 4 through 13 as well as their potential to yield insights for improving the crypto ecosystem of Rust. The questions were grouped by topic and displayed on separate pages:

1. demographics (7 questions)
2. involvement of the contributor (10 questions)
3. Rust's crypto ecosystem (4 questions)
4. API design (4 questions)
5. questions about the concrete libraries the contributor works on (8 questions)
6. final comments (3 questions)

The full list of questions is reproduced in appendix B. While the first two parts consisted mostly of multiple choice questions and numeric answer fields, the latter parts were dominated by open-ended questions. Group 5 in particular asked the contributors about the concrete projects they work on. As this makes them personally identifiable, these questions were grouped separately. Survey participants were informed about a possible loss of anonymity and could choose between skipping the entire part, answering it anonymously (that is, their individual answers are not published anywhere) or providing a contact address (in which case they have been asked for permission before any individual statements were published).

Overall, the survey was estimated to take between 10 and 20 minutes, depending on the detail of the answers to the open-ended questions. The survey was implemented with a self-hosted, HTTPS-protected installation of the open-source, web-based tool LimeSurvey [LS16] version 2.58.0 build 170104.

5.4.2 Participant recruitment

For a small initial test of the survey design, on 9 January 2017, I posted the link and a short introduction to the #rust-crypto IRC channel on the Mozilla IRC network, where about a hundred users are permanently logged in and about ten actively chat on a weekly basis. This resulted in three responses.

To recruit further participants without the bias introduced by their IRC use, I systematically gathered the e-mail addresses of the contributors to the major crypto libraries

(specifically the libraries for primitives and the TLS libraries, section 5.2) who had contributed at least five commits. Invitation e-mails were sent on 21 and 22 January 2017. If a contributor's e-mail address was not specified on their GitHub profile, on the personal website linked from there or in the meta data of one of their commits, they were not sent an invitation e-mail. Overall, 58 contributors were invited via e-mail and 20 additional responses were collected before the survey closed on 5 February 2017, resulting in a total of 23 respondents.

5.4.3 Sample

Of the 23 responses, 3 were incomplete and had to be excluded. All remaining 20 responses came from people who self-identified as *contributors* in the survey. Because many contributors only make minor or non-technical contributions like small bugfixes, fixes to the build process or improvements to the documentation, the survey asked whether they implement "cryptographic algorithms, interfaces, libraries, etc. in Rust." 5 respondents answered "No" and are therefore not treated as *implementers*, but only as *contributors* in the following evaluation. While some results are presented based on the answers of all 20 *contributors* (also referred to as *respondents*), other results that are implementation-specific are evaluated only for the 15 *implementers*.

5.4.4 Demographics (RQ 4)

All respondents are male, almost half are 25-29 years old and another third are between 30 and 39 years old. About a third lives in the USA, a quarter in Germany, two live in the UK and two in the Netherlands; the remaining countries only get a single mention. Almost half have a graduate degree, a third have a bachelor's degree, and three are still younger than 22 and have high school degree. 60% work full time, 30% have a part-time job and 15% of those study at the same time. None of the respondents is a student without a part-time job.

These results roughly correspond to the results of the 2016 Stack Overflow developer survey [Sta16] and previous research (see section 3.1). In conclusion regarding RQ 4, the contributors in the Rust crypto ecosystem are **demographically average developers**.

5.4.5 Qualification (RQ 5)

The participants were asked to rate their own cryptography and Rust skills as well as to specify the years of experience they have. Regarding their cryptography skills, most implementers rate themselves as "experienced" and 20% only as "educated" and hence without practical experience, whereas the Rust programming skills are consistently rated

"experienced" or better, with the majority answering "advanced" (see figure 5.3). Only 4 implementers (26.7%) rate their crypto skills above their Rust skills.

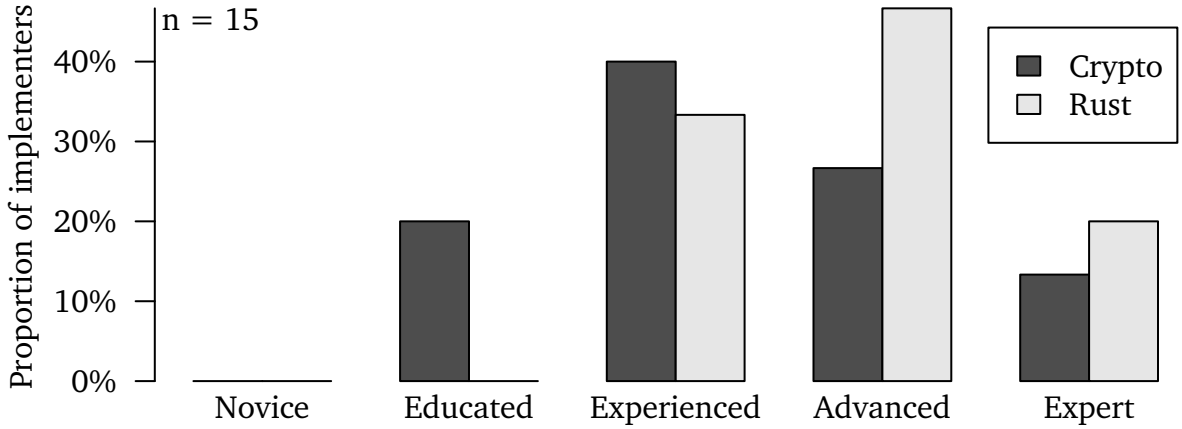

Figure 5.3: Self-ratings for cryptography and Rust skills

The years of experience have considerably different distributions in the different areas (see figure 5.4). All contributors have extensive programming experience between 4 and 25 years (median 14). This is significantly above the values reported by Stack Overflow [Sta16], where the median is 6.5 and only 26.5% have 11 years of experience or more, compared to 60% in the present survey. On the one hand, most programmers seek help on Stack Overflow and only few are experienced enough to provide it, which introduces a bias towards inexperienced programmers. On the other hand, the difference is large enough to support the hypothesis that Rust crypto contributors have above-average programming experience.

The experience with Rust itself naturally cannot be as long because Rust was published as recently as 2010, and indeed the maximum of 4 years is only as high as the minimum of the programming experience distribution. The median of 2 years indicates that most implementers have only recently started using Rust. For instance, the creator of the well-recognized `rustls` library (@ctz) only has one year of experience with Rust (and therefore crypto in Rust) but a strong programming and cryptography background (25 and 13 years, respectively). Little experience with Rust is the norm rather than the exception, and it is compensated by feedback and contributions from the community. @ctz, for instance, announced his `rustls` library with the remark: "It's my first large-scale rust project, so I'd appreciate some comments."[19]

[19]https://reddit.com/4s2n9q (2017-03-23)

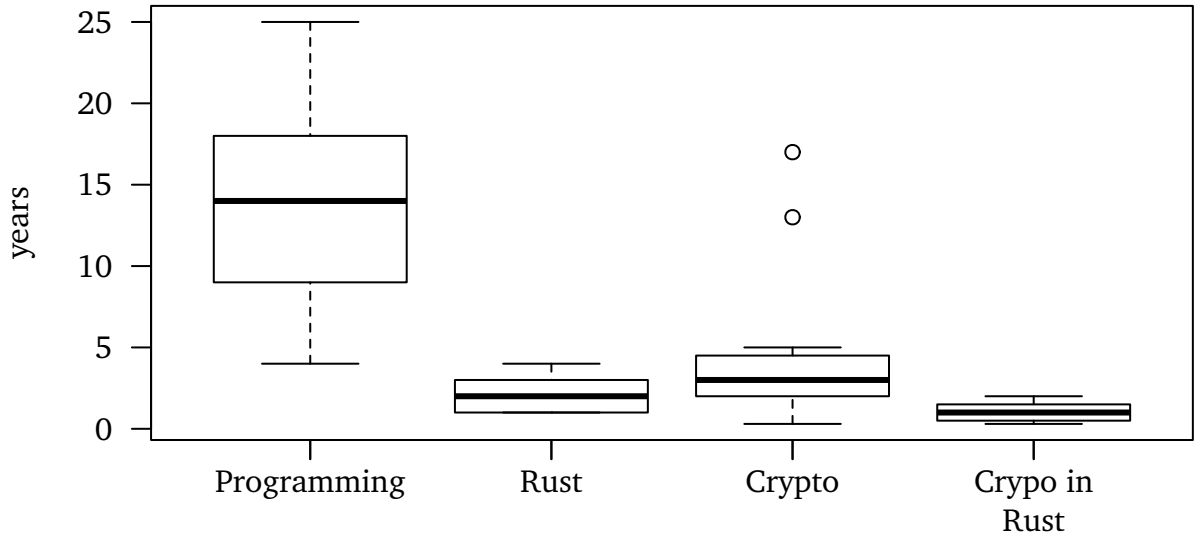

Figure 5.4: The implementers' experience with programming and cryptography in Rust and elsewhere

Unlike the programming experience, the cryptography experience is not normally distributed but visibly right-skewed, hence most contributors are relatively inexperienced (the median is 3 years), but there are a few experts with up to 17 years of experience.

To summarize the qualifications (RQ 5), Rust crypto contributors have **above-average programming experience, little Rust experience due to the youth of the language and there are only a few experienced cryptographers among them**.

5.4.6 Motivation (RQ 6)

When asked for their motivation to work on crypto in Rust, many contributors included the goals for their specific libraries in their answer, namely portability, idiomatic APIs, interoperability of different primitive implementations and high-quality TLS support. One respondent simply names the lack of other suitable libraries as their motivation. This indicates that the main motivation is simply an intrinsic desire or a practical need for the result.

In general, money would be the most straightforward extrinsic motivation. Only one respondent—a minor contributor to the `ring` library—is currently paid for implementing crypto code in Rust and another two were paid in the past. In contrast, three respondents (in this survey among *Rust* crypto contributors) are currently paid for crypto implementation work in *other* programming languages, and another two were paid for other crypto-related tasks in the past. The paid work on crypto code does not necessarily result in publicly available contributions to the crypto ecosystem, though. This shows that

money is not a motivational factor in the Rust crypto ecosystem, and most contributors work on the public Rust crypto projects in their unpaid free time.

As other extrinsic motivational factors, respondents mostly named properties of the Rust language, namely *safety* (mentioned 5 times), *expressiveness* and its *ability to run in bare-metal environments*. Intrinsic motivational factors are the desire to learn and gain insights and to help grow the Rust ecosystem (mentioned twice).

In conclusion regarding RQ 6, Rust crypto contributors have **several motives that also appear in the literature**. Hertel, Niedner and Herrmann [HNH03], for instance, also find **pragmatic motives** like an interest in the produced software itself and **hedonistic motives** like the desire to learn. Beyond that, Rust crypto contributors are specifically motivated by the **desirable properties of the Rust programming language**.

5.4.7 Time commitment (RQ 7)

Figure 5.5 shows the amount of time that the respondents spend on crypto-related tasks today and have spent in the past when they were most active. While the hours per week at each contributor's individual peak time form an only slightly right-skewed distribution, the time commitments today (at a given point in time which is the same for all contributors) are heavily right-skewed with medians at zero. This latter distribution is consistent with the amount of contributions that the developers make (see section 5.5.1).

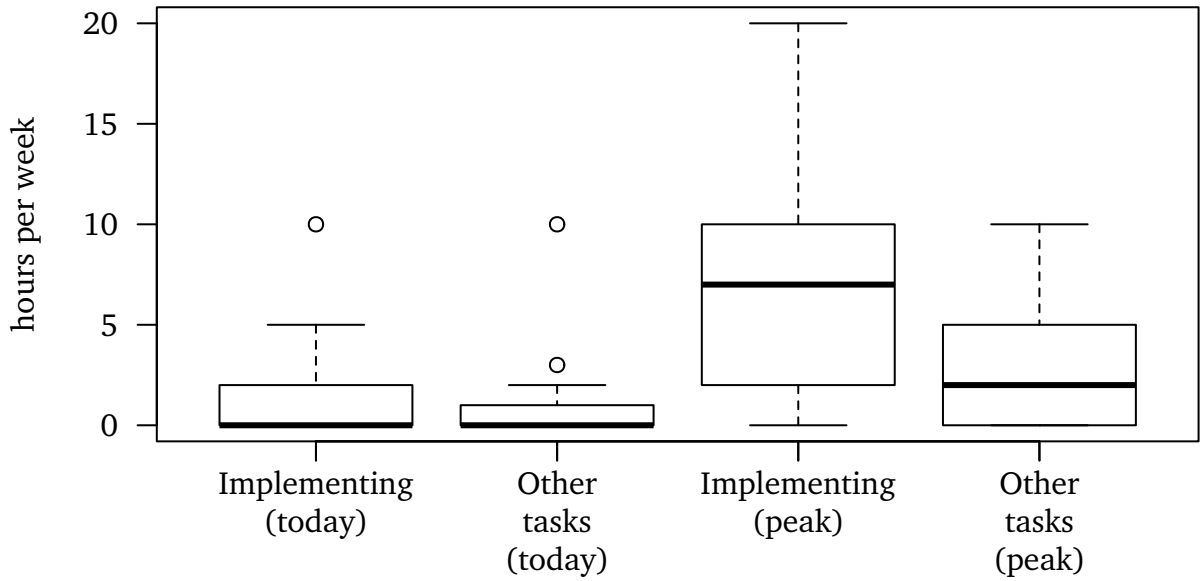

Figure 5.5: Amount of time in hours per week spent on implementation and other tasks related to cryptography in Rust. The two distributions on the left show the time spent today and the two on the right refer to the (possibly past) time when the respective contributor was most actively involved.

Of the 15 implementers, nearly half (7) spent 10 hours or more per week at some point. Today, the only developer who works 10 hours on implementation and an additional 10 hours on other tasks is also the only one who is paid to do so (see the previous section), whereas the others spend much less time (the overall medians are both zero, the means are 1.6 and 1.3 hours, respectively).

It is crucial to note that the time spent on cryptography tasks does *not* necessarily mean an investment in the public (Rust) crypto ecosystem. The survey question (unfortunately) does not distinguish between the two, so that RQ 7 cannot be answered definitely. However, the one contributor with the highest time investment answered a respective follow-up question and indicated that part of his 20 hours per week result in contributions to public crypto libraries.

Hertel, Niedner and Herrmann [HNH03] find that a "lack of time is one of the biggest obstacles for participating in the Linux kernel project" and the situation seems to be similar in the Rust crypto ecosystem. For example, the main developer of `rust-crypto` writes: "I've been quite busy at work over the last few months and its hard to program 8 - 10 hours during the day and then jump into more programming at night."[20] And a survey respondent writes: "No funding appears to be available for such work, making it hard to sustain." Even @briansmith, the main developer of `ring` and currently the most productive contributor in terms of commits per week (he is likely not among the survey respondents), sees the same issue: "People working on Rust crypto are making money doing stuff that has nothing to do with Rust, and then investing it into Rust. That's not a sustainable strategy." [Cua+16, post 14]

Regarding RQ 7, it can be concluded that the **time commitment of all but the main contributors is well below 5 hours per week**. Most contributors work on Rust crypto in their **unpaid free time**, making the **lack of time a major obstacle** to the success of the crypto ecosystem.

5.4.8 Recommended library

The survey asked the respondents to recommend a library for string encryption to a Rust beginner. For such a basic task, the survey respondents could ignore the feature richness of the libraries and focus on other quality attributes like stability, usability, documentation, etc.

`ring` received 9 recommendations, which is more than all others combined. The given reasons included the solid BoringSSL base, the credible main developer, high-level and misuse resistant APIs and the fact that it is actively maintained. `sodiumoxide` and

[20] https://github.com/DaGenix/rust-crypto/issues/106#issuecomment-74023642 (2017-03-23)

`rust-openssl` received 3 recommendations each and `rust-crypto` received 2. Although these recommendations partly reflect the involvement of the respective survey respondents, which some pointed out as a threat to validity, the overall picture is consistent with recommendations given out on other occasions on the #rust-crypto IRC channel and elsewhere. Particularly Tony Arcieri [Arc17], who is an active crypto expert in the Rust crypto ecosystem but not directly involved in any of the libraries in question, repeatedly recommends `ring`.[21]

5.4.9 Threats to validity

The sample is the main threat to the validity of the presented survey results. Firstly, it only encompasses 20 respondents because the overall number of contributors in the Rust crypto ecosystem is rather low—the response rate of about a third is above average and resembles response rates of company-internal surveys.[22] Thus, no statistically significant correlations could be drawn from the collected data. Secondly, the participants were not sampled randomly from a larger population. To increase the absolute number of respondents, all relevant Rust crypto contributors had to be invited, so the sample is biased towards users who are generally more willing to participate in surveys (about Rust) or who are more passionate about Rust. Other than that, the recruitment mode (IRC and e-mail) and the web-based format of the survey likely did not introduce any additional bias. The proportion of repository owners who participated in the survey (i.e. the sub-sample from the sub-population of all main contributors) is also a third, indicating that the sample has a reasonable coverage in this respect.

Another threat to the honesty of the responses is the non-anonymous part in the survey. The survey was designed to avoid any possible bias by placing and mentioning the respective part only at the very end so that no participants would abort the survey early or decide not to participate because of it. In the non-anonymous part itself, 8 out of 15 implementers (or 12 out of 20 contributors) decided to remain anonymous and skip it or not to provide a contact address. The additional bias and the even lower number of responses in this last part have only little impact on their validity because the last part consisted mostly of free-text questions for opinions and ideas, which were aggregated without statistical analyses and will be presented as part of the discussions in the following section.

[21] https://reddit.com/4xyb5j (2017-03-23)
[22] https://www.surveygizmo.com/survey-blog/survey-response-rates/ (2017-03-01)

5.5 GitHub analysis

Because the Rust ecosystem is entirely open source, (almost) all contributions and other interactions happen publicly. This chapter uses the publicly available data on GitHub, responses from the survey introduced in the previous section and personal observations to shed light on how the Rust crypto community works together.

Section 5.5.1 addresses RQs 8 and 9 by quantitatively analysing contributions data from GitHub. The data shows an overall lack of contributors, which is discussed in section 5.5.2. Section 5.5.3 describes the various ways in which contributors interact (RQ 10) based on qualitative observations and section 5.5.4 investigates the frequency of discussions about API design and usability on GitHub (RQ 11). To put the presented numbers into perspective, section 5.5.5 draws from survey responses and public statements of repository owners and identifies which projects and developers make usability a priority (RQ 12). Finally, section 5.5.6 discusses threats to the validity of the presented analyses of GitHub data. The technical details of the automated GitHub analyses are not elaborated, but the R scripts and raw data are available for reproduction (see appendix A items 6 and 7).

5.5.1 Quantitative contributions analysis (RQs 8 and 9)

This section quantifies the relationship between contributors (GitHub users) and libraries (GitHub repositories). Some users contribute more to a library than others, and some contribute to multiple libraries related to cryptography in Rust. The "Contributors" graph on every repository's "Graphs" page shows every contributors' involvement over time as well as the total number of commits, additions and deletions. For an automated analysis, I downloaded this data through the GitHub API on 1 March 2017.

The 80 libraries in the Rust crypto ecosystem (see section 5.2) have 392 contributor relations in total, though there are only 279 unique contributors. While most of them contribute to only one project, 38 contribute to two and 19 contribute to three or even more projects (up to 11). Looking at the top ten contributors (with respect to the *number* of repositories the contribute to), there seem to be two main reasons for the high numbers: Some contributors like @quininer (11 repositories), @ctz (5) or @cesarb (5) simply split their work into many smaller crates, each of which has a separate repository, and thus contribute mostly to their own repositories. On the other hand, users like @frewsxcv (9) or @tarcieri (5) are active across a relatively wide range of projects which they do not own (@frewsxcv owns none of the 9 Rust crypto projects he contributes to). Regarding RQ 8 it can thus be concluded that **a few developers work on multiple Rust crypto libraries but most have only contributed to one** (and possibly to many other non-crypto Rust libraries).

The reverse relations are distributed similarly: almost half of the projects only have a single contributor, many projects have a few (less than ten) and only the bigger projects attract more contributors—up to 100 in the case of `rust-openssl`. The other libraries with many contributors are `ring` (where all BoringSSL contributors are counted in), `rust-crypto`, `sodiumoxide`, `rustls` and `octavo`, in decreasing order.

But even among the libraries with many contributors the contributions follow an extremely skewed distribution—so much so that they effectively also have a single main developer. Following previous work [CGJ16, for example], the contributions of a user are measured as the number of commits. Chełkowski, Gloor and Jemielniak [CGJ16] analyse 263 Apache projects and find that they "predominantly [rely] on radically solitary input." They contrast this observation with the general perception that open source development is collaborative.

In the Rust contributors survey, when asked about the number of main developers of the library they work on (without looking it up), the respondents answered with fairly different numbers for the same libraries, depending on their notion of a *main developer*: most answered that `ring` and `rust-openssl` have a single main developer, but one respondent estimated 8 for `ring` and another said 20 for `rust-openssl`. In fact, all larger crypto libraries in Rust (including those two) have a single contributor whose contributions far outweigh all others combined.[23] While a main contributors' contributions encompass hundreds of commits and 10k–100k lines of codes, the remaining contributors have on the order of ten commits and 100-1000 lines of codes.

In summary, **most developers only contribute to a single library but a few are more active** (RQ 8). A few projects thus receive much more attention than the majority, which have very few contributors overall (less than ten). The distributions of contributions towards individual libraries (RQ 9) are heavily skewed: a **single main developer contributes more than all others combined and is often the only contributor in the first place**.

5.5.2 Lack of contributors

The data presented in the previous section shows that most crypto projects are driven by a single main developer. A couple of survey respondents named the low number of contributors as an obstacle that the Rust crypto ecosystem currently faces. They make out two causes. Firstly, some survey respondents bemoan a **lack of funding**, which

[23] If you manually confirm this hypothesis with the "Contributors" pages on GitHub, note that `ring`'s page includes all BoringSSL contributors and that the main developer @DaGenix is missing from `rust-crypto`'s page because the commits were made by a different Git user (Palmer Cox).

forces developers to work on their projects in their (limited) free time (cf. section 5.4.7). Secondly and more importantly, the work on cryptography in Rust requires **two rare skill sets**, the combination of which is even rarer. One respondent says that there is especially a lack of cryptographers who become active in the Rust ecosystem, meaning that it would be easier for current crypto experts to learn Rust than for Rust programmers to become crypto experts.

For crypto libraries, a low number of active contributors is particularly harmful. Cryptographic code is always security-critical and therefore should be reviewed, which obviously requires a second active developer on every project. To some degree, the lack of reviewing can be mitigated by having a certain version of the code professionally audited at some point, though none of the existing libraries is currently at this point and it is unclear whether accordingly qualified experts would be available. Furthermore, the design of a library also needs to be reviewed—especially its API design. APIs are contracts between the implementer and the user of a library and need to be designed with the interests of both sides in mind, but when implementers design the APIs themselves as they implement the underlying algorithms, they naturally focus on the implementer's interests. As research has shown (see section 3.3.2) and as will be discussed in chapter 8, bad API design can lead to vulnerabilities, which makes it especially important for cryptographic APIs. In short, **crypto projects require multiple active developers** for reviews and for API design to improve their level of security, though the additional developers do not necessarily have to contribute a lot of code.

5.5.3 Contributor collaboration (RQ 10)

All projects have a main developer who normally acts as a benevolent dictator for life (BDFL).[24] That is, as repository owners, the main developers have the final say when it comes to making a decision, though they cannot prohibit forks. Some make use of this decision power more determinedly than others. For example, `ring`'s owner @briansmith makes strong decisions even against (friendly) opposition, yet never without giving reasons and always open for discussions.[25] On the other hand, `rust-crypto`'s owner @DaGenix accepted pull requests—during the time when he was still active—with relatively few comments (mostly just "thanks") and therefore acted more as a manager of the project rather than a dictator.

This kind of dictatorship or at least central management makes involved development processes unnecessary as long as the number of contributors is low. Survey respondents

[24] Unlike in politics, *dictatorship* does not have negative connotations in the context of open source projects and is a widespread, accepted and effective form of organization.

[25] Example: https://github.com/briansmith/ring/issues/414 (2017-03-23)

characterized the processes as "ad-hoc" and "passive (bugfixes and PRs only)." These options were selected 4 and 3 times, respectively, and none of the other, more structured forms of organization like Scrum or test-driven development were selected. The processes are mostly shaped by the tools of the platform GitHub, where virtually all Rust crypto libraries are hosted. The platform implements a "fork and pull request"-based model, where the issue tracker and the pull requests (PRs) with review functionality provide space for discussions. I will not discuss the technical details, since these parts of the process are no different from other GitHub projects. Note that bug reports and new ideas submitted through the issue tracker also constitute a form of contribution.

As it is customary on GitHub, all projects welcome contributions, many advertise this and some name areas in which support is especially needed. `ring` even has a "good-first-bug" label in its issue tracker to show new contributors where to start. The numbers presented in the previous sections show that active contributors are nevertheless rare. In his talk, @ctz happily reports about a feature where "there is someone, like, actually working on it, who is not me, which is really cool" [Bir17], which shows again that external contributions are rare but genuinely appreciated. Most input from the outside does not come in the form of pull requests but in the form of feature requests in the issue tracker, and it is mostly (though not always) the main developer who then decides about, designs and implements those features. Discussions about the necessity and design take place directly in the issue tracker, if at all.

Some owners try to encourage discussions by posting their intentions and design suggestions in the issue tracker a while before implementing them, both for others to discuss or to take over the work. This can be quantified by the proportion of issues (not PRs) opened by the repository owner. When considering only repositories with at least 5 issues (of which there are only 17 in the Rust crypto ecosystem as of 27 February 2017), a few stand out with particularly high ratios: all issues of @mikeycgto's `message_verifier` and 89% of @burdges's `zerodrop-rs` utility library were created by the respective owners, and some even have titles that ask a design question like "Should we have a `Deref` trait?"[26] The repositories of @briansmith also have many issues created by him,[27] for the major projects `ring` and `webpki` it's more than 78%. On the other end of the spectrum, there are `rustls` (10%), `rust-crypto` (8%) and `rust-openssl` (7%), whose issue trackers are mostly filled with feature requests and bug reports from users. Note that lower numbers are not necessarily worse, they simply indicate that the development is driven or at least initiated by other contributors, or that the main developer simply does not use the issue tracker to attract ideas, opinions and collaborators.

[26] https://github.com/burdges/zerodrop-rs/issues/9 (2017-03-23)
[27] Example: https://github.com/briansmith/ring/issues/419 (2017-03-23)

Besides the interactions on GitHub, there is an IRC channel on the Mozilla network called #rust-crypto, where about a hundred "crustographers" are permanently logged in. Their participation there follows the usual power-law distribution, i.e., only a few of these IRC users regularly participate in discussions. Topics on the chat range from issues concerning the Rust crypto libraries, their development and their usage, over crypto-related topics not specific to Rust, up to very rare off-topic messages. Technical issues regarding the implementation of a specific library are rarely discussed there, though it does happen occasionally.[28] Moreover, there are regular Meetups about Rust in the whole world. The ones in the San Francisco Bay Area, which are recorded and published online, sometimes cover cryptography in Rust.[29] Some Rust crypto developers use Reddit or their Twitter feed to publish news about new libraries, new releases or other big changes.

In conclusion regarding RQ 10, the contributors **collaborate in a way that is typical for small to medium GitHub projects**: the repository owner acts as a dictator and contributions are made in the form of pull requests or reported issues, which is where discussions take place. **Some repository owners follow their own vision more strongly than others, and some facilitate and solicit contributions more than others.** Other communication channels include IRC and in-person Meetups.

5.5.4 API design discussions (RQ 11)

As shown in the previous section, discussions take place on GitHub in issues and pull requests and possibly other private or non-recorded places. To the best of my knowledge, the vast majority of technical discussions regarding API designs happens on GitHub, where the discussed code can be seen and commented on directly. This section analyses those public discussions to find out how often the API design is discussed at all and how often usability (and misuse resistance) are discussed in particular. To this end, I systematically examined all 710 issues and 1001 PRs (collectively called *items* in the following) associated with crypto libraries that have at least 100 items [cf. KGB+16, sec. 4.6]. In decreasing order, those libraries are `rust-openssl`, `ring`, `rust-crypto`, `sodiumoxide` and `octavo`. The analysis was conducted on 22–26 February 2017.

As a first indicator, I downloaded the entire contents of all items and marked the ones that matched the keywords *usability* (or *usable*), *API*, *misuse* and *doc*. These keywords alone produce many false positives (e.g. build problems with "libwinapi" match the *API* keyword) and also many false negatives because API design discussions only occasionally mention these exact keywords explicitly. I manually looked at the title of every item

[28] https://github.com/briansmith/ring/pull/111 (2017-03-23)
[29] https://www.meetup.com/Rust-Bay-Area/events/past/ (2017-03-23)

along with the results of these keyword matches and grouped the items by topic. In some cases, the title was inconclusive and I had to investigate the entire content. From these items, 24 topics emerged. Most items only belong to a single topic, though a few items belong to two.

The 24 topics were further aggregated to eliminate irrelevant fine-granularity in areas not related to API design, resulting in the following, higher-level topics:

- `usability`: related to the usability and misuse resistance of the library (mostly of its APIs)
- `docs`: related to the documentation in the code, readme files and code samples
- `feature`: adding or changing a feature that is visible on the user-facing API, including simple additions of new algorithms
- `apiquestion`: a library user asking a question about the API or a question that boils down to a misunderstanding of the API
- `bug`: bug reports and bug fixes not related to the API
- `tech`: technical changes regarding the build, adaption to other target architectures, unit tests, debugging facilities and error handling
- `internal`: structural or performance improvements done entirely inside the library and changes that are forcibly done to remain compatible to the latest Rust version
- `other`: organizational matters, duplicates, user questions and removal of features and algorithms

In addition, I analysed all items in the `feature` category and evaluated whether the API design, usability and misuse resistance of the added/changed feature were discussed. The proportion of items with any such discussion (no matter how long) is hatched on the `feature` bar in figure 5.6.

The results in figure 5.6 show that `sodiumoxide` and `ring` have more `usability`-related items than the others. They also have the highest proportion of `feature`-related items where discussions about API design take place (45.7% of the respective `sodiumoxide` items and 40.6% of the respective `ring` items). In conclusion regarding RQ 11, **API design discussions do indeed take place on GitHub, though the amount differs between libraries**. From the survey and other observations, there is no indication that major API design discussions take place elsewhere. These results correspond to the stated priorities of the respective developers, which will be discussed in the next section.

5.5.5 Importance of usability (RQ 12)

The previous section showed that `ring` and `sodiumoxide` have more API design discussions in `feature` issues/PRs and they have more `usability`-related issues/PRs on

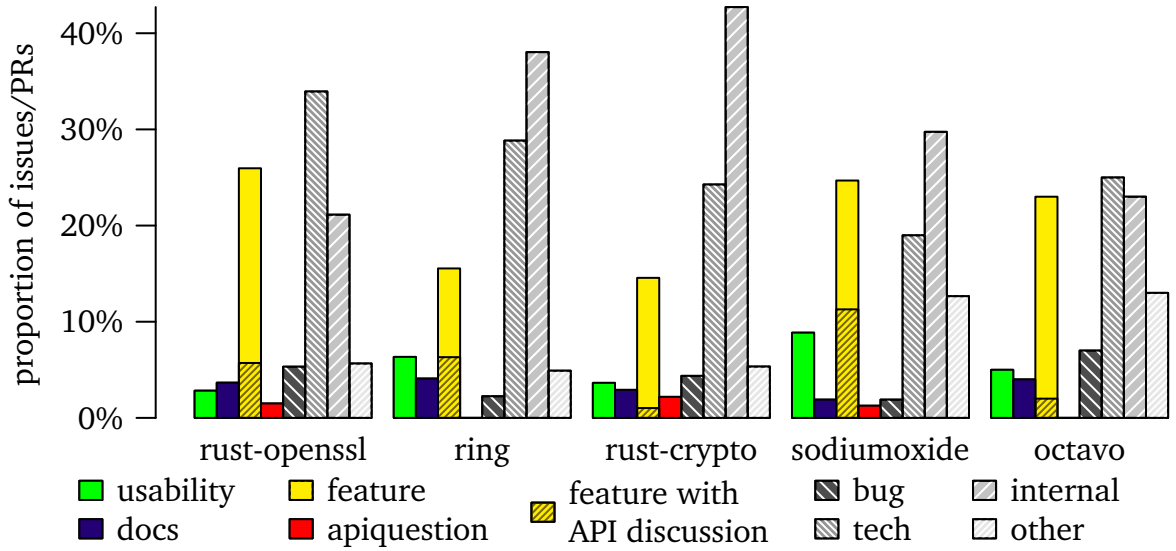

Figure 5.6: GitHub issue and PR topics. Note that a few items have multiple topics, therefore the percentages for each library sum up to slightly more than 100%.

GitHub. Having many usability-related *issues*, in particular, *could* be a sign of usability problems—whether such an issue documents that a particular part of the API is difficult to use or whether it makes a suggestion, it always implies that the usability is or was suboptimal. However, the usability is arguably always suboptimal, especially in the early stages of a project. These usability-related issues also, and more importantly, signal *awareness* of the problem and the will to improve the usability. In fact, many of these issues (36.4% for `sodiumoxide`, 75% for `ring`) were created (and usually solved) by the owner. On the other hand, questions about the API—often initially reported as supposed bugs—do point at unresolved and unknown usability problems. `rust-crypto` has the most API questions (see figure 5.6), which is plausible given its priorities discussed in the following.

To find out their opinion on the importance of usable API designs (with respect to how many end-user applications are vulnerable), the survey asked the contributors directly. Such a direct question biases the respondents to give higher ratings simply because the question is being asked in the first place (if API design was not important, there would be no survey question) and because respondents might want to avoid insulting the (API usability) researcher who evaluates the answers. The survey design tried to minimize this bias by making it the first question about API design and not mentioning the topic in the introduction either, so that participants were asked "out of the blue." In any case, the results can give a rough indication: the majority of contributors (10) said that API design was as important as a secure implementation, whereas only one respondent finds it less important and 8 find it (much) more important, respectively (see figure 5.7).

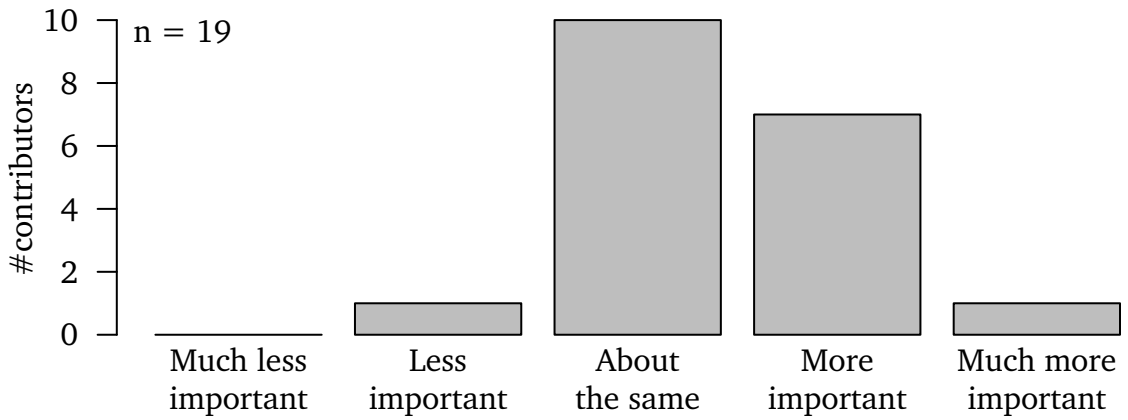

Figure 5.7: Importance of API design (vs. secure implementation) to the contributors

Not all API design discussions lead to an improvement of the library's usability or misuse resistance, as there are other requirements that an API needs to fulfil. When it comes to making API design decisions, the BDFLs' opinions matter more because they ultimately decide. The survey responses from repository owners regarding the importance of usable API designs do not notably differ from the general opinion. However, on the web, several owners make explicit and sometimes detailed statements regarding API design, usability and misuse resistance. In the following, I briefly review the most important statements found during my analysis of all GitHub issues/PRs and further research on the web.

The owner of `rust-openssl` characterizes the API as "rustic but fairly direct bindings" to OpenSSL. Even though many commits during the major overhaul in 2016 did improve API usability, the owner also says that "ideally very few people would interact with rust-openssl directly and instead use wrappers" for higher-level APIs. This implies that he is aware of the importance of usable crypto APIs but puts the project's focus on other goals like feature completeness.

`rust-crypto`'s main developer has a similar take on API design. Even though he acknowledges that "there are tons of great reasons for [making APIs] very hard to misuse" and finds that "ergonomics are extremely important," he wants his project to "focus on creating high quality implementations [...] with idomatic [sic], maximally powerful interfaces without making (too many) concessions to ergonmics [sic]."[30] As an example he names the AES-CBC interface and calls it "yucky to use" and "very unergonomic" but also "extremely flexible." In particular, he does not intend to invest his own time to improve the APIs but would appreciate other contributors doing so.[31] Instead, he wants "easier to use interfaces [to be] in higher level crates,"[30] just like the owner of `rust-openssl`.

[30] https://github.com/DaGenix/rust-crypto/issues/236#issuecomment-72353781 (2017-03-23)
[31] https://github.com/DaGenix/rust-crypto/issues/167#issuecomment-67441572 (2017-03-23)

In contrast, `ring` puts its goal to provide an "easy-to-use (and hard-to-misuse) API" on the front page of the repository. Elsewhere on the web, there are countless other statements of the main developer which underpin this goal (see section 5.3.5 for a few examples). He strictly prioritizes API improvements over API compatibility[32] and is reluctant to add new features without knowing their use case and making sure that they are misuse resistant.[33] There is a frequently used `usability` label in the issue tracker, though it is used on several issues that I did not associate with usability during my analysis. Even so, usability is not the only major API design goal. For example, `ring` often has to make compromises because it aims to support heap-less environments (where no `Vec`s can be allocated, for instance).

`sodiumoxide` naturally inherits the focus on API design from the underlying libsodium library. According to the main repository page, it aims to be "just as easy to use."

To summarize regarding RQ 12, the survey responses and statements of repository owners show that **the Rust crypto community is generally *aware* of the importance of usable API designs**. While some libraries (most notably `sodiumoxide` and `ring`) make usability an explicit and important goal, others (`rust-crypto` and `rust-openssl`) consciously focus on other goals. The **varying importance of usable API designs** is reflected in the usability and misuse resistance of these libraries, as will be shown in chapter 7.

5.5.6 Threats to validity

Firstly, this analysis of GitHub data serves to describe and quantify the collaboration of the contributors. In particular, no conclusions can be drawn about the resulting quality of the libraries from this data alone, e.g., more commits, more contributors or more discussions about usability do not imply more features or better usability. There are many ways a well-thought-out and previously discussed piece of code can reach the `master` branch of a repository without the discussions being visible in its issue tracker or a pull request. I confirmed that obvious alternatives like external review tools or comments on the commits themselves (rather than the PRs) do not play a role. However, most repositories have a large set of commits (mostly from the early phases of the library) that were pushed directly to the `master` branch. Even though these commits can positively or negatively affect the library's usability, the analysis method does not capture them at all and it remains unclear whether the respective changes have been discussed offline, in another (private) repository or elsewhere.

[32] https://github.com/briansmith/ring/issues/445 (2017-03-23)
[33] https://github.com/briansmith/ring/issues/414 (2017-03-23)

Secondly, the manual analysis of issues and PRs only covered the major crypto libraries and therefore it only serves to picture the core of the crypto ecosystem and cannot be generalized to all Rust crypto libraries. The smaller libraries have disproportionally fewer discussions, mostly none at all, and many of them seem to be one-time uploads of private projects. The described ways of developer collaboration obviously do not apply to such projects.

Thirdly, there are various pitfalls generally associated with analyses of GitHub data [KGB+16]. Some of them do not apply to the presented analysis because of its manual nature, e.g., the set of libraries definitely does not contain private repositories only used for file storage because the libraries were discovered through their published crates on crates.io and unfitting projects were excluded manually. Kalliamvakou et al. [KGB+16, sec. 4.5] point out that GitHub might not be the only platform that a project uses for issue tracking, reviews, discussions, planning and so on. As already discussed in section 5.5.4, the use of other communication channels and IRC in particular cannot be ruled out, though it seems that discussions on the technical level of APIs and implementations mostly take place on GitHub—closer to the discussed code.

Lastly, GitHub ignores commits without an associated account when computing the contributors page [see KGB+16, sec. 4.7]. This data was used in the contributions analysis (section 5.5.1). To uncover significant omissions, I compared the total number of commits reported through the contributors page with the total number of commits on the `master` branch according to Git itself. Four smaller libraries have more than 10% (but less than 50%) of their commits coming from unidentified authors, which does not distort the analysis too much. In the case of `rust-crypto`, however, 68% of the commits come from users without an associated GitHub account, mostly because the main developer used a different Git user. This was manually corrected using the output of `git shortlog -s -n -no-merges` in a cloned `rust-crypto` repository.

5.6 Crypto in the standard library

Currently and in the foreseeable future, there is almost no cryptographic functionality built into the Rust standard library because of a consensus in the Rust community that the standard library should generally be kept small [Sap+16]. As an alternative, the Rust project maintains official crates under its `rust-lang` GitHub organization (so-called *rust-lang crates*) as well as the `rust-lang-nursery` for new crates [TC15]. For example, the `std::rand` module is deprecated in favour of the `rand` crate. It contains several RNGs and the documentation points to `OsRng` for cryptographic purposes, explicitly.

There is also a `std::hash` module in the standard library, but, confusingly, it is not suitable for cryptographic purposes. The only implementation at the moment is SipHash,

which is characterized as and proven to be "cryptographically strong" by its inventors [AB12]. However, they have hash maps in mind as the primary use case. To protect users from Denial of Service (DoS) attacks based on hash-flooding, a hash function needs to guarantee that *multi*collisions (many inputs that all hash to the same value) cannot be computed easily, as they would degenerate the structure of a hash map. For most other cryptographic use cases, it is crucial that not even a *single* collision or second preimage can be found. No fast hash function can guarantee this as a matter of principle if its output size is only 64 bits, because the output space can be brute-forced entirely with enough computing power. Since the `std::hash::Hasher` trait hard-codes **`u64`** as the output size, it can and will never be suitable for cryptographic purposes. Unfortunately, the only warning about this is in the documentation of the `SipHasher` implementation, which is discouraged to be referenced directly, as users should use `DefaultHasher` instead (where there is no warning). This potential for misunderstandings and unintentional misuses could be mitigated by warning messages on the module-level and by moving the module closer to its only intended application, e.g., `std::collections::hash`.

Before the rust-lang crates were introduced, there was a discussion about including cryptographic functionality into the standard library.[34] The opening post starts with a differentiation between crypto written in Rust itself and bindings to other libraries. After some discussion of the various alternatives (OpenSSL, NaCl, libsodium), @tarcieri proposes an "abstract, high-level API" to be included in the standard library—similar to Java's provider architecture (JCA). The advantage is that no particular implementation needs to be shipped with the standard library itself, allowing implementations to be updated faster and to be swapped out more easily. It also keeps up the need for competing implementations, which is important to keep an ecosystem alive as the examples in chapter 4 have shown. Last but not least, @taoeffect points out that such a generic API "allows you to focus on perfecting the API itself without the distractions of the actual implementation," i.e., it is beneficial for API usability. Section 5.7.3 will discuss the potential creation of such a generic interface.

5.7 Areas for improvement

After analysing all populations of Rust's crypto ecosystem, possible improvements can be derived. This section gives an overview of all areas of improvement, whereas the remainder of the thesis focuses on one particular aspect, namely the usability.

[34] https://github.com/rust-lang/rust/issues/14655 (2017-03-23)

5.7.1 Developer recruitment

As discussed in section 5.5.2, the Rust crypto ecosystem suffers from a lack of contributors, especially because API design and reviews of cryptographic code fundamentally need multiple developers to look at the same code. Currently, all projects only have one major contributor. Both in the survey responses and on the web, the lack of contributors has been attributed to a lack of time and funding. And clearly, one approach is to somehow **attract funding** to support developers who work on Rust crypto so that their contributions are not limited to their unpaid spare time.

Instead of donating money, a company like Mozilla, who is already using Rust productively, can invest the working time of their software engineers in the crypto ecosystem because they need certain cryptographic functionality. For instance, MaidSafe improved `sodiumoxide`'s build system and published the result in `rust_sodium`. Such funding is motivated by but also limited to the needs of the respective company—and general usability of the library is probably not one of the urgent needs. In addition, the engineers are not necessarily cryptography experts, whom the Rust crypto ecosystem primarily needs. And most importantly, this kind funding will increase as Rust itself gains popularity and a bigger user base, but by that time, it is already too late to ensure good usability and misuse resistance.

Getting current Rust developers to work on crypto code is one way to recruit Rust crypto contributors. The other obvious one is to get cryptographers to work with Rust. Fortunately, **Rust is becoming increasingly attractive for cryptographers** to implement their algorithms in, because it is a memory-safe, low-level programming language. Any attempts to further push this trend therefore have the potential to leverage the world's crypto expert knowledge for the Rust ecosystem. One survey respondent proposes to advertise Rust in this context and names the ability to export a C-compatible foreign function interface (FFI) as another advantage, making crypto algorithms implemented in Rust available in many other ecosystems.

Universities combine both advantages: they can provide funding like companies and they can give their employees the freedom to work beyond what is required to build a product, resulting in more contributions to the public (crypto) ecosystem. As the Rust language itself and cryptography in general are suitable subjects for research, it should be possible to create positions at the intersection. Fortunately, this is already starting to happen.[35]

Besides contributors, software developers who act as users can become more involved and help improve the Rust crypto libraries—particularly their usability—by asking

[35] https://twitter.com/sahuguet/status/839198110819762177 (2017-03-23)

questions, reporting unclarities and helping with the documentation. User feedback can be increased in two ways: gain more users and gain more feedback from existing users. **To motivate users to give feedback, libraries could ask for it on their repository and documentation pages**, just like they recruit contributors there. To attract more users beyond the natural demand for crypto in Rust, the Rust ecosystem as a whole could direct attention towards the crypto ecosystem, e.g. by **including crypto-related examples and tasks in tutorials, contests** and so on.

Once more developers join the Rust crypto libraries, more involved processes regarding reviews, planning and coordination will become necessary. For the time being, the relatively unstructured processes are suitable for the low number of contributors.

5.7.2 Platform

In the contributors survey, many respondents named features missing from the Rust language and platform that constrain the implementation of cryptographic algorithms and APIs.

Constant-time operations: To prevent timing-based side-channel attacks, many cryptographic algorithms have to ensure that their runtime does not depend on the input data. Three main steps are required to reach this goal:

- Firstly, the algorithms have to be implemented without branches, loops or table lookups dependent on secret data. The Rust platform cannot reasonably facilitate writing such code, but it has been suggested (and rejected for complexity reasons) that the *compiler could verify the constant-time property* on annotated blocks.[36]
- Secondly, all basic operations (comparisons, additions, etc.) have to run in constant time. A working solution is to use assembly language code, which can be inlined in Rust programs with the `asm!` macro. There are several libraries that encapsulate constant-time operations and it has been proposed to *add constant-time functions to the standard library*.[37]
- Thirdly, and most frequently wished for, the Rust compiler must not optimize the code for performance, which can easily destroy the constant-time property. While this can be prevented by providing assembly code directly using the `asm!` macro, Rust crypto contributors wish for a way to tell the compiler to *turn optimizations off* through an annotation.[36]

[36] https://github.com/rust-lang/rfcs/issues/847 (2017-03-23)
[37] https://github.com/rust-lang/rfcs/issues/1814 (2017-03-23)

Type-level numbers: Array types in Rust always include the array length, e.g. `[u8; 16]` could be a 128-bit key. As arrays are stored on the stack, their length needs to be known at compile time. For flexibly sized data, functions either have to reference data stored elsewhere using a slice (`&[u8]`) or use a vector stored on the heap. Many crypto libraries want to support heap-less environments and slices usually cannot be used for return values, so neither of these solutions work. In cases where the length is not arbitrary but one of a few options (e.g. key lengths are usually 128-bit, 192-bit, 256-bit, etc.), generics can be used instead. That is, the respective function has a generic parameter `L` which represents a number, and `[u8; L]` is used as the array type. At compile time, multiple variants of the function are generated from the same code, one for each length that is used anywhere. The `generic-array`[38] crate already emulates this functionality, but ideally it would become part of the Rust platform itself, most likely by making *associated constants* work in array declarations[39] or through *pi types*.[40]

AES-NI and single instruction, multiple data (SIMD): Survey respondents wished for platform support for these instruction sets of modern processors to improve performance. Since then, the `simd` crate by @huonw has been promoted to the `rust-lang-nursery` repository.[41]

To answer RQ 13 regarding existing and missing platform features in more detail, I complement the survey responses above with additional features I encountered during my research in general and through a search for "crypto" in the official `rust-lang` repositories in particular.

Clear-on-drop and immovable types: To prevent accidental leaks, it is best practice to erase cryptographic keys and other sensitive data from memory once they are not needed anymore. Rust provides a `Drop` trait whose only method is called when the value goes out of scope, making it the perfect place to clear the memory before deallocation. However, Rust's memory model only guarantees safety and does not rule out memory leaks. Several crates try to get the many subtleties right and provide a wrapper type that clears its contents reliably.[42] Multiple requests for comments (RFCs) unsuccessfully proposed built-in attributes for the Rust language itself, so that the compiler could guarantee it. Regardless of the technical solution (attribute, wrapper type, compiler support or not), it should eventually become part of the platform, e.g. as a rust-lang crate. At the time of writing, support for immovable types seems to be the blocking issue.[43]

[38] https://github.com/fizyk20/generic-array (2017-03-16)
[39] https://github.com/rust-lang/rust/issues/34344 (2017-03-23)
[40] https://github.com/rust-lang/rfcs/issues/1930 (2017-03-16)
[41] https://github.com/rust-lang-nursery/simd (2017-03-23)
[42] For example: https://github.com/cesarb/clear_on_drop (2017-03-23)
[43] https://github.com/rust-lang/rfcs/pull/1858 (2017-03-23)

Secure randomness: Devlin [Dev14] recommends a CSPRNG that simply passes through the secure randomness generated by the operating system and that documents these sources well. The official `rand` crate fulfils this role perfectly today.

Testing mode: Indela et al. [IKND16] propose a strict compile-time distinction between the development environment, where workarounds like short keys or disabled certificate validation are permitted, and the production environment, where they must lead to compiler errors. Green and Smith [GS16] call this a "testing mode." The `cargo` build tool already provides separate *debug* and *release* builds which could be sufficient. Green and Smith [GS16] suggest that the testing mode is enabled per machine/device (identified by some ID). Dedicated platform support for such a testing mode would be helpful because it makes the functionality more explicit and because a developer might want to test something in a *release* build, as well.

Rustdoc: There are currently a few issues that make navigating the generated documentation pages difficult. Note that these issues were discovered during the experiments (see chapter 7) and were not mentioned in the survey. They also do not hinder the implementation of good crypto libraries, but solving them would result in significantly better usability of all crypto libraries.

- The search function does not search everything. For instance, searching for "hmac" in `rust-openssl`'s documentation[44] does not find the code sample in the `sign` module, even though the term is mentioned there multiple times.
- docs.rs is not optimized for external search engines. Google searches for the documentation of any library (e.g. "rust-openssl documentation") only turn up older versions of the documentation at low ranks in the search results, whereas the top ranks are occupied by the repository, crates.io and many unrelated pages.
- Modules that only consist of a re-export (which happens frequently when libraries expose multiple primitives or algorithms under a single umbrella, see sections 8.4.3 and 8.5) have virtually empty documentation pages[45] even though all the re-exported items can be used normally in the code. This situation should be detected and the documentation should either be imported as well, or there should be a highlighted link to the respective documentation page.

5.7.3 High-level crypto API

The idea of a generic, high-level API that plugs into various crypto libraries has been discussed for a long time (see section 5.6) and libraries like `rust-openssl` or `rust-crypto`

[44] https://docs.rs/openssl/0.9.10/openssl/?search=hmac (2017-04-02)
[45] Example: https://docs.rs/octavo/0.1.1/octavo/crypto/index.html (2017-04-02)

have intentionally low-level APIs as they expect wrapper libraries to take care of the high-level interfaces for them (see section 5.5.5). There is a subtle difference between these two ideas, though. A **generic *interface* consisting of traits** would be created either way and could be incorporated in the standard library or live in an **officially endorsed rust-lang crate**. One approach is to rely on all crypto libraries to implement this interface, which is more likely to happen if it is officially standardized. The alternative approach is a separate crate per existing crypto library that acts as a wrapper and implements the interface. Although such a wrapper would introduce an additional layer of code written and maintained by developers with less cryptography experience, the high-level API would probably prevent more vulnerabilities through misuses than the wrapping layer introduces. The first approach is preferable if the library creators are willing to adapt their libraries.

Either way, the new high-level API should be designed to last as long as possible. Because some crucial Rust language features for a perfectly ergonomic crypto API are currently missing, as discussed in the previous section, the **creation of this high-level API should be postponed** until they become available. The proposals for usable Rust crypto libraries presented in chapter 8 will hopefully still be valid by then.

5.7.4 Usability

Perhaps the most important area of improvement is the crypto ecosystem's usability. The usability as a whole depends on multiple factors:

- Do users find *the right library* to use? Depending on the use case, a higher- or lower-level interface is more suitable, and depending on the cryptography skills of the user, misuse resistance plays a significant role. As discussed above, a generic, high-level crypto API could eventually become officially endorsed. Until then, an overview of the major crypto libraries along with directed recommendations could prevent users from using a dangerously low-level one.
- Do users find and *understand the documentation*? Ideally, the first contact with the chosen library is the documentation rather than the API itself. Section 8.1 will discuss best practices for the documentation.
- Do users find *the right primitive* and choose *a suitable algorithm* (if they do not have specific requirements)? Sections 8.2 to 8.4 will discuss abstraction levels and organizational structures for primitives and algorithms.
- Do users use the API *correctly*? The remainder of chapter 8 will discuss various technical details regarding usability and misuse resistance.

The latter two concern the structure of the library and its API design, both of which are **difficult to change once the library has matured**. Therefore, these usability concerns are more urgent than the former two (ecosystem overview and documentation) and

more urgent than other possible security improvements, particularly the cryptographic algorithm implementations themselves. Before the recommendations to improve the usability of crypto libraries in Rust, which are presented in chapter 8, the following chapters report on two analyses to inform these recommendations: chapter 6 investigates where and how crypto functions are currently used in Rust code, and chapter 7 experimentally evaluates the usability of the major existing crypto libraries.

5.8 Conclusion

The Rust crypto ecosystem consists of surprisingly many, manifold libraries, four of which were identified as major libraries based on the number of dependent crates (`rust-crypto`, `rust-openssl`, `sodiumoxide` and `ring`). All of the major libraries provide multiple cryptographic primitives and therefore fall in the same category. The contributors in the Rust crypto ecosystem are demographically average software developers with above-average programming experience and a few experienced cryptographers among them. Their motivation is typical for open source developers and many of them like Rust for its desirable properties. Most contributors have well below 5 hours per week to work on Rust crypto in their unpaid free time, though very few invest more time. Consequently, there is a general lack of developers and especially cryptographers, and most libraries only have one main contributor who outweighs all others. This makes complicated development processes unnecessary—most projects use the typical GitHub workflow and discuss new features in issues and pull requests, if at all. Although the contributors generally recognize the importance of usability and misuse resistance in cryptographic APIs, only some of them see it as a major goal for their library. Besides API usability, which is discussed in the following chapters, possible improvements include several measures to recruit more contributors, several technical additions to the Rust platform to facilitate the implementation of cryptographic libraries as well as the development of a common high-level crypto API after some desirable changes to the Rust platform have been released.

6 Usage analysis

One of the guiding API design principles is simplicity, also known as the KISS ("keep it simple, stupid") principle. However, there is a broad variety of use cases and requirements for cryptographic APIs. To only name a few examples: users need different kinds of ciphers, some need certain key lengths or IVs while others do not want to deal with such decisions, some need to avoid heap allocations at all costs, a few want to attach "associated data" when using an AEAD cipher whereas others are confused by too many parameters, and so on.

Clearly, an API cannot be simple and perfectly tailored to all those use cases at the same time. One approach to resolving this conflict is to first focus on the *80% use case* [Kau11; Kob12]—reminiscent of the Pareto principle—and to design an API for those users. In a second step, it needs to be ensured that the remaining 20% of users can reach their goals in a reasonable way, too, possibly by adding or modifying API elements. These changes must not confuse or otherwise interfere with the 80% use case and there are other important considerations like performance and maintainability. In order to design a usable API or improve an existing one, it is therefore essential to understand where and how it is going to be used, that is, to quantify and understand the "use" relation between users and libraries in the ecosystem (see figure 4.1 on page 38) and to find out what the 80% use case encompasses.

This chapter reports on a manual analysis of current usages of cryptographic primitives in publicly available Rust code, which addresses RQ 14 (see page 46). Section 6.1 describes how the analysis was conducted, section 6.2 presents overall results and the further sections 6.3 to 6.5 contain more detailed analyses of particular API types, namely hashing, HMAC and symmetric encryption. Finally, section 6.6 discusses, among other threats to validity, a significant bias that is inherent to this type of analysis and needs to be considered when using the results for API design.

All numbers and figures presented in this chapter were generated by a collection of R scripts from a single raw data file. The raw data in `xlsx` format as well as the scripts and their outputs are available for download (see appendix A).

6.1 Approach

For each of the libraries introduced in section 5.3, I did the following (on 4 and 5 December 2016):

1. Consider all dependent crates (crates which use the library) on `crates.io` and filter out crates:
   - which have been yanked (retracted),
   - which are cryptographic libraries and offer a (higher-level) crypto API themselves (these are not interesting because their developers have to be knowledgeable in cryptography and do not run the risk of misunderstanding or misusing the API in question),
   - which are only dependent according to their `Cargo.toml` but do not actually use the library anywhere in their code,
   - which are not easily searchable (e.g. because they do not have a `master` branch or because they are hosted on a platform other than GitHub which does not offer code search).
2. Open the repository linked on `crates.io`, if available, or search for the corresponding repository manually.
3. Search the code for "**`extern`** `crate <library name> [`**`as`**` <alias>]`" (where the alias defaults to the library name if the **`as`** clause is not present).
4. Search the code for "**`use`** `<alias/library name>`" and inspect all relevant matches:
   - Open the matching file and switch to the `master` branch.
   - Inspect all matching **`use`** clauses as well as the lines where they are referenced and categorize the functionality which is used there.

To find additional dependent crates not listed on `crates.io`, I did the following for each library (on 13 and 14 December 2016):

1. Globally search for "**`extern`** `crate <library name>`" on GitHub.[1]
2. Sort the results to show most "recently indexed" usages first.
3. Filter out matches:
   - which were previously discovered as a dependent crate (see above),
   - which are cryptographic libraries and offer a (higher-level) crypto API themselves,
   - which do not actually use the library anywhere in their code,
   - which are solutions to the Advent of Code.[2]

[1] There are some limitations to global code search on GitHub (https://help.github.com/articles/searching-code/#considerations-for-code-search), though none of those seriously impact the exploration of library usages. Searching for "**`extern`** `crate crypto`" in particular also yields results like `crypto_hash`, `crypto_tests` and so on, which were identified and excluded/subtracted manually.

[2] https://adventofcode.com/ (2017-03-23); Because the analysis was conducted in December and sorted to show recent usages first, there were quite many repositories with solutions to the artificial

4. Go through the results one by one until either 20 proper usages are found or all search results have been considered.
5. Analyse and categorize the usages as described above.

Those *usages* can be split into the following categories:

- Random number generators (RNG)
- Collision-resistant hashing
  - MD5
  - SHA-1
  - SHA-2
  - Other hash (Blake2b, RIPEMD-160, Whirlpool)
- Message authentication codes (MAC)
  - HMAC[3]
  - Poly1305
- Password hashing and key derivation (KDF)
  - bcrypt
  - scrypt
  - Argon2
  - PBKDF2, HKDF[3]
- Unauthenticated symmetric encryption ("symm")
  - AES block ciphers (ECB, CBC)
  - Other block ciphers (Blowfish)
  - RC4
  - AES stream ciphers (CFB, CTR)
  - Salsa20, XSalsa20, ChaCha20, XChaCha20
  - Other stream ciphers (Sosemanuk, HC-256)
- Authenticated symmetric encryption ("symm (auth)")
  - AES-GCM
  - ChaCha20-Poly1305
- Asymmetric cryptography ("asymm")
  - RSA
  - Elliptic curve cryptography (ECC) (Curve25519, Ed25519)

There are more algorithms that fall into these categories, but this list only contains the ones that were found to be used. Note that a single dependent crate can have multiple usages of different kinds. Every dependent crate is counted at most once per category.

Advent of Code puzzles at the top. These solutions include `rust-crypto` or `rust-openssl` and use the MD5 function in one of a few particular ways. Only 10 of these repositories were analysed and each kind of usage was counted only once (see section 6.3).

[3]If a hash algorithm is only used inside an HMAC, the use of the inner algorithm is not counted separately because the respective API is not used. The same applies for HMACs used inside a PBKDF2 or HKDF.

6.2 High-level results

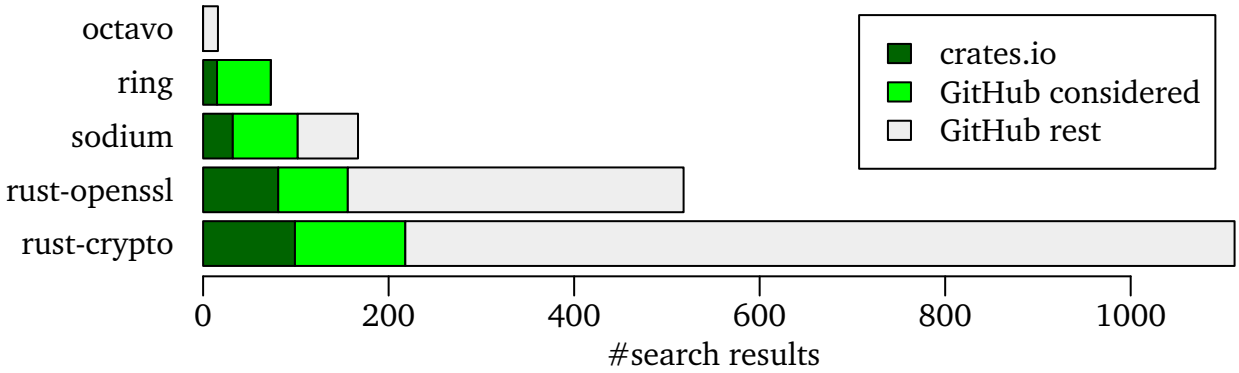

Figure 6.1: Number of considered search results per library (before filtering and duplicate removal)

As can be seen from figure 6.1, `rust-crypto` and `rust-openssl` are used significantly more often than the others. Because the analysis was performed manually, only a fraction of the 1659 GitHub results (namely 322) could be considered (this number was not predetermined; it is simply the number of GitHub results that needed to be processed before 20 new, unique and proper usages per library had been found). In addition, all 227 results from `crates.io` were considered for the next step. Note that `sodiumoxide` and `rust_sodium` are combined into a single item called `sodium` and `octavo` was not considered at all because it had less than five relevant dependent crates.

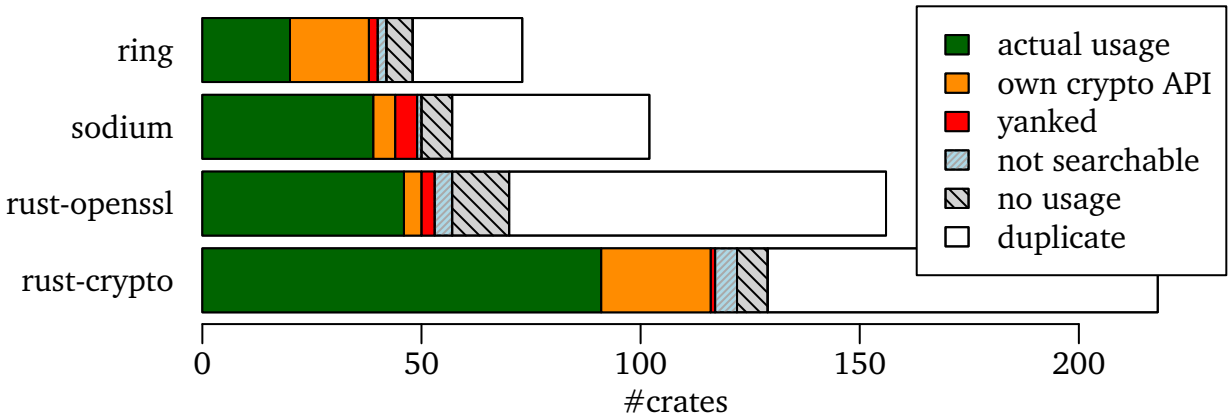

Figure 6.2: Filtering the crates found in the previous step

Figure 6.2 shows the effects of "filtering" all the considered search results according to the criteria detailed in the previous section. The GitHub search led to 245 duplicates, which were already known either from `crates.io` or a previous GitHub search result. In total, 196 unique crates were classified as "actual usage" and all following analyses are based only on this set of crates.

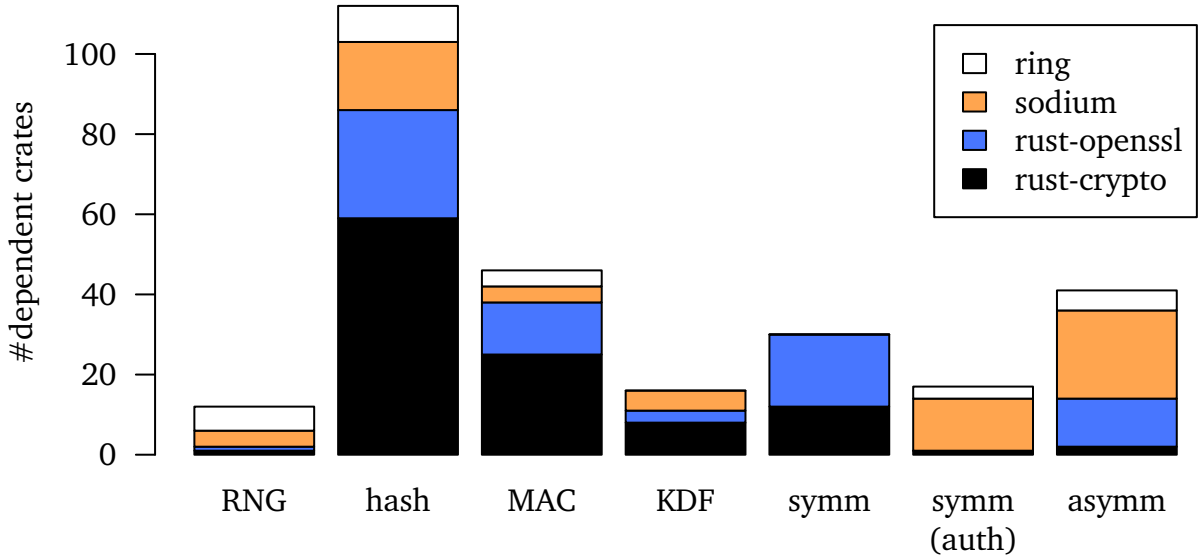

Figure 6.3: High-level usages per category and library

Of the 196 analysed crates, more than half (112) use a hash function (see figure 6.3) and 71 use nothing else. Hence, more than a third of all dependent crates only depend on a crypto library because they need a hash function. While hashing is the most important use case for each individual library, there are some differences with respect to the other categories. Overall, the second most popular category is MAC (46), followed by asymmetric (41), unauthenticated (30) and authenticated (17) symmetric encryption, some of which outweigh the MAC usage in `sodium` and `rust-openssl`. When counted together, the two symmetric encryption categories outweigh MAC or asymmetric encryption. `sodium` and `ring` do not offer unauthenticated encryption at all, forcing users to use authenticated encryption or switch to another library. It seems that many users could be convinced to do the former, considering that `rust-crypto` and `rust-openssl` do not have disproportionally many symmetric encryption users overall and almost none of their users decided for authenticated encryption.

The following sections analyse the most popular categories (hashing, HMAC and symmetric encryption, both authenticated and unauthenticated) in more detail to gain insights for API design.

6.3 Hashing

Hash functions are not only the most widely used cryptographic primitive but also the simplest: the input is a bit stream of arbitrary size and the output has a fixed size—there are no keys, tags or the like. However, there are different formats in which the data can be represented and multiple pieces of input data can be combined, if necessary.

For every dependent crate that uses the hash API directly (that is, not only inside an HMAC), I did the following to analyse the usage in more detail:

1. Open all matching files identified in the previous step (see section 6.1).
2. Find the code parts (usually only one) where the hash API is used.
3. Determine whether there is a single piece of input data or multiple ones (the number of calls to the respective `input()` function, or similar).
4. Determine the present input data type(s).
   - If the data needs to be converted beforehand, determine the original data type *before* all conversions.
5. Determine the desired output data type.
   - If the digest value is converted afterwards, determine the eventual data type and format *after* all conversions.

There are the following input data types:

- "slice": The data is available as a `&[u8]` function parameter or variable.
- "arr": The data is available as a `[u8]` array on the stack.
- "vec": The data is available as a `Vec<u8>` on the heap.
- "int": The data is an integral number, e.g., `u8`, `u32`, `u64`, and so on.
- "&str": The data is available as a `&str` parameter or `&'static str` string literal.
- "String": The data is available as a `String` on the heap.
- "Read": The data is read from a source that implements the `std::io::Read` trait. This is usually achieved with a helper function[4] that reads until no more data is available or with the `std::io::copy()` function if the hasher implements the `std::io::Write` trait.
- "Write": The hash implementation supports the `std::io::Write` trait and the `write/write_all/write_fmt` methods are used directly (without `std::io::copy()`).

The output data types can be categorized as:

- "arr": The hash value is stored in a fixed-size `[u8]` array on the stack.
- "vec": The hash value is stored in a `Vec<u8>` on the heap.
- "slice": The hash value is stored in an array or vector and it is directly and only passed on to a function that accepts a `&[u8]` slice.
- "hex": The hash value is converted to hexadecimal format and stored as a `String` on the heap.
- "base64/base58": The hash value is converted to Base64 or Base58 format and stored as a `String` on the heap.
- "comp": The hash value is stored in any format, but the value is only used for comparison with either a fixed value (which could easily be converted into any format) or another hash value that was computed the same way.

[4] https://github.com/NicolasDP/git/blob/4073194/src/protocol/hash.rs#L131 (2017-03-23)

Conversions are "excluded" from the analysis because the goal is to find out the best possible API for the respective use case, which would not require any conversion but would accept the data in the very format that the user already has. For example, this `rust-openssl` user[5] converts a `String` to a byte slice using the `as_bytes()` method and converts the resulting `Vec<u8>` to a Base64 `String` using the `to_base64()` method from the `rustc_serialize` crate (type annotations added):

```
// self.key is of type String
let res: Vec<u8> = hash::hash(Type::SHA1, self.key.as_bytes());
let response_key: String = res.to_base64(STANDARD);
```

This usage is counted as a single "String" input and "base64" output.

This user,[6] on the other hand, implements a helper function that hashes a `String` and a `&[u8]` slice and returns the result as a `Vec<u8>`:

```
fn hash(app_id: String, data: &[u8]) -> Vec<u8> {
    let mut hasher = Sha1::new();
    let mut output = vec![0x0; hasher.output_bytes()];
    hasher.input(app_id.as_bytes());
    hasher.input(data);
    hasher.result(&mut output[..]);
    output
}
```

Helper functions like this are always counted towards the data type categories indicated by their signature because a helper function around a library function indicates that the library user was dissatisfied with the API and built the desired API around it. Therefore, this usage is counted as multiple inputs ("String" and "slice") and "vec" output.

The discovered data types are aggregated as follows: A single dependent crate which uses the hashing API in multiple places or which has multiple pieces of input data can count towards multiple categories (all the ones that occur in the code), but every crate can only be counted once per category. That is, multiple similar or identical uses in the same crate are not counted twice. In many cases, these are merely code clones that could be avoided with better abstraction or helper functions.

6.3.1 Results

Only 10 hash API users (9.8%) have multiple pieces of input data. By far the most common input data type is the byte slice (42.2%, see figure 6.4 left). It is also the primary data format accepted by all libraries (see table 6.1) because it is the most

[5] https://github.com/jfager/d3cap/blob/f0c4/src/json_serve/src/rustwebsocket.rs#L63 ('17-03-23)
[6] https://github.com/manuels/bulletinboard-dht/blob/cf51/src/dbus_service.rs#L49 (2017-03-23)

flexible: a slice can reference an entire vector or array or only part of it, and a slice can be used with the `Deref` and `AsRef<[u8]>` traits to read from structs. A `&[u8]` parameter is the standard way to "borrow" a read-only sequence of bytes from the caller. Hence, slices also cover the "vec" (12.5%) and "arr" (3.9%) use cases, which are not supported directly by any library.

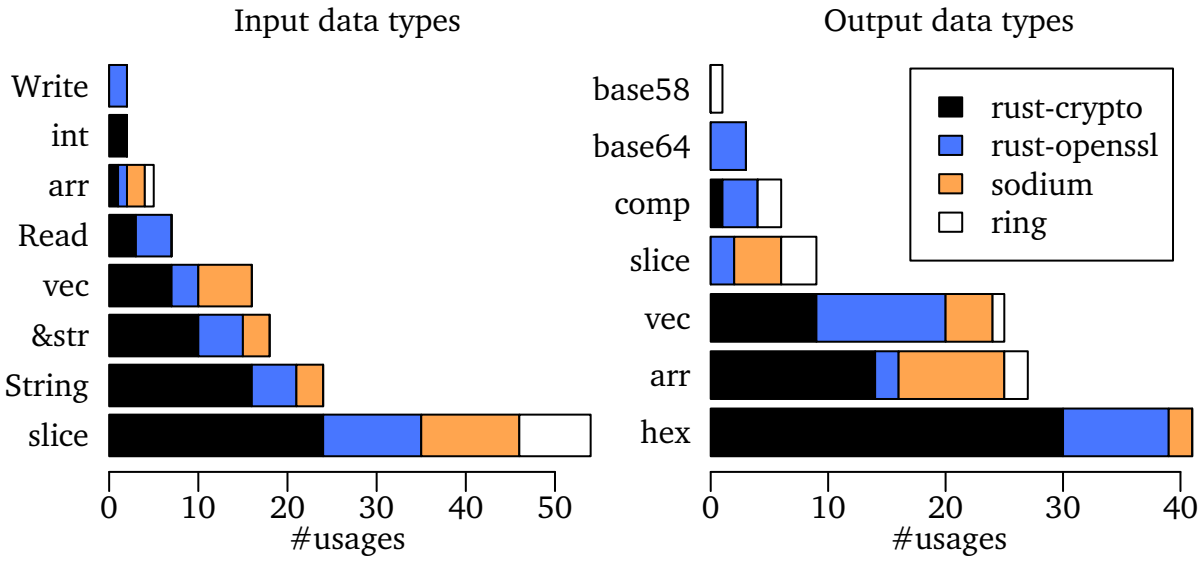

Figure 6.4: Hash function usage

The second and third most common input types (32.8% together) are strings (either as a heap-allocated `String` or a string slice `&str`). It does not make sense for a hash function to accept a `String` parameter because it would unnecessarily take ownership of the data on the heap. The standard way to borrow a `String` is a `&str` parameter, which `rust-crypto` implements in its `input_str()` method (see table 6.1). All other libraries do not accept strings, so the caller must convert them manually.

The remaining input types are the `Read` and `Write` traits (5.5% and 1.6%, respectively). No library accepts `Read` values directly, but `rust-openssl` implements the `Write` trait and `Read` values can easily be copied over.

The most common output data types are hexadecimal strings (36.6%, see figure 6.4 right), `[u8]` arrays on the stack (24.1%) and `Vec<u8>` vectors (22.3%). While `rust-crypto` provides a `result_str()` method for hex values, users of all other libraries have to resort to the `rustc_serialize` crate for conversion to hex/Base64/Base58.

Each library has its own approach to returning the hash value as a byte sequence (see table 6.1). `rust-crypto` does not have a return value at all but takes a `&mut [u8]` parameter which can write out to a vector or an array. `rust-openssl` allocates and returns such a vector itself. `sodium` does almost the same with an array but wraps it in a

		rust-crypto	rust-openssl	sodium	ring
Multiple inputs possible		✔	✔		✔
Input	slice	✔	✔	✔	✔
	`&str`	✔			
	Read				
	Write		✔		
Output	hex	✔			
	base64/base58				
	`&mut [u8]`	✔			
	vec		✔		
	arr			✔	
	slice (`AsRef<[u8]>`)			✔	✔
	comp			✔	

Table 6.1: Hash function data types accepted/produced by current libraries

`Digest` struct. This requires a separate `Digest` type for every hash function (because they have different output sizes), but it comes with several advantages: no heap usage, type safety, expressiveness of the API and the possibility to implement various traits on that value. For example, the `PartialEq` implementation allows the digest to be compared to others simply with the `==` operator. `ring` also returns a custom `Digest` type but currently only implements `AsRef<[u8]>` for it, which requires the user to call `as_ref()` for all comparisons and further conversions.

When comparing the data types that users have/need (figure 6.4) with the ones that the libraries accept/return (table 6.1), there are some obvious discrepancies. For example, many users have strings as their input and most users need their hash value as a hexadecimal string, but only one library accepts and returns strings. It is worth pointing out that this discrepancy is not necessarily a concern: as long as *most* users can *easily* reach their goals with little code and *all* users can reach their goals with *reasonable* effort, the API is good enough. And most string conversions require nothing more than an in-line call to a single conversion function.

Together, the completely broken algorithms MD5 (27.7%) and SHA-1 (18.5%) account for almost half of all hashing usages. It remains unclear whether these usages would require a cryptographically secure hash function (and thus result in vulnerable applications) or if they simply need any hash function. This question could be answered in future work (see section 9.2) and result in separate MD5 and SHA-1 crates for non-cryptographic purposes.

6.4 HMAC

All MAC users use some HMAC-SHA variant. Other kinds of MACs like Poly1305 are only used by crates that were excluded because they expose a higher-level cryptographic API themselves, that is, they are currently only relevant for usage by crypto experts.

The approach is analogous to the one for hashing (section 6.3), but there are four points of interest now:

- the data type of the *key*,
- the *input data* type,
- the *transfer format* used to send, receive or store digest values (if the computed digest remains within the same process, i.e., there is no actual transfer happening, this format is simply the data type of the digest variable or return value; it is thus comparable to the hash function output type),
- the *verify or comparison function* used to check if a signature is valid or if two digests match.

The verify/comparison functions can be categorized as:

- "built-in verify": `sodium` and `ring` offer functions that accept all parameters required to compute a digest value plus an existing digest value to compare against. The newly computed digest value is never returned to the user and is only used for the internal comparison.
- "built-in compare": All libraries offer constant-time comparison functions like `rust_sodium::utils::memcmp()` or `openssl:memcmp::eq()`. `rust-crypto` even makes its constant-time comparison available through the `==` operator by implementing the `PartialEq` trait for its `MacResult` type.
- "manual compare": A few users implement their own constant-time comparison function, copy it from elsewhere or reference the `constant_time_eq` crate.
- "insecure compare": Some users use the `==` operator or `PartialEq.eq()` function on strings or byte slices, both of which are not constant-time.
- "none": The user does not verify digests.

Constant-time comparison functions are best practice because comparisons which do not guarantee a constant runtime could potentially make the application vulnerable to timing attacks (see section 2.2.8). However, timing attacks are not realistic in all scenarios, as the attacker needs many attempts and has to access the system as directly as possible to obtain useful timing data. Whenever a timing attack could not obviously be ruled out, I filed a GitHub issue in the respective repository and some users reacted by resorting to the safer built-in comparison functions. The results presented below do *not* include these corrections. That is, they reflect the situation based on the existing APIs and documentation without my intervention.

As part of the many refactorings in 2016, `rust-openssl`'s old HMAC API was removed[7] and replaced with a new API based on `PKey`, which generalizes signing and verifying with RSA, HMAC and others. Most users reference an older version of the library and still use the old API. Users of both APIs were counted together.

6.4.1 Results

The data types for HMAC keys and input data are largely comparable to the hash input data types (see figure 6.4): most users have a string or a byte slice, whereas vectors and arrays are less popular. The `Read` and `Write` traits are not used at all to feed HMACs. Only 5 users (10.9%) have multiple pieces of input data. Compared to the output types of hash functions, Base64 encoding is much more popular with HMACs (see figure 6.5 left). The reason is that Base64 encoding is often used for client authentication when contacting a server. Only the "vec" category has as many users as Base64 (28.1%), but vectors are usually used for process-internal comparisons rather than internet requests.

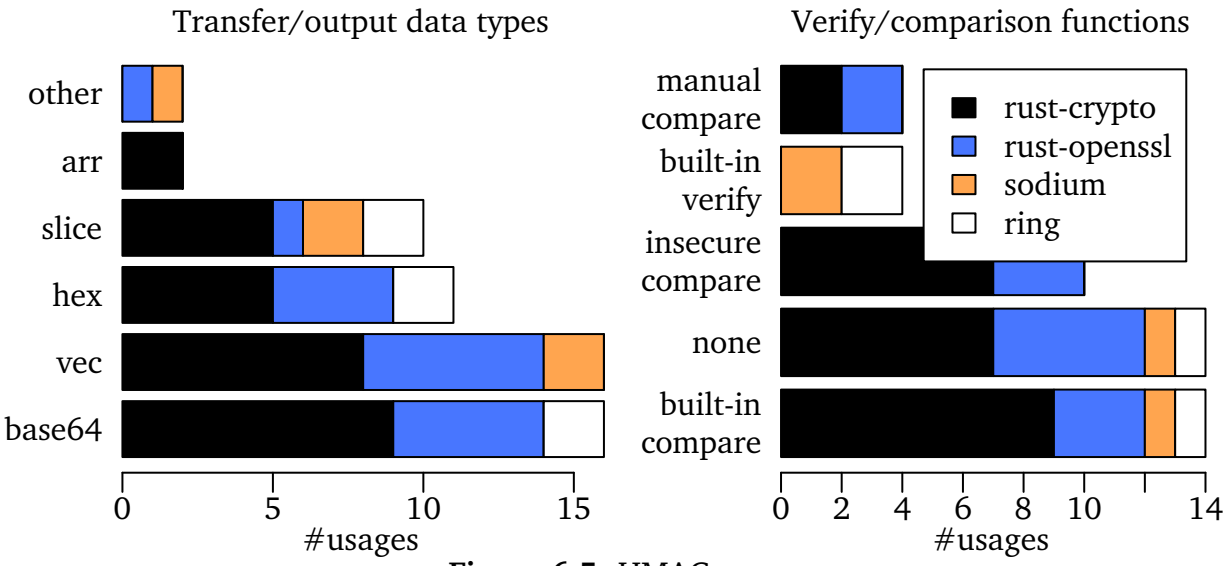

Figure 6.5: HMAC usage

30.4% of all HMAC users do not need verification or digest comparisons at all (see figure 6.5 right). Just as many use the comparison function provided by the library. 21.7% use an insecure comparison function, though this does not necessarily lead to a vulnerability, as mentioned above. Interestingly, none of `ring`'s and `sodium`'s users use an insecure function, as they have all discovered the built-ins designed for secure verification. However, the absolute number of users of these two libraries is too low for this observation to be statistically significant. Please see section 8.9 for a discussion of mitigation strategies against accidentally insecure comparisons.

[7] https://github.com/sfackler/rust-openssl/pull/474 (2017-03-23)

6.5 Symmetric encryption

The 30 users of unauthenticated symmetric encryption and the 17 users of authenticated symmetric encryption are analysed together because it is reasonable to assume that the requirements are similar and most users of unauthenticated encryption should probably be using authenticated encryption.

The approach is analogous to the previous analyses, but there are six points of interest:

- the data type of the *key*,
- the data type of the *initialization vector*, if applicable,
- the *input* and *output* data types (plaintext and ciphertext),
- the chosen *padding* (none or PKCS padding),
- whether *additional data* is specified for AEAD ciphers and, if so, its data type,
- whether a *blockwise/piecewise* API is used instead of encrypting/decrypting all the data at once.

`rust-crypto`'s API for block ciphers in ECB or CBC mode uses its own buffer types for inputs and outputs, and the `encrypt()` function does not always encrypt the whole input. Instead, it either returns `BufferUnderflow` or `BufferOverflow` and the caller has to react accordingly. This makes the API incredibly difficult to use, as the caller would need to understand all these concepts from the sparse documentation and then combine them to form a solution. Fortunately, there is a code sample[8] in the repository that contains a helper function, which essentially lifts the low-level API of the library to a higher-level API that is much more usable. All users except one (88.9%) consequently copy this helper function into their code and some slightly adapt it. These cases were not counted as usages of the piecewise API because that API is only used within the generic helper function that could as well be part of the library itself (i.e. it does not contain application-specific code). For comparison, 46.7% of `rust-openssl` users also create their own helper function, mostly because they use the fine-grained `Crypter` API rather than the one-shot function `encrypt()`. In all those cases, the data types used *outside* the helper functions were analysed.

6.5.1 Results

The input data types (see figure 6.6 left) are still similar to the ones of hashing and HMAC, but (heap-allocated) `String`s are less important (11.4%) and heap-allocated vectors are used instead (30.0%). The output is mostly used as a byte vector (30.0%) and only rarely converted to UTF-8 strings (only works for decryption outputs as ciphertexts are usually not valid UTF-8) or formatted as Base64. Similarly, the key is often present as a slice (30.8%, see figure 6.6 right), in a vector (30.8%) or in an array (19.2%), whereas

[8] https://github.com/DaGenix/rust-crypto/blob/b6e3294/examples/symmetriccipher.rs (2017-03-23)

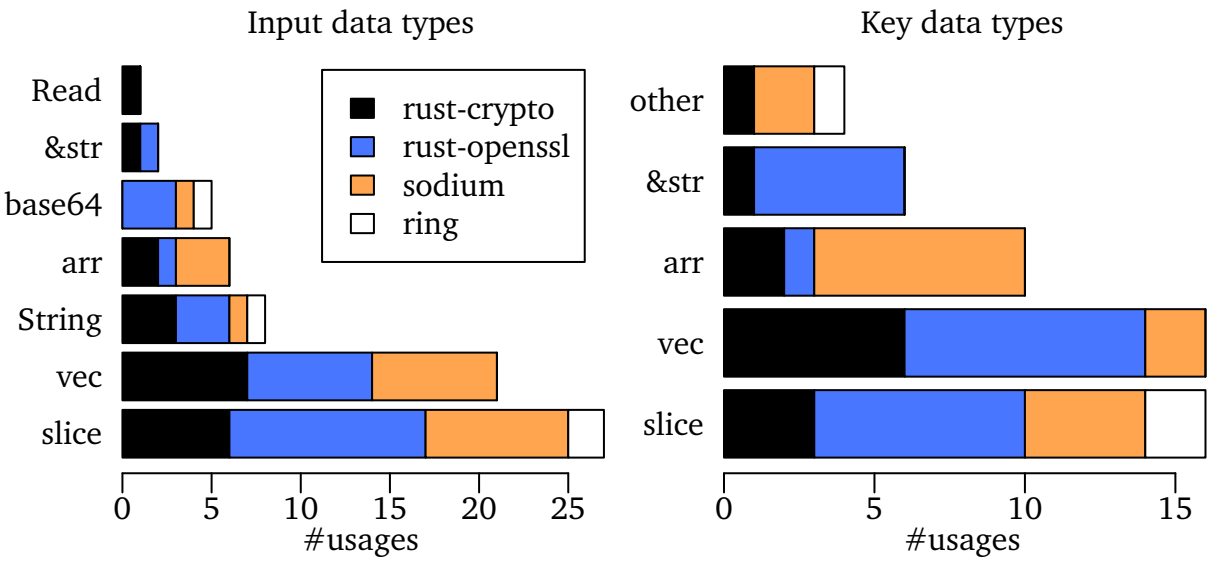

Figure 6.6: Symmetric encryption usage

strings are again rare (11.5%), especially compared to HMAC usages where strings are the second most popular data format for keys (35%). A brief follow-up analysis showed that HMAC keys are often passwords (entered by the user or read from a file) or tokens/keys received from a server in an HTTP request, so that they are naturally strings. Keys for symmetric encryption, on the other hand, need to be of a specific size (whereas HMAC accepts arbitrary-length keys), so that they are either generated randomly, derived from a given password or loaded from a file. All of these operations tend to return heap-allocated vectors of plain bytes rather than human-readable strings.

The data types for the initialization vector (IV) are dominated by arrays (40.7%) because the easiest way to generate a random IV is to allocate an array of the respective (fixed) size and fill it. Furthermore, `sodium`'s widely used `gen_nonce()` function returns an array. During decryption, the received IVs are mostly present as byte slices (25.9%) or vectors (22.2%); strings again play a minor role.

Most users (58.1%) stick with the library's default padding. This option is always secure: `rust-crypto` and `rust-openssl` default to PKCS, `sodium` always uses a custom padding format and `ring` does not offer ciphers that require padding to begin with. 23.3% set the padding to PKCS explicitly (which has no effect) and 18.6% turn it off.

The AEAD APIs of `rust-crypto` and `ring` have a parameter for additional associated data (AAD), and `rust-openssl` added it recently.[9] However, they only account for 4 of the 17 uses of authenticated encryption, and *all* 4 users leave the parameter empty. The remaining 13 crates use `sodium`, which does not have the parameter in the first place.

[9]https://github.com/sfackler/rust-openssl/pull/519 (2017-03-23)

Only 7 users (14.9%) need the blockwise/piecewise API, that is, they need to feed a block cipher with individual blocks or a stream cipher with individual chunks of data. A follow-up analysis of these usages shows that 5 of them encrypt streamed/buffered data that is not available all at once because it is sent over a network connection or similar, and the remaining 2 need to encrypt single blocks with AES to implement the custom key transformation function of KeePass.[10] The latter is an advanced cryptography application and there might be more use cases like this that have been excluded because they expose a cryptographic API themselves (see figure 6.2).

6.6 Threats to validity

There are several threats to the external validity of this analysis. Most importantly, the results should not be applied blindly to design new APIs because of self-reinforcement effects similar to the echo chamber effect in news media [cf. JC09, chapter 5]. The assumption that users have a solid application design, which determines the data types they use and the kind of algorithms they need, is flawed. Especially in smaller projects, the algorithms offered by a crypto library and the data types used by its interfaces can influence the design of the application using the library.

For empirical evidence, consider figure 6.7, which is a different view of the output data types presented in figure 6.4 on page 88. If there were no influences between a project's code structure and the used libraries, all four bars would be similar, with the usual statistical variance (the percentages of `ring` are problematic because is has rather few users in this analysis, but at least `rust-crypto` and `rust-openssl` would be similar).

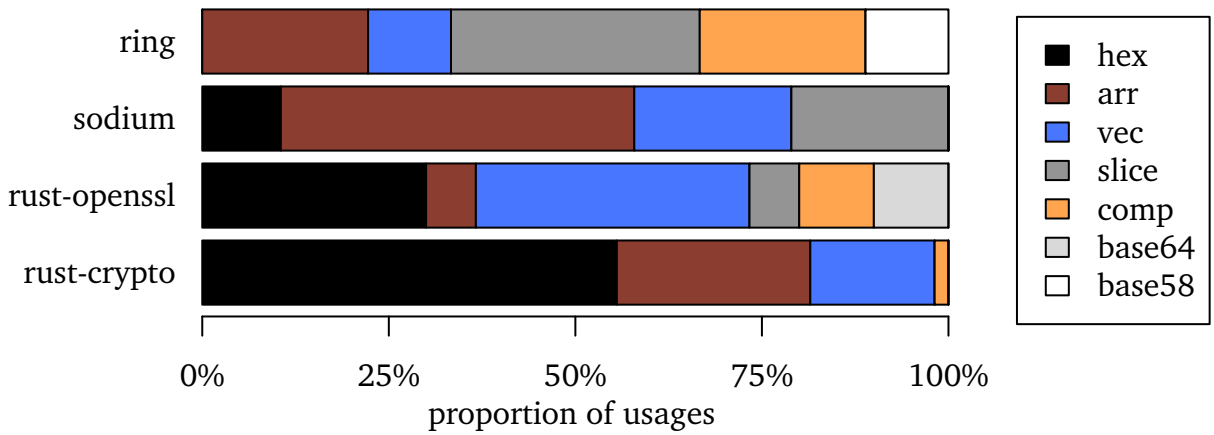

Figure 6.7: Hash function usage: output data types, percentages per library

[10] https://github.com/raymontag/rust-keepass/blob/a15a9b8/src/kpdb/crypter.rs#L313 (2017-03-23)

Nevertheless, more than half of `rust-crypto`'s users work with the hexadecimal format, which `rust-crypto` conveniently provides through its `result_str()` getter, whereas the proportion is much smaller for the other libraries. Similarly, `rust-openssl` has disproportionally many "vec" users, probably because `Vec<u8>` is the return type of its hashing functions, and many `sodium` users work directly with the `[u8]` array embedded in the `Digest` result type without further conversions.

There are two relevant causes that can explain these empirical observations and, unfortunately, it is impossible to tell them apart with the data available on GitHub. On the one hand, a user who only needs a crypto library for a specific purpose (a scenario in which the desired algorithm and data types are already determined) could base their decision for a crypto library on the specific needs at hand and choose the one that fits best. On the other hand, a user who already uses or has chosen a certain library could adapt the surrounding code design to fit the data types of the crypto API. And if the user is not bound to use a specific crypto algorithm because of some protocol or another external interface, they will choose one of the algorithms available in the library. In particular, only very few users will make an informed decision about the best algorithm independently of the library they already have—and then possibly look for another library that implements it. In order to find out whether the API design influenced the application's code design or the application's requirements influenced the library selection, one would have to ask the developers directly because these decisions are generally not documented publicly.

If at least some users adapt their code to fit the current APIs, then using the insights gained from their code to design new APIs would result in an echo chamber: the old APIs would indirectly influence the design of the new API—not by being a good API to learn from, but by giving the impression that this is the kind of API that users need. Hence, the data in this analysis cannot perfectly reflect the pure needs of Rust crypto users: it is biased towards the currently existing APIs. `rust-crypto` and `rust-openssl` have the most users and thus the biggest influence. This bias needs to be kept in mind when working with the data presented in this chapter.

Another threat to external validity is posed by the selection method for dependent crates. Normally, crates published on `crates.io` are reusable libraries, as opposed to end-user applications. As such, they potentially have different (presumably more complex) requirements with respect to a crypto API than the average end-user application would have. On the other hand, end-user applications should rarely use crypto APIs directly in the first place. The second group of dependent crates was selected through a GitHub search ordered by "recently indexed." This provides sufficient randomness with respect to the kinds of users, but it naturally constrains the selection to open-source code published on GitHub. As crates with a cryptographic API of their own were excluded as users from this analysis, the most sophisticated use cases are naturally missing. This

is done on purpose and limits the applicability of these results to the 80% use case as discussed in the introduction.

The main threat to internal validity is the small number of dependent crates in some groups. While the total number is well in the hundreds, `ring` has only 4 HMAC users, for example. To mitigate this threat, no conclusions are drawn in this thesis from statistics based on populations smaller than 10 or the threat is pointed out explicitly.

In addition, library usages and especially the data types before and after conversions are sometimes ambiguous. The available body of open-source Rust code is diverse, every project has its own coding style, the notion of "idiomatic Rust" has changed over the years and thus it is not always possible to find the perfectly fitting category for every usage. This only affects a few special cases though, where I tried to find the most reasonable fit, and it impacts the internal validity only slightly.

6.7 Conclusion

This section presents an analysis of how often certain features of Rust crypto libraries are currently used by code available on GitHub, in order to answer RQ 14. **Hashing, MAC, symmetric encryption and asymmetric encryption are the most frequently used primitives.** Data is mostly passed around in the form of **byte slices, vectors and arrays**, though the primitives also have other particular favourites: hash values are often represented as hexadecimal strings, HMACs are converted to Base64 strings and symmetric encryption is mostly used with plain **`u8`** arrays or vectors.

Interestingly, the used primitives do not quite correspond to the results of Nadi et al. [NKMB16, table 4], who analyse the cryptographic *tasks* of Java projects on GitHub and find symmetric encryption and signatures (which includes MAC) to be the most common, whereas hashing is extremely rare. The difference shows that *tasks* are rather high-level, but the *implementation* often calls low-level primitives. For instance, generating a hash value is rarely the ultimate goal, but it is often done to reach a goal. It would be interesting to analyse the Rust crypto usages with an open-coding technique like the one by Nadi et al. (see section 9.2).

Conclusions have to be drawn with care from this data because they can result in an echo chamber effect. For instance, the AAD parameter of AEAD ciphers is currently not used at all. Based on this data alone it could seemingly be removed, but instead it should be promoted and explained because more users arguably *should* be using it to make their applications more secure. Similarly, library authors should not feel obligated to convert hash values to hexadecimal strings only because most users need that particular format—the conversion can be done with a single function call in the same line of code, which improves the separation of concerns. This data can nevertheless be used to inform design decisions, and it will be used in sections 8.3.2, 8.5 and 8.9.

7 Usability analysis

This chapter investigates one particular aspect of the existing Rust crypto primitive libraries, namely their usability and misuse resistance (RQ 15), through a series of experiments. Section 7.1 reports on a self-experiment I conducted at the very beginning of my work on this thesis and section 7.2 reports on a controlled experiment with Rust beginners. While this chapter only analyses the existing libraries, chapter 8 will make concrete recommendations to improve the usability and misuse resistance of existing and future crypto libraries in Rust.

7.1 Self-experiment

One of my first activities for this thesis was a self-experiment with the major Rust crypto libraries. Before starting this thesis, I had never used Rust before, but I have significant programming experience in Java, .NET, C++ and others as well as basic cryptography knowledge. Right after completing a Rust tutorial and playing around with the language for a week, I started the four-stage self-experiment described in the following. The self-experiment was conducted between 4 and 16 November 2016 with all five libraries introduced in section 5.3 (not counting the forks) and the resulting implementations are available for download (see appendix A).

7.1.1 Protocol

The tasks of the four experiment stages are designed to cover the most important APIs (hashing, HMAC, symmetric encryption, AEAD and key management) and to explore the libraries' structure and documentation. In particular, the cryptographic protocol implemented here is not sensible in itself but only designed to drive the experiment and to get a runnable sample application.

There are a server and a client, which establish a Transmission Control Protocol (TCP) connection and send messages both ways. The technical details of the TCP connection are not relevant to this analysis and the code is the same for all experiments. Inside that code skeleton, I inserted the respective crypto library calls.

To help explain the protocol steps, the `rust_sodium` implementation is also given below (crypto API calls are highlighted in bold print). `rust_sodium` was in fact the last library I experimented with, but it provides a one-line solution to almost all the steps listed below, so it serves well as an illustration.

1. Before the TCP connection is established, the main application **generates a random key** and saves it in a shared key file.

```
let key = secretbox::gen_key();
keyfile.write_all(&key.0).unwrap();
```

2. Both client and server **load the key** from the file, then the client connects to the server.

```
file.read_exact(&mut key_bytes).unwrap();
let key = secretbox::Key::from_slice(&key_bytes).unwrap();
```

3. The client **encrypts a fixed test string** with the key. Depending on the encryption method, this requires **generating a random nonce** and produces a ciphertext and possibly a tag.

```
let nonce = secretbox::gen_nonce();
let ciphertext = secretbox::seal(&text, &nonce, &key);
```

4. The client sends the ciphertext and possibly the nonce and the tag to the server.
5. The server receives the data and **decrypts** the text.

```
let text_bytes = secretbox::open(&ciphertext_bytes, &nonce, &key).unwrap();
```

6. The server **computes a digest** of the text (a hash or HMAC value).

```
let hash_value = hash::hash(&text_bytes).0.to_hex();
```

7. The server sends the digest to the client.
8. The client receives the digest and **verifies** that it is correct (either with a dedicated verification method or by computing the digest as well and comparing it).

```
let received_hash = received_data.from_hex().unwrap();
let received_digest = hash::Digest::from_slice(&received_hash).unwrap();
assert_eq!(received_digest, hash::hash(&text_bytes));
```

7.1.2 Tools and practices

I used the `cargo` command line tool, `rustc` version 1.12.1 and IntelliJ IDEA with the `intellij-rust` plugin, which provides code completion and navigation unless too many Rust macros are involved. While I generally used the library version available at the start of my experiments (4 November 2016), I had to update in some cases after reported build problems other other issues had been resolved. If available, I used the publicly hosted Rustdoc of the respective library, otherwise I read the Rustdoc comments from the source code. The first step for every library was to read the top-level documentation

of the crate and of the relevant module(s). Whenever I found applicable example code, I tried to use it by copying and adapting.

7.1.3 Stages

I. Integrate encryption and hash function from the library into the code skeleton.
   - Use the documentation to find the suitable module and algorithm. Generally, any symmetric encryption method is fine here. Prefer well-known algorithms when a decision needs to be made.
   - Use the documentation to choose a hash algorithm. Prefer newer SHA versions over everything else when a decision needs to be made.

II. Switch to another encryption cipher.
   - If the previous cipher was not AEAD, switch to an authenticated cipher now.
   - Otherwise, just try another cipher.

III. Replace the hash function with HMAC.
   - For convenience, use the same key as for the encryption (even though this is not generally recommended, it simplifies the experiment).
   - If a hash function needs to be specified to use HMAC, prefer newer SHA versions over everything else.

IV. Tamper with the authenticators and observe the effects.
   - Modify a byte of the AEAD tag.
   - Modify a byte of the HMAC digest.
   - Examine how easily the checks could be forgotten or ignored.

During the experiment, I paid particular attention to the following aspects of the libraries:

- Their documentation: How easy to navigate, how detailed and how helpful is it?
- Their structure: How are the algorithms organized into modules? How easy is it to find the right one? Which algorithms use the same interface and are thus easily substitutable?
- The function signatures: Are they at a suitable level of abstraction? How difficult is it to supply the right values to every parameter?
- Their misuse resistance: Does the library warn about deprecated and insecure algorithms? (How) does the library promote or enforce the use of constant-time comparison functions?[1] How easily can important checks be skipped (e.g. by accidentally ignoring a return value)?

[1] At the time of the experiment, I was not aware that the regular comparison functions (==) are insecure for these purposes—like most developers with only basic cryptographic knowledge. This evaluation was therefore done several months later.

7.1.4 rust-crypto (0.2.36)

Documentation: The documentation is *helpful and detailed, if present* (e.g. for hashing), but *many modules have no documentation* at all, including the modules for symmetric encryption and AEAD. Fortunately, the only code sample[2] of the entire library implements AES-CBC encryption, which would be difficult to do without guidance (see below). Every primitive is represented by one or more traits, whose implementations live in other modules but the automatically generated "implementors" sections in the documentation make it easy to find them.

Signatures: The API for AES-CBC cannot process an entire message at once. Instead, it requires the caller to feed the input and output through custom buffer types and react dynamically to buffer overflows and underflows, because "each encryption operation will 'make progress'. 'Making progress' is a bit loosely defined [...]," as the code sample explains.[2] Hence, the API user needs to implement a `loop` that keeps calling the `encrypt()` (or `decrypt()`) function until all data has been processed. Considering the task at hand, the API is extremely *low-level and difficult to understand*, so I just used the higher-level API exposed by the code sample:

```
fn encrypt(data: &[u8], key: &[u8], iv: &[u8]) -> Result<Vec<u8>, ...>
```

The AEAD, hashing and HMAC functions were *relatively straightforward* to use, though they do not have convenience functions like most other libraries. They return authentication tags and digest values through `&mut [u8]` parameters, for which an array or vector has to be preallocated. While the hashing and HMAC APIs inform about the appropriate size in bytes through separate getters, the user has to guess the size of the authentication tag (and adjust until no more error occurs), look it up online or extract it from the `rust-crypto` source code.

API structure: It is difficult to switch between block, stream and authenticated ciphers or between hashing and HMAC because they use entirely *different* APIs. Switching between algorithms is trivial because the library's concept cleverly uses Rust's type system: primitives are traits and algorithms are structs which implement these traits. This concept is rigorously applied, which results in *complex constructions*: the generic CTR mode implementation can be used with any block cipher, the `Hmac<D>` takes a hashing algorithm (digest) `D` as a parameter, and so on. By definition, such an accurate reflection of the implementation cannot hide implementation details from the user, making it more difficult to use and less misuse resistant. There is one module per primitive and one per algorithm, all of which are on the top level, which *clutters* the namespace and the documentation landing page, especially if more algorithms were added in the future.

[2] https://github.com/DaGenix/rust-crypto/blob/b6e3294/examples/symmetriccipher.rs (2017-03-23)

Misuse resistance: The HMAC API returns a custom struct which overrides the `==` operator so that *comparisons are performed in constant time*. However, I still made the mistake to retrieve the underlying value (overlooking the warning on the `.code()` getter) and to compare it insecurely, likely because the same code previously used a plain, unauthenticated hash value—and the hashing API does not encourage constant-time comparisons like the HMAC API does. Regarding the last stage, `rust-crypto` returns **`false`** from the `decrypt()` function if the authentication tag does not match. In case the caller ignores the return value (for which there is no compiler warning), *no harm is done* because `rust-crypto` does not write the decrypted message to the output parameter when the verification fails.

7.1.5 ring (0.5.3)

Documentation: The documentation is *brief but complete*. Most parts of the library have a code sample, though the difficult AEAD API does not (yet).

Signatures: The encryption API is relatively *cumbersome* to use.[3] It works "in-place," that is, it uses the same memory space for input and output, whereby the latter is longer because it also contains the authentication tag. Therefore, extra space (called `out_suffix`) has to be allocated beforehand, but it is not always fully used, so the result needs to be truncated to the appropriate length. The API allows efficient implementations even in heap-less environments, but it *does not fit the use case* in the experiment. The hashing and HMAC APIs were *easy to use*.

API structure: For every primitive, `ring` has a top-level module with *global functions* for the main operations and static instances for the supported algorithms. This makes substituting algorithms *straightforward* and other primitives are also easy to use because all modules have a similar structure.

Misuse resistance: `ring` does not offer unauthenticated encryption at all. Therefore, already during the first stage of my experiment, I was *directed to use the safer* AEAD instead. As discussed in section 6.4, `ring` has a dedicated `verify()` for HMACs, which never exposes the digest to *prevent* accidental misuse. Although the user could still use the `sign()` function during verification, the function names, the symmetrical design of the module and the code sample all *encourage* the use of the safe `verify()` function. Tampering with the tag or digest produces errors in the return values of the decryption or verification functions, and ignoring these return values leads to *compiler warnings*.

[3] There is an issue to improve it: https://github.com/briansmith/ring/issues/371 (2017-03-23)

7.1.6 rust-openssl (0.9.1)

My initial attempts to use AES-GCM in `rust-openssl` version 0.9.0 failed because the wrapper API was not designed for AEAD ciphers, as it turned out.[4] The missing code was added within a few hours and released in version 0.9.1.

Documentation: The documentation contains *helpful code samples* for the most important parts of the library, though the remaining documentation is *rather sparse*. For example, I missed information about nonce and key lengths and about which ciphers need a nonce in the first place. Theoretically, the official OpenSSL documentation could be used in addition to the Rustdoc, but it contains many low-level details that `rust-openssl` users do not need to worry about. The missing information turned out to be returned from other API functions rather than being documented. The documentation *does not guide* towards the right module and does not recommend algorithms or at least warn about dangerous ones. It took me a while to find the HMAC implementation because there is no module named `hmac` or similar. A documentation search led me to the `pkey::PKey::hmac()` constructor, but there was *no hint* towards the `sign` module that contains the necessary main functions and even an extensive code sample for HMAC.

API structure: `rust-openssl` mostly mirrors OpenSSL's structure. Besides the cryptographic primitives, it contains a TLS implementation, though they are *not noticeably separated*. Like in `ring`, there is roughly one module per primitive. Every module has a low-level (usually multi-step) API that is accessible through the main struct of the module, and most modules additionally offer *global convenience functions* that do common, simple operations in one step. This is a best practice, in principle, as it minimizes the amount of documentation the user has to read, the number of mistakes they can make and the effort required to switch to another algorithm.

Signatures: The `symm` module unites all symmetric encryption primitives despite their differences: some modes require an IV, AEAD ciphers accept an AD parameter and they return an authentication tag. The `encrypt()` function declares an `iv: Option<&[u8]>` parameter so that the IV can be omitted if the cipher does not require one. A new `encrypt_aead()` function handles all AEAD ciphers separately, but it accepts instances of the same `Cipher` struct. Given such an instance, the user has to find out manually which function to use and whether to supply an IV. In particular, the compiler *cannot prohibit nonsensical calls* like using ECB mode with a nonce, CBC mode without a nonce, a non-AE cipher with the AE API or vice versa. The former goes unnoticed entirely, a missing nonce leads to a good error message during encryption and the latter two just fail at encryption or decryption with a confusing message. Otherwise, calling the

[4] https://github.com/sfackler/rust-openssl/issues/518 (2017-03-23)

library's functions was *easy* thanks to the global convenience functions and because it returns vectors instead of using `&mut` parameters.

Misuse resistance: The API *prevents* the user from accidentally using the decrypted plaintext when the verification failed, since the return value of type `Result<Vec<u8>, ErrorStack>` does not contain the plaintext at all in this case. However, I unknowingly implemented an insecure comparison with `==` on hexadecimal `Strings` because the API made it easy to convert the HMAC digests into that format. The `Verifier` struct, which could take care of the constant-time comparison, *does not support HMACs*.

7.1.7 rust_sodium (0.1.2)

As `sodiumoxide` failed to build on Windows, I used the `rust_sodium` fork with exactly the same API instead.

Documentation: The top-level documentation *points out the right modules* for the most common use cases, including the `secretbox` module for symmetric encryption. Every module's documentation contains a *code sample* that can be used right away and all relevant items are documented.

Signatures: The `libsodium` bindings were *by far the easiest to use* because the primitives are rather high-level. `rust_sodium` improves on the underlying library with *strong types* (e.g. `Key` and `Nonce`), generator functions for these types in the same module and return values instead of `&mut` parameters. The stricter type safety exposed the issue in my experiment design, which uses the same key for encryption and HMAC, and forced me to create two instances from the same key material.

API structure: Switching to other ciphers for stage two would have been easy if there had been another AEAD implementation, but XSalsa20-Poly1305 is currently the only algorithm exposed in the Rust bindings. Instead, I tried to use a different hash algorithm, which is as *easy* as replacing `rust_sodium::hash` with `rust_sodium::hash::sha256` (or any other submodule). This is because every primitive has its own module and all submodules have exactly the same structure generated from the same macro. While the macros do not impact the readability of the code, they make the documentation *more difficult to navigate* and *confuse* the `intellij-rust` plugin, so that code completion and navigation were not available in the experiment.

Misuse resistance: Besides the type-safe function signatures, `rust_sodium` *prevents many misuses* through an opinionated selection of algorithms (inherited from libsodium) that rigorously excludes insecure algorithms and dangerous primitives like unauthenticated encryption. According to the documentation, even hashing and HMAC should not be used "unless you know what you're doing." Like in `ring`, a constant-time `verify()`

function is placed right next to the HMAC generation function and its use is demonstrated by the code sample, so *misuse is unlikely*. And like in `rust-openssl`, the decryption function only returns a plaintext if the verification was successful.

7.1.8 octavo (0.1.1)

Completeness: Because ChaCha20 is currently the only symmetric cipher, I could not use AES in the first stage and had to skip the entire second stage. There is *no authenticated encryption* and no constant-time comparison function.

Documentation: A few parts of the library have *extensive* documentation (though sometimes irrelevant to average users, e.g., an explanation of Kerckhoff's Principle and a formal definition of a cryptosystem) and the hashing module has a helpful code sample. Yet, most parts relevant to the experiment were *practically undocumented*.

Signatures: Unlike other libraries, `octavo` does *not offer convenience functions* but always requires separate steps to instantiate a cipher, pass in the inputs and retrieve the outputs. The otherwise easy-to-use API also uses a `&mut` parameter that *needs to be preallocated*.

API structure: The different primitives live in *separate crates* which can be built individually to reduce compile times and binary sizes. Although there is a main crate that re-exports them and allows to use them almost like a single crate, the design makes the code and documentation *more difficult to navigate*. Like in `rust-crypto`, the HMAC API takes a hash algorithm as a generic parameter, making it *inconvenient* to use:

```
let mut digest = hmac::Hmac::<sha2::Sha256>::new(&key);
let mut hash_value = vec![0; hmac::Hmac::<sha2::Sha256>::output_bytes()];
```

Misuse resistance: As there is *no constant-time comparison function*, I ended up with an insecure == comparison. The documentation *visibly warns* about insecure algorithms.

7.1.9 General observations

As mentioned above, I found `&mut [u8]` parameters difficult to use in general. While `ring`'s use of the parameter was actually complicated (requiring extending and truncating of a vector), most other `&mut` parameters simply serve to return a value from a function whose length is known beforehand. I had trouble allocating the vectors for these parameters and many other Rust users do, too: there are solutions that require two lines of code and an `unsafe` block[5] (still the accepted answer) or that create an infinite iterator and collect it (I used copied this solution from `rust-crypto`'s internal code):

[5] http://stackoverflow.com/a/28209155 (2017-03-23)

```
use std::iter::repeat;
let mut buf: Vec<u8> = repeat(0).take(length).collect();
```

Since March 2015, there is a much easier but surprisingly unknown solution: `vec![0; length]`. There is also a `resize()` method to extend or truncate an existing vector, but even `ring`'s more recent unit test[6] instead uses a loop to push the right number of zeros to the end.

This was one of many occasions when I turned to a library's own source code (particularly the unit tests) for help after the documentation was unhelpful. It is reasonable to assume that other library users would do the same, especially when no code samples are provided. While I rarely had trouble finding a part of the API that addressed my needs, more links between documentation elements or more code samples would have helped to put the pieces together. In addition, I could not find the documentation at all for `rust-crypto` and `octavo` because it was not linked from the repository pages or from crates.io. It was only after the experiment that I found out about docs.rs, which hosts all documentation for all versions of all published crates.

Note that I only experimented with the major libraries for cryptographic primitives, so this experiment is *not* representative of all Rust crypto libraries. Quite contrary, there is a bias towards further developed and polished libraries, which appropriately reflects the experience of the average crypto user in Rust. The other major group of libraries are TLS libraries, which were not covered here.

7.2 Controlled experiment

In joint work with Kai Mindermann (my supervisor), the self-experiment was extended to a controlled experiment with a group of 29 Rust programmers. A detailed report paper has been submitted for review to the Thirteenth Symposium on Usable Privacy and Security (SOUPS '17). This section only briefly explains the experiment setup and summarizes the usability-relevant results.

7.2.1 Setup

In the winter semester 2016/17, the University of Osnabrück offered a semester-long lecture on the Rust programming language. Most of the students in the course participated in the experiment on 7 and 8 February 2017.

[6] https://github.com/briansmith/ring/blob/503ac19/src/aead/aead.rs#L359 (2017-03-23)

The experiment took place in an on-campus computer lab, where participants were given an instruction sheet that briefly introduced symmetric cryptography and explained the task, as well as a running Ubuntu virtual machine (VM) with the Sublime Text editor, the latest Rust compiler and a browser to freely surf the web without limitations. Upon completion of the task or after approx. 50 minutes, participants were asked to fill in a LimeSurvey questionnaire regarding their background, their experience with the library and with the task itself. In addition, the VM's screen was recorded and the resulting code was obtained from the VM.

As the time in the experiment had to be limited to about an hour, the task was much simpler than the self-experiment. Participants were given an application frame that receives a "business text" from a submodule and were supposed to encrypt and decrypt this text within the same `main()` method. The students were divided into two groups: one worked with `rust-crypto` (version 0.2.36) and one worked with `ring` (version 0.6.3). From both groups, a few results had to be excluded because the participants had misunderstood the task and implemented their own encryption function without using the respective library at all, hence their success did not depend on the library or its usability at all. We used several metrics computed from the resulting code and the video recordings to evaluate and compare the usability of the libraries, including effectiveness, efficiency, satisfaction and lostness.

7.2.2 Summary of results

Given my experience in the self-experiment, the result of the controlled experiment is quite counter-intuitive: **none of the `ring` users finished the task but 4 out of 11 `rust-crypto` did.** We demonstrate that they only succeeded because they found and used the code sample from the `rust-crypto` repository, which fitted the use case and required only little more code to be written. The **`rust-crypto` users who missed or ignored the code sample were all unsuccessful**, like all `ring` users, where no sample existed at all. These remaining participants mostly could not finish because of the time limit but made progress in the right direction nonetheless. **Consistent with this observation, many successful participants name the code sample as crucial to their success, and many `ring` users name the lack of a code sample as the major obstacle.**

In addition, the `rust-crypto` users were significantly more experienced Rust programmers according to their self-rating, even though the groups were randomly assigned. The results clearly show that more experienced Rust programmers across both groups were more successful, so there is a natural tendency towards higher success in the `rust-crypto` group which is not due to the library's usability. Therefore, **the *hard* data from the experiment cannot fairly compare the two libraries**, but the experiment

still provides many valuable insights regarding the overall usability of Rust crypto libraries and the usability of particular technical constructs. The major technical issues regarding the libraries' documentation and APIs are briefly summarized below, combining the video observations with the participants' answers in the questionnaire. Note that most of these issues have also arisen in the self-experiment (see the previous section) and are explained there in more detail.

None of the participants researched the security implications of the chosen algorithms. The `rust-crypto` users who found the code sample stuck to the suggested AES-CBC encryption with random nonce. Even after completing the task, they did not (re)consider this decision. Some `ring` users did not identify the `ring::aead` module as the encryption module, but most did and used the AES-GCM algorithm there. In the questionnaire, most participants indicated that they were rather unsure about the security of their code and it remains unclear how much time they would have invested researching the security aspects if they had been given more time.

Several parts of the API were difficult to deal with for multiple users. Mostly `ring` users were **unsure about the `nonce` and `ad` parameters** (`rust-crypto`'s code sample demonstrates how to generate a random nonce and there is no `ad` parameter in unauthenticated encryption). Although the majority figured out the correct solution (random `nonce` and empty `ad`) after some research, a few ended up passing zeros, the plaintext or the secret key into these parameters instead. Secondly, all `ring` users had **difficulties using the `in_out` parameter**, dealing with the `out_suffix` and truncating the result correctly.

While the majority of `rust-crypto` users criticized that its **documentation was mostly unhelpful** and difficult to find (even though a URL was provided on the instruction sheet), the `ring` documentation was **harder to navigate and not detailed enough**: From the top-level documentation page, many did not decide for `aead` as the right module and a few participants copy-pasted the code samples from the unrelated `hmac` or `pbkdf2` modules. Those who identified the `aead` module and the `seal_in_place()` function inside tried to instantiate an `Algorithm` to pass to the `SealingKey` because they had not found the static instances in the same module. Those who did not initially choose the `aead` module later found the `ring::aead::AES_GCM_256` instance through a search on the web or in the documentation but then struggled to find the `seal_in_place()` function to use it with. Besides the libraries' and Rust's documentation, **participants often used Wikipedia and Stack Overflow** as information sources. None of these sites currently offers Rust-crypto-specific advice.

7.3 Conclusion

To put the experimental findings described in this chapter into perspective, this section relates them to the literature. Please refer to section 3.3.2 for an overview of the existing research on the usability and misuse resistance of cryptographic APIs. I systematically reviewed all these sources for:

- criteria that they use to measure or otherwise judge the quality of crypto APIs,
- best practices they derive from experience, experiments and analyses,
- ideas, suggestions and pleas on what to improve in future crypto APIs, and
- design problems they discovered in existing crypto APIs.

In particular, I considered the "Common Rules in Cryptography" and the "Mitigations" by Egele et al. [EBFK13, sec. 3 and 7], the "Vulnerabilities" section by Lazar et al. [LCWZ14, sec. 2], all seven "Points of Interest" by Das and King [DK14, sec. 5], the "Ten Principles for Creating Usable and Secure Crypto APIs" by Green and Smith [GS16] and the seven "Crypto Interface Pitfalls" by Devlin [Dev14]. For each of these, I analysed how they apply to the existing Rust crypto APIs and summarized the findings in table 7.1.

Overall, **the libraries which focus on usability—namely `sodiumoxide/rust_sodium` and `ring`—do almost everything right**, though `ring`'s AEAD API is currently difficult to use. Both libraries are **highly misuse resistant** already. **The other libraries consciously do not prioritize these goals and consequently exhibit some of the design weaknesses described in the literature and especially a lack of documentation.** Despite its unintuitive structure, `rust-openssl` is more usable than the lower-level, pure-Rust library `rust-crypto`, though the only **code sample** of the latter helped many experiment participants succeed. `octavo` cannot be evaluated properly because many relevant features are still missing.

Best practice / suggestion	rust-crypto	rust-openssl	sodium	ring	octavo
Extensive, helpful documentation [DK14; EBFK13]	✗	✔✗	✔✔	✔	✔✗
High-level interfaces [Dev14; LCWZ14; NKMB16]	✗	✔✗	✔✔	✔	✗
Exclude deprecated algorithms / warning [DK14]	✗✗	✔✗	✔✔	✔	✔
Offer and advertise AE [Dev14; EBFK13]	✔✗	✔✗	✔✔	✔✔	✗✗
Offer and advertise CSPRNG [Dev14; LCWZ14]	✗✗	✔✗	✔✔	✔✔	✗✗
Safe defaults [Dev14; DK14; EBFK13; GS16, . . .]	✔	✔	✔	✔	✔
Don't leak unauthenticated plaintext [FLW12]	✔	✔	✔	✔	✔
Prevent nonce reuse [Dev14; DK14; FLW12]	✗	✗	✗	✗	✗

Table 7.1: Major libraries' compliance with recommendations from the literature

8 Improving usability and misuse resistance

When designing or improving an API, it is natural to look at other existing APIs and to learn from their innovations, best practices as well as their mistakes. The previous chapter experimentally evaluated the usability of the existing Rust crypto APIs for primitives and related the results to the literature on crypto API usability. In this chapter, insights from all these sources are bundled into a single discussion, which is split into topics from documentation in section 8.1 to `&mut` parameters in section 8.10, ordered by their level of technical detail. Only topics that affect usability or misuse resistance are discussed, but every discussion naturally considers other goals such as performance, maintainability or the implementation effort because every API design is ultimately a compromise.

Several topics are omitted because they are not specific to crypto APIs. However, a few of them should be pointed out because they are particularly important for crypto APIs:

- *Error handling conventions*: See section 2.1.4 for a description of how error handling works in Rust and the RFC 236 [TC14] for the important distinction between contract violations and obstructions. Only for the latter, the `Result<T, E>` type should be used. Unlike simple numeric return values, which are often used in C programming and have led to several vulnerabilities [GIJ+12, sec. 4.1], the `Result` type generates a compiler warning when a return value is ignored. It should therefore be used for all security-relevant return values.
- *Data conversions*: As the analysis in section 6.3.1 showed, there is a disconnect between the data types that hash functions (and others) accept/provide and the types that users have/need. There are multiple approaches to making conversions more "convenient and idiomatic" [Mar16], but they arguably also make them more subtle and the function signatures harder to understand. In cryptography, most conversions involve a `String` format: UTF-8, hexadecimal, Base64 or Base58. Because all of these have the type `String`, there is not much that a library can do to simplify the process, as the user needs to explicitly specify the format anyway. To maintain a proper separation of concerns, crypto libraries should just work with sequences of bytes (slices, arrays and vectors of `u8`) and *leave all conversions*

to other libraries, as this does not entail any significant extra effort for the caller. Based on the data presented in chapter 6, API designers should ensure that the 80% use case can be implemented with little code and the documentation could refer to appropriate third-party crates/functions for the most commonly needed conversions.
- *Naming conventions*: Please refer to section 3.2.1 for general styleguides, which also cover naming conventions.

All code snippets presented in this chapter are available for download in a runnable sample application (see appendix A).

8.1 Documentation

The top two of Nichols' "six easy ways to make your crate awesome" concern documentation and example code [Nic16]. Good documentation improves an API's learnability and, in the case of crypto APIs, it may teach some cryptography basics, as well. It is essential for misuse resistance, too, because users will consult less reliable sources when the documentation is missing or incomplete. For example, Acar et al. [ABF+16] find that programmers who use Stack Overflow produce significantly less secure code and call for better documentation with "secure and functional code examples." This section discusses how the documentation should be structured and what elements should be included. As Das and King [DK14, sec. 6] point out, "developers may still accidentally skip over" the documentation and code samples. Therefore, the misuse resistance of an API can never rely solely on its documentation.

8.1.1 Structure and navigation

A library's documentation should obviously be as complete as possible. However, this not only includes explanations for every struct, trait and function, the structure of which is given by the code structure, but also "landing pages" [DK14, sec. 5.5] on the crate and module levels. Such landing pages should point developers to the right (sub)modules and functions for their use case, especially if the library offers (too) many cipher implementations [DK14, sec. 5.3].

During my self-experiment, I was often lost for multiple minutes trying to find the right module. The symmetric (authenticated) encryption modules in different libraries, for example, go by the names `symmetriccipher`, `aes`, `symm`, `aead`, `secretbox` and `crypto::stream` (no name occurs twice). Similarly, HMAC can be found under `hmac`, `sign` and `pkey`, `mac` or `auth`. This shows that module names alone are not enough. The

landing page should mention primitive and algorithm names and point to the right modules. Currently, `sodiumoxide` is the only library with a true landing page.

> **DO** include landing pages with introductions, explanations and pointers at the crate and module level.

Aside from this top-down navigation, the existing libraries could also benefit from more pointers between related items in different modules. Two current examples are the `openssl::pkey::PKey::hmac()` factory, whose result needs to be used with functions from other modules (see section 7.1.6), and `ring::aead::seal_in_place()`, which takes a `nonce:&[u8]` parameter whose length has to be retrieved elsewhere (see section 7.2.2).

8.1.2 Recommendations and education

As part of the top-down navigation, the documentation should also recommend which of multiple modules/algorithms to choose, at least for the most common use cases. Although most developers probably have an idea of what encryption and authentication are, very few of them understand why certain ciphers or hash functions are secure while others are not, why authenticated encryption should be used over unauthenticated encryption, and so on. As Das and King [DK14, sec. 5.5] put it, "developers are likely to take the path of least resistance and use the first primitive they come across that seems to solve their problem." Hence, the documentation should warn users about using widely known, yet insecure algorithms like MD5 and RC4 (or the library should omit them in the first place, see section 8.2.2) and recommend suitable alternatives.

At a lower level, the documentation also needs to educate about the consequences of certain parameter values if they can impact security. For instance, `octavo` leaves the choice of key length to the user but includes tables in its documentation[1] to explain the levels of security that each key length provides. Georgiev et al. [GIJ+12, sec. 11.2] propose that the documentation of such parameters should not only state what they do (e.g. "turn on/off hostname verification") but also explicitly state the possible implications (e.g. "if turned off, anyone can impersonate the server"). There are certainly cases in which these implications are irrelevant or impossible (which is why the parameter exists in the first place), but the developer can always make an informed decision beyond the information contained in the documentation and ignore the warning. That is, the documentation does not need to discuss all eventualities, a brief but visible warning is sufficient.

> **DO** warn users about insecure algorithms and dangerous parameter values.

[1] https://docs.rs/octavo-crypto/0.1.1/octavo_crypto/ (2017-04-01)

8.1.3 Code samples

A code sample serves many purposes at once that the documentation would otherwise have to fulfil individually: point to the right API to use, recommend sensible choices for algorithms and key lengths, explain the order in which the functions must be called and illustrate how the returned result can be used. The controlled experiment showed that a code sample can crucially improve the usability of a library. Therefore, all important APIs should be accompanied with code samples for the most common use cases (Hertleif even recommends a small code sample for "everything" [Her16b]).

Code samples can either be placed in the documentation or in separate files (in an `examples` directory). In some programming languages, the latter might be the preferred location because the code will then be compiled against the actual API, which prevents it from getting outdated. In Rust, it is arguably better to keep code samples in the documentation. Firstly, `rustdoc` compiles included Rust code and, secondly, it makes the code samples easier to find. In my experience from the self-experiment, code samples are best kept in the module-level documentation, as this might be the last level of documentation that the user navigates to before they start coding and using the IDE's code completion instead. An excellent example are `ring`'s code samples for the `hmac` module,[2] whereas I overlooked the documentation for the `digest()` function entirely.[3]

Whenever a code sample becomes so long that it seemingly needs a separate file,[4] this is a strong indication that the API is not high-level enough and needs to be adjusted (see section 8.3.2). A code sample for an appropriate API should only be a few lines long.

> **DO** include code samples in the module-level documentation.

Many authors demand that example code needs to be of high quality—it needs to be copy-pasteable without causing trouble for the developer or introducing security bugs. Das and King [DK14, sec. 5.6] point out that code samples must not use "unsafe practices," that is, they should use secure ciphers, secure parameter values (like key lengths), no hard-coded keys, and so on.

During my self-experiment, I discovered that unit tests can easily be abused as code samples, especially when actual code samples are not anywhere to be found. In Rust, it is common practice to place unit tests in a submodule called `test` in the same file. The code is not only available online through GitHub, but it is also included in the documentation through the "src" link. For instance, there is no example code for `rust-openssl`'s `encrypt_aead()` function, where the user can specify the desired tag length by allocating

[2] https://docs.rs/ring/0.5.3/ring/hmac/ (2017-03-23)

[3] https://docs.rs/ring/0.5.3/ring/digest/fn.digest.html (2017-03-23)

[4] Example: https://github.com/DaGenix/rust-crypto/blob/b6e3294/examples/symmetriccipher.rs (2017-03-23)

an accordingly sized array or vector for the respective output parameter. Uninformed users could turn to the unit tests to find out how to allocate that vector and copy the dangerously small 4-byte array from there.[5] (In the meantime, warnings in the unit test and explanations in the function documentation have been added.)

> **DON'T** use insecure primitives, parameter values or hard-coded keys/IVs in code samples. If they are needed in unit tests, place visible warnings around the test code.

8.2 Scope of included algorithms

A crypto library offers a certain selection of algorithms for a number of crypto primitives (a few symmetric ciphers, a couple of hash algorithms, and so on). The set of algorithms and their organization has significant impact on the usefulness, usability and misuse resistance of the library. This section discusses which algorithms to include and which to leave out, section 8.3 discusses levels of abstraction of user-facing APIs, and section 8.4 discusses how to organize the primitives and algorithms.

8.2.1 Completeness

Omitting certain classes of algorithms entirely can pose a risk. Firstly, a secure random number generator (RNG) and a constant-time comparison function should be included (that is, implemented or referenced), as every user needs to generate random keys and nonces and needs to compare hash values and authenticators. If users have to find their own RNG, they could unknowingly choose an insecure one (see section 8.8). And if there is no constant-time comparison function, they could accidentally use Rust's built-in comparison operators (see section 8.9).

Secondly, the most widespread and the most recommendable algorithms should be included. The former are needed for compatibility with existing systems and the latter are hopefully used for new projects. If the library behind an excellent API does not support an important algorithm, several users are forced to turn to another library or even a plain implementation of that algorithm, which is likely less safe and less usable.

> **DON'T** leave out important algorithms or functions like RNG and constant-time comparisons.

[5] https://github.com/sfackler/rust-openssl/blob/9137239/openssl/src/symm.rs#L653 (2017-03-23)

8.2.2 Insecure algorithms

A crypto library could ban all broken algorithms entirely. This is, for instance, the policy of `sodium` and `ring`. Most old and insecure algorithms will not find their way into a modern crypto library simply because nobody needs them and they would have to be maintained and compiled for no benefit at all.

There are, however, a few legitimate use cases for certain insecure algorithms. Especially if they are widespread like MD5 or SHA-1, excluding them is in conflict with the completeness principle in the previous section. Whether these algorithms should be part of a modern crypto library is highly controversial.[6] It would need to be analysed whether these algorithms are used for *cryptographic* purposes, where their security matters, or otherwise. In the latter case, they should live in a separate library that has nothing to do with cryptography and explicitly states that.

If a library includes such deprecated or insecure algorithms, it must be made exceedingly clear to the user that they are deprecated/insecure. There is an ongoing discussion about the various ways to inform the user in the `rust-crypto` repository[6] because it currently has no warnings at all. As a positive (yet not perfect) example, `ring` marks SHA-1 as "deprecated" in the documentation. It is important to place the warnings in the right position(s): when `rust-openssl`'s `hash()` function is used with MD5 (`hash::hash(hash::MessageDigest::md5(), ...)`), the user does not come across the warning on the `hash::Hasher` trait.[7] Similarly, the `RustCrypto` fork has a nice visual representation for the security of hash algorithms[8] (where SHA-1 was marked insecure only hours after the "shattered" attack was published [SBK+17]), but users might not see these warnings when they discover the individual crates like `md4` on crates.io, where there are no further warnings. In addition to deprecation warnings, the documentation should also guide users to choose the right algorithms for new projects (see section 8.1.2).

Even if a warning is well visible in the documentation, some users might not see it because they navigate the library with their IDE's code completion function. For example, a developer might type "`hash::MessageDigest::`" and then choose an insecure algorithm from the popup list. To protect these users, too, these algorithms can be moved to a separate crate or module, which has "deprecated" or "insecure" as (part of) its name. Section 8.4 discusses appropriate locations for insecure/deprecated algorithms with respect to different library structures.

[6] https://github.com/DaGenix/rust-crypto/issues/365 (2017-03-23)
[7] https://docs.rs/openssl/0.9.0/openssl/hash/struct.Hasher.html#warning (2017-03-23)
[8] https://github.com/rustcrypto/hashes/#supported-algorithms (2017-03-16)

A combination of both solutions (documentation and separate module/crate) provides the best misuse protection and does not hurt the informed user either who needs to use deprecated ciphers. When a cipher's status changes (i.e. it becomes deprecated), it is moved to the respective module or crate, which is a breaking change and requires bumping the major version (see section 2.1.8). The user has to decide if using or supporting the algorithm is still necessary and, if so, actively has to point the code to the new coordinates (and possibly add a dependency). There are various approaches to ensure that algorithms from different modules or crates are substitutable so that no further changes are required (see section 8.4 for details).

Consider Crypto++ as a perfectly positive example: it even requires a preprocessor directive to enable a flag called `CRYPTOPP_ENABLE_NAMESPACE_WEAK` before algorithms like MD5 can be used, in addition to the namespace being called `CryptoPP::Weak::` and the explicit warning in the documentation.[9]

> **DO** include warnings about deprecated and insecure algorithms and move them to a module with an off-putting name.

8.3 Level of abstraction

Many researchers have found low-level cryptographic APIs to be a cause of misuse. For example, Duong and Rizzo characterize OpenSSL as "powerful" but criticize that it "requires its users to know how to use cryptographic primitives securely" [DR11, sec. VII]. Surveys conducted by Nadi et al. [NKMB16] find "that the APIs are generally perceived to be too low-level." On the other hand, the higher-level APIs of NaCl, libsodium and Keyczar are often named as positive examples because they "hide details like encryption algorithms, block cipher modes, and key lengths from programmers" [LCWZ14, sec. 3.1].

There are two aspects that make up an API's level of abstraction: the exposed primitive and the API's granularity, as explained in the following two sections. Note that *high-level* and *low-level* are relative terms. For instance, OpenSSL's wiki states that its "EVP functions provide a high level interface"[10] because they abstract over multiple implementations. Compared to the APIs discussed here, this OpenSSL API is very fine-grained and low-level, though. Protocols like TLS, on the other hand, are clearly on a higher level than any primitive. In the following, only APIs that offer access to cryptographic primitives are considered and the terms *high-level* and *low-level* are used in relation to this spectrum.

[9] https://www.cryptopp.com/wiki/Hash_Functions#The_MD5_algorithm (2017-03-14)
[10] https://wiki.openssl.org/index.php/EVP

8.3.1 High-level and low-level primitives

Low-level primitives include encryption, authentication and hashing, whereas high-level primitives like AEAD or password hashing combine or augment other primitives (see section 2.2). Libraries should definitely include and promote these two particular high-level primitives because the corresponding low-level primitives (unauthenticated encryption and regular hash functions for passwords) are deemed insecure [Dev14; DK14; DR11]. Other high-level primitives can also be helpful and are worth including if they solve a common use case.

Similar to the previously discussed algorithms like MD5, which are insecure but popular, there are even more legitimate use cases for low-level primitives. After all, they are the "building blocks" of cryptosystems and higher-level cryptographic protocols—both of which are developed by cryptography experts, though. Instead of trying to strike a balance between low-level and high-level APIs that cater to average developers and experts, it is recommended to offer two API layers at different levels of abstraction.[11]

The lower layer offers individual algorithms for low-level primitives, fine-grained controls and is optimized for performance. It should nevertheless be designed with usability and misuse resistance in mind. The higher layer builds on the low-level primitives, but its API is independent from the underlying algorithms and geared towards the user's task at hand (e.g., storing/verifying a password or sending data securely). Nadi et al. [NKMB16] coined the term *task-based* for such designs. Good examples of task-based APIs are PHP's `password_hash()` function and the entire "recipes layer" of the `cryptography` Python library [KG+]. The "recipes layer" is built on low-level APIs in the so-called "hazardous materials" layer. The latter's deliberately off-putting name does not only describe the layer, but it also gives its name to the `hazmat` module, which contains all the low-level interfaces and needs to be written out in every source code file using it, similar to the `deprecated` and `insecure` modules suggested in section 8.2.2.

To my knowledge, there is currently no Rust crypto library that splits its interface into two layers as consequently as the `cryptography` Python library. `sodium` makes a distinction in its documentation and puts hash functions and others in a "low-level functions" section, which is not to be used "unless you know what you're doing," but the separation is not reflected in the module names. `ring` goes a step further and only exposes its "recipes" layer, consciously leaving users with special needs no choice but to use another library, in order to maximize misuse resistance.[12]

The "recipes and hazmat" structure is certainly worth adopting in Rust libraries. Firstly, the recipes layer accomplishes a central API design goal: "simplify the most common use

[11] https://cryptocoding.net/index.php?title=Coding_rules&oldid=195 (http://archive.is/vkai6)
[12] https://github.com/briansmith/ring/issues/414#issuecomment-275571259 (2017-03-23)

case" [DK14, sec. 6]. Secondly, the naming and placement of the hazmat layer protects uninformed users from accidentally using the wrong primitives. And last but not least, it adheres to the completeness principle, i.e., the overall API is powerful enough even for expert users.

> **DO** split the crypto interface (API) into a high-level "recipes" and a low-level "hazardous materials" layer.

8.3.2 API granularity

Most parameters, except the one for the actual input data, can potentially be eliminated from the API and be replaced with a fixed value, based on a hopefully informed and possibly opinionated decision. Keys and nonces can be generated and returned by the library implementation, obviating the need for the respective parameters. Otherwise, high-level APIs should hide all rarely needed parameters whereas low-level APIs should offer all relevant parameters without setting defaults for them (see section 8.6).

Another kind of granularity results from primitives which accept multiple inputs or primitives where it is convenient to provide the input in multiple pieces. For example, this is the case for all the primitives investigated in chapter 6: hash and HMAC can be fed with individual chunks and produce a single digest in the end, and symmetric block ciphers can encrypt one block at a time. To allow for multiple inputs, the library has to keep the state of the cipher across function calls, so there must be some state object (usually called `Context` or `Cipher`) that accepts multiple input calls and one final output or optional cleanup call. Most users (hash: 90.2%, HMAC: 89.1%, block ciphers: 85.1%) only have a single input, though. As this is clearly the 80% use case, it should be supported with a "one-shot" API that only takes a single piece of input data and saves the user the effort of instantiating a `Context` object. Section 8.4 discusses appropriate locations for the one-shot API and the regular multi-step API with respect to different library structures.

During my self-experiment, I experienced the lack of such a "one-shot" API a couple of times. While most multi-step APIs only require around three function calls, `rust-crypto`'s low-level block cipher API becomes much easier to use with the 14-line helper function from the code sample. Most users (all successful participants in the controlled experiment and 88.9% of users on GitHub, see section 6.5) simply copy this code sample and use the "one-shot" API provided there.

> **DO** include a "one-shot" API for the most common use case.

8.4 Organization of included algorithms

Once the right levels of abstraction for primitives, the desired API granularity and the algorithms to be included are determined (as discussed in the previous sections), they need to be organized so that users can find the right API for their use case. More specifically, structural elements of the Rust language like modules, structs and methods are used, along with their documentation, to help users *discover* the library, i.e.:

- identify the right primitive to use,
- identify the right implementing algorithm to use and
- steer clear of dangerously low-level primitives (see section 8.3.1) and insecure algorithms (see section 8.2.2).

Algorithms are always grouped by the primitive they implement in some way, so that all grouped algorithms have a common API, making them *substitutable*. This allows the user to switch between algorithms easily (and possibly even dynamically at runtime). Some organization strategies allow the library designer to choose a *default* algorithm, which is used when the user does not explicitly specify another.

On a more technical level, there are often slight differences between similar algorithms. For example, different hash algorithms have different output lengths, different encryption ciphers have different key lengths, some block cipher modes require an initialization vector and others do not. Although they can still be called through the same function, the API contract is slightly different. A good API does not only let the user choose an algorithm but also exposes the necessary *meta information* to deal with these differences. If the user needs to preallocate a buffer for a return value or needs to generate a key, the API should provide information about its required length, for example.

Finally, a minor requirement is the ability to *parameterize algorithms* with other algorithms for a different primitive. For example, an HMAC needs a hash function internally. One approach is to list all combinations as separate algorithms: `HMAC-SHA1`, `HMAC-SHA256`, and so on. Alternatively, a library can provide a single, generic `HMAC` implementation that takes the underlying hash algorithm as a parameter.

There are multiple ways to implement *discoverability*, *substitutability*, *default algorithms*, *meta information sources* and *parameterization*, which are discussed in the following sections and summarized in table 8.1 on page 126.

8.4.1 One instance per algorithm (instance-based)

These approaches use one module per primitive (called `primitive1` below; examples are `hash`, `mac` and `aead`) and a single main type (called `Primitive1` below; examples are `Hash`, `Mac` and `AEAD`, though the type can also be called something generic like

`Algorithm`), the *instances* of which represent algorithms. The approaches differ in how they generate the instances and what type they use (see the sections below).

8.4.1.1 Placement of the main API

As explained in section 8.3.2, some users need a fine-grained multi-step API (represented by the `Context` type and its `step1()` function below), though most users can be served with a simpler one-shot API that only requires a single function call (represented by the `execute()` function below). The one-shot functions can be implemented as methods of the algorithm instances:

```
mod primitive1 {
    pub [struct/enum] Primitive1;
    impl Primitive1 {
        pub fn execute(&self, ...) -> ... { ... } // one-shot API
    }
    pub struct Context(&Primitive1, ...);
    impl Context { // multi-step API
        pub fn new(algorithm: &Primitive1) -> Context { ... }
        pub fn step1(&mut self, ...) -> ... { ... }
    }
}
```

or as global functions which take the algorithm instance as a parameter:

```
mod primitive1 {
    pub [struct/enum] Primitive1;
    pub fn execute(algorithm: &Primitive1, ...) -> ... { ... } // one-shot API
    pub struct Context(&Primitive1, ...);
    impl Context { // multi-step API
        pub fn new(algorithm: &Primitive1) -> Context { ... }
        pub fn step1(&mut self, ...) -> ... { ... }
    }
}
```

The former design is an idiomatic use of methods in object-oriented programming (allowing the method syntax: `algo_instance.execute(...)`), but the latter design is more consistent because the one-shot function `execute()` and the multi-step type `Context` are next to each other on the same level and both receive a reference to the algorithm instance. For example, `ring` uses the latter design to let the user choose from the same set of hash algorithms when using the global `digest(&SHA256, ...)` function and when instantiating a `digest::Context::new(&SHA256)` for multi-step hashing, where a reference to the `SHA256` instance is embedded in the `Context` instance. Both designs let the user substitute the chosen algorithm trivially.

8.4.1.2 Placement of meta information

If the main type is a **struct** (and not an **enum**), the meta information can be placed inside the struct:

```
pub struct Primitive1 {
    pub meta_information_1: ...,
    pub meta_information_2: ...
}
```

The alternative is to provide getters for the meta information on the main type:

```
impl Primitive1 {
    pub fn get_meta_information_1(&self) -> ... { ... }
    pub fn get_meta_information_2(&self) -> ... { ... }
}
```

The latter has the usual benefits of using getters over public fields—most importantly, it allows to keep backward compatibility when changing the **struct**.

8.4.1.3 Instance retrieval

If the main type is an **enum**, the instances are simply its variants. For example, a global API function could be called like this:

```
execute(Primitive1::Algorithm5, ...)
```

No library currently uses **enum**s for algorithm selection. The main disadvantage is the lack of a common **struct** where meta information can be placed. Instead, the many meta information getters, as shown above, would have to be implemented with large **match** blocks or lots of **if** blocks because there is no common field to store the data.

For **struct**s, instances can be provided as static variables (which is what `ring` does):

```
pub static ALGORITHM1: Primitive1 = Primitive1 { meta_information_1: ... };
pub static ALGORITHM2: Primitive1 = Primitive1 { meta_information_1: ... };

// Usage with methods:
use primitive1::ALGORITHM1;
ALGORITHM1.execute(...);

// Usage with global functions:
use primitive1::{execute, ALGORITHM1};
execute(&ALGORITHM1, ...);

// Usage with multi-step API:
use primitive1::{Context, ALGORITHM1};
let mut ctx = Context::new(&ALGORITHM1);
ctx.step1(...);
```

This design matches the "A module full of constants" pattern by Hertleif [Her16b]. It has slight performance benefits and allows for an `insecure` submodule to contain deprecated and broken algorithms. But, when the main module contains a lot more than just the constants, it makes the algorithms slightly less discoverable because they are on the same level as all the types and global functions.

Alternatively, instances can be created by factory functions on the main type (which is what `rust-openssl` does):

```
impl Primitive1 {
    pub fn algorithm1() -> Primitive1 { Primitive1 { meta_information_1: ... } }
    pub fn algorithm2() -> Primitive1 { Primitive1 { meta_information_1: ... } }
}

// Usage with methods:
use primitive1::Primitive1;
Primitive1.algorithm1().execute(...);

// Usage with global functions:
use primitive1::{execute, Primitive1};
execute(Primitive1::algorithm1(), ...);

// Usage with multi-step API:
use primitive1::{Context, Primitive1};
let mut ctx = Context::new(Primitive1::algorithm1());
ctx.step1(...);
```

This design supports dynamic parameterization (`pub fn hmac(hasher: Hash) -> MAC`, for example), which is inherently impossible with statics (a constant can be *used* as a parameter value, but it cannot *accept* another algorithm as a parameter itself). Factories for insecure algorithms could be placed in a separate `InsecurePrimitive1` type, though they would confusingly have to return `Primitive1` instances, whereas `InsecurePrimitive1` instances do not make sense.

Both designs allow to specify a default algorithm by adding another item named `DEFAULT` or `default()`, respectively.

8.4.2 One struct per algorithm (trait-based)

This approach also uses one module per primitive, but inside the module there is one *type* per algorithm (as opposed to one instance of a common type). The only reasonable language item to use for the type is a **`struct`**. In order to let the user access the algorithms through a common API, a trait needs to be added and implemented by each of the algorithm **`struct`**s:

```rust
mod primitive1 {
    pub trait Primitive1 {
        fn execute(...) -> ... { ... }; // no "self", calls the multi-step API below
        fn new() -> Self;
        fn step1(&mut self, ...) -> ...;
        fn get_meta_information_1() -> ...;
    }
    pub struct Algorithm1;
    impl Primitive1 for Algorithm1 { ... }
}

// Usage:
use primitive1::{Primitive1, Algorithm1};
Algorithm1::execute(...);

// Usage with multi-step API:
use primitive1::{Primitive1, Algorithm1};
let mut ctx = Algorithm1::new();
ctx.step1(...);
```

Note that the `Primitive1` trait needs to be imported with **`use`** every time one of its methods is used. Meta information getters can be added as associated functions (known as *static methods* in other languages) on the trait. The trait and the structs do not necessarily have to live in the same module (`rust-crypto` has one module per algorithm), though a single module is preferable for better discoverability. At first it might seem that a lot of redundant code is required to implement the trait for every algorithm, but the implementer can use private helper functions for most of the code and/or wrap a prototypical implementation in a macro.

Because the algorithms are now distinguished types, the one-shot APIs like the `execute()` function above become associated functions. This leaves the *instances* of the algorithms for the multi-step APIs, where the state can be conveniently stored in the **`struct`**, that is, no separate `Context` is required. A default algorithm can be specified with a type alias (**`pub type`** `Default = AlgorithmX`). One implementation can be substituted with another by simply referencing the other type, as long as it implements the same relevant trait(s). And insecure algorithms can easily be moved to a submodule called `insecure`.

The trait-based approach is much more flexible than the ones above because structs can implement multiple traits. Special cipher features (such as resetting, additional input data like the "associated data" for AEAD ciphers and so on) could be exposed as separate traits. Empty traits can serve as "flags", for example to declare that an algorithm runs in constant time, fulfils certain security requirements, etc., and users of the algorithms could specify requirements with trait bounds. A struct for a single algorithm could even implement the traits for multiple primitives, which demonstrates that the flexibility of this approach is at the same time its downside, because it can lead to confusingly

complex designs. Note that the other two approaches can also, to some degree, benefit from traits, though the traits would need to be added in specifically and are not naturally part of the design.

For external helper methods that want to accept an algorithm as a parameter, and for parameterized algorithms like HMAC, generics need to be used instead of function parameters. A password hashing function, which is based on any hash function and applies a random salt, could have the following signature:

```
pub fn password_hash<H: hash::Hasher>(password: &str) -> Vec<u8>
```

and an `Hmac` type could generically depend on a hash function like so (which is how `octavo` implements it):

```
pub struct Hmac<H: hash::Hasher> { ... }
```

Even though this works throughout any use case, the syntax for accessing associated (static) members needs getting used to, as it involves the "turbofish"[13] operator:

```
Algorithm1::<InnerAlgorithm2>::execute(...);
let mut ctx = Algorithm1::<InnerAlgorithm2>::new();
let value = Algorithm1::<InnerAlgorithm2>::get_meta_information_1();

let mut ctx = hmac::Hmac::<hash::SHA256>::new();
let out_size = hmac::Hmac::<hash::SHA256>::output_size();
```

Instead of using this syntax, the library or its users can define type aliases for the parameterizations they need (e.g. `type HmacSha256 = hmac::Hmac<hash::SHA256>`).

The most outstanding and unique advantage of the type-based approach is the separation of the primitive APIs (**`trait`**s) from the implementations (**`struct`**s and **`impl`**s)—so much so that the traits could live in another crate. This separation can be useful to unify the APIs of various crypto libraries and it hopefully allows designing and thinking about the API independent of the implementation, as pointed out by @taoeffect [Sap+16, post 45].

At the time of writing, traits in Rust still have some limitations and the language design is still emerging. Future changes can interfere with a heavily trait-based API design or open up new possibilities. For instance, associated constants and types are being improved and the variants of an **`enum`** could become first-level types. As the Rust language is declared stable, trait-based designs will likely become more powerful in the future.

[13] https://twitter.com/steveklabnik/status/659034597062262784 (2017-04-01)

8.4.3 One module per algorithm (module-based)

The most static approach implements each algorithm in its own module. `sodiumoxide` and therefore `rust_sodium` use this approach.

```
mod primitive1 {
   pub  mod algorithm1 {
        pub fn execute(...) -> ... { ... }
        pub const META_INFORMATION_1: ... = ...;
        pub struct Context(...);
        impl Context {
            pub fn new() -> Context { ... }
            pub fn step1(&mut self, ...) -> ... { ... }
        }
    }
    pub mod algorithm2 {
        pub fn execute(...) -> ... { ... }
        pub const META_INFORMATION_1: ... = ...;
        ...
    }
}
```

Meta information is exposed through constants. Every module is self-contained in the sense that it does not share common structs, constants or functions with its siblings. Internally, they may still have common helpers or multiple similar modules can be generated from a single source file with macros. As far as their APIs are concerned, the modules are separate—but not independent: to make algorithms for the same primitive substitutable, the modules must have the same structure.

Instead of declaring all these modules at the crate level, they are grouped by the primitive they implement (and possibly submodules for insecure algorithms or the "hazardous materials" layer, see section 8.3.1). This, for one, makes them discoverable more easily and creates another module layer where general documentation can be placed. It also allows the primitive module to re-export one of the algorithm modules, effectively making it the default implementation:

```
mod primitive1 {
    mod algorithm1;
    mod algorithm2;
    pub use self::algorithm1::*; // algorithm1 is the default
}

// Usage:
primitive1::execute(...);
primitive1::algorithm2::execute(...);
let mut ctx = primitive1::Context::new();
```

This is the only design which hides the implementations completely from users who are not interested in them, i.e., neither an algorithm name nor the word "default" occur in the default use case. Furthermore, a user who previously used the default implementation—imported by `use library::primitive1`—can switch to a specific implementation simply by changing the `use` statement to `use library::primitive1::algorithm2 as primitive1`.

Another advantage is the possibility to reference the meta information statically. Particularly arrays always have to be declared with their size as part of the type (`[u8; SIZE]`), hence the size needs to be available statically. The trait-based design introduced in the previous section might also support that in the future, when associated constants for traits become stable.

Because the individual algorithms lack a type or an instance to represent them, they cannot be used to parameterize other functions or algorithms intuitively. Instead, one would have to pass a function pointer or a representative value (like a string or enum value), which is neither elegant nor type-safe. In `rust_sodium`'s case, parameterization is certainly not missing because it would make the API too complex for the target users. If an API also wants to support more complex use cases with a usable API, the lack of parameterization could become an obstacle.

Last but not least, the module-based design can be combined with the trait-based design to gain some of the benefits (parameterization, separation of API and implementation, ability to implement multiple traits, as described in the previous section).

8.4.4 Conclusion

The approaches detailed in sections 8.4.1 to 8.4.3 (summarized in table 8.1) all use a module for every primitive, but different language elements to represent algorithms. Instances are the smallest and at the same time the most dynamic elements, types are static yet flexible, and modules are rather bulky but still provide some elegant solutions.

From a usability perspective, the trait-based approach with one struct per algorithm (section 8.4.2) is preferable because it separates the interface from its implementation and because it intuitively places the one-shot API and the multi-step API of an algorithm/primitive in a single type as static and instance methods, respectively. It is also the most flexible and allows for fine-grained traits, though this advantage runs the risk of producing overly complex designs. And while traits in Rust are not matured, yet, they are already reasonably stable and feature-rich for this design to work well.

> **DO** create one module and trait per primitive and one struct per algorithm.

Approach	instance-based	trait-based	module-based
Section	8.4.1	8.4.2	8.4.3
For every primitive, create ...	a common struct	a common trait	a module
For every algorithm, create ...	an instance of the struct	a struct which implements the trait	a submodule
To choose an algorithm, the user ...	refers to a global variable / calls a factory	`use`-s the trait and refers to the struct	`use`-s the module
Select default with ...	yet another instance / factory	type alias	re-export (`pub use`)
Place one-shot API in ...	instance methods or global functions	associated functions	global functions
Place multi-step API in ...	a separate type (e.g. `Context`)	instance methods	a separate type (e.g. `Context`)
Place meta info in ...	instance fields or getters	associated functions	global variables or constants
Parameterize ...	by passing instances	with generics	-
Examples	`rust-openssl`, `ring`	`rust-crypto`, `octavo`	`sodiumoxide`, `rust_sodium`
Unique advantages	-	fine-grained traits possible, API separate from implementation	elegant selection of default algorithm, static meta information
Limitations	-	Rust traits are not yet mature, trait must be `use`-d	difficult parameterization

Table 8.1: Summary of alternative approaches to algorithm organization

8.5 Split into multiple crates

Some Rust programmers prefer many smaller crates over a big one.[14] `octavo` follows this principle and ships its hashing, MAC and key derivation modules as separate crates. The top-level `octavo` crate merely bundles and re-exports them—some refer to this as a "collection" or "meta" crate. The fact that the smaller crates are collectively managed, developed and shipped solves most of the issues with small crates: it ensures that the

[14] https://github.com/rust-unofficial/patterns/blob/bfd4b5c/patterns/small-crates.md (2017-03-23)

versions match, it makes them a single dependency to manage and it provides them under a single license. The `RustCrypto` project goes one step further and publishes a single crate per algorithm and one per primitive just for the traits. Through the "meta" crate, users can use these libraries like any other one—but they additionally have the possibility to only include a subset and hence reduce their build times and binary size.

However, too many crates make the overall ecosystem more confusing and the library harder to use, which is why `ring`'s owner refuses to do it.[15] Besides the possible confusion about similarly named crates on crates.io and the n:1 relationship to GitHub repositories, there are currently a few bugs in Rust tools which make separated libraries harder to use. My self-experiment showed that autocompletion with the `intellij-rust` plugin breaks and the meta crate's documentation becomes unusable.[16] In addition, users have to find and include every required crate manually. Thus, if the library does not strike the right balance, all users end up including the full-blown "meta" crate instead.

To find this balance, the data from the usage analysis (chapter 6) can help identify primitives which are often used alone and could benefit from a separate crate. As can be seen from figure 8.1, more than half of all dependent crates could benefit from a separate hash or MAC crate. Splitting off single *algorithms* usually does not warrant the resulting confusion from crates like `sha3`,[17] which are difficult to associate with the corresponding library. The only exception are widespread insecure algorithms like MD5 and SHA-1 (see section 8.2.2), which could be excluded completely rather than factored out and re-exported from the meta crate without breaking API compatibility, as they could still implement the common traits.

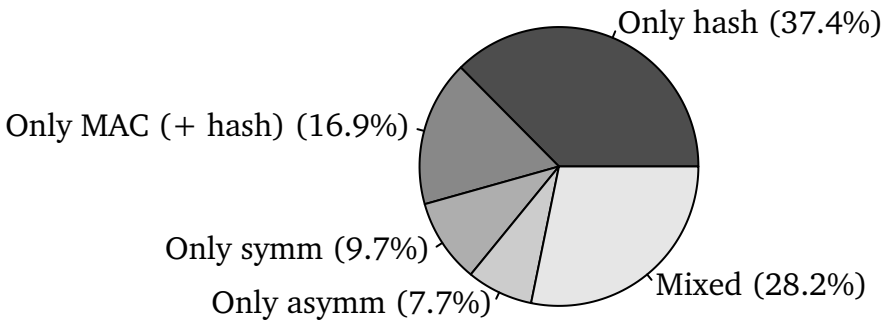

Figure 8.1: Proportions of crypto-using crates which only need a single primitive

DON'T split the library into too many crates, especially not prematurely. Only split off entire primitives if there is a user base who can provably benefit from it.

DO move insecure algorithms to a separate crate without re-exporting them.

[15] https://github.com/briansmith/ring/issues/432 (2017-03-23)
[16] https://docs.rs/octavo/0.1.1/octavo/index.html (2017-03-23)
[17] https://crates.io/crates/sha3

8.6 Defaults and future security

By setting default values for parameters or default algorithms for primitives, a library can spare its user the decision, for which the user might have to read up on the topic and weigh all the arguments for and against all possible choices. However, the history of defaults in cryptographic APIs teaches us that they are dangerous. The notorious example is Java's `Cipher.getInstance("AES")`, which uses the insecure ECB mode by default [EBFK13, sec. 4]. PyCrypto does the same and it sets the initialization vector (IV) to zero for CBC mode [DK14, sec. 5]. Georgiev et al. [GIJ+12, sec 11.1] even recommend application developers not to rely on any default values but to "always explicitly set the options necessary." Even perfectly secure values may become obsolete when new weaknesses are discovered or simply in the course of time because computing power becomes cheaper and longer keys are required for the same level of security.

On the other hand, not offering a default could be the worse decision. Choosing the appropriate block cipher mode is too much to ask of many users. The only solution they have is to turn to the documentation for advice and hard-code the recommended value, which is no better than if the library had set its recommendation as the default. And for all users who do not override the default, the library can change the values to match the latest recommendations without requiring the users' code to change.

The fundamental technical concern is backward compatibility: Ciphertexts and digests generated with outdated algorithms or parameters (block cipher mode, key/IV lengths, etc.) still have to be readable, as they might be received from outdated systems or be stored on disk. If Java changed its API so that `"AES"` refers to the more secure CBC mode, existing applications would break because they could not decrypt old ciphertexts.

These thoughts lead to the core issue: How can an application be made secure in the long term as the state of cryptography advances, requiring nothing but library updates on the application developer's part and especially no code changes? Of course, there is no perfect solution to this problem—quantum computers and other ground-breaking discoveries could turn cryptography upside down and make an API entirely obsolete. And yet, it is worth designing libraries to work as long as possible.

Higher-level APIs can solve this problem by storing an identifier for the used algorithm and its parameters alongside the data. For example, PHP's `password_verify()` function recognizes passwords hashed with any supported hash function because the used algorithm and its parameters are prepended to the hash value. Similarly, the Fernet format [Rar13] for authenticated encryption contains a version field to identify particular algorithm-parameter-combinations. Both examples show that this kind of backward compatibility is only suitable for high-level primitives in the "recipes" layer (see section 8.3.1). There are no defaults in such designs because the parameters themselves have been eliminated and replaced with sensible and updateable choices [Dev14].

For lower-level primitives, i.e., those in the "hazardous materials" layer, any attempt to solve backward compatibility for the user would fail—and it would also be a misdirected attempt: these lower-level primitives are supposed to serve as building blocks for higher-level protocols rather than being used in applications directly. In this scenario, the protocol can take care of backward compatibility, and it should also specify a value for *every* parameter to make implementations in different platforms compatible. Users of low-level APIs are knowledgeable enough to choose a suitable value themselves. Therefore, defaults in low-level APIs are neither necessary nor helpful.

Besides being discouraged, defaults for function parameters are also difficult to implement in Rust. There is no native support for optional parameters and no function overloading to provide alternatives with fewer parameters. Instead, libraries can define multiple functions with different names, declare `Option<T>` instead of `T` as the parameter type, use the builder pattern for constructors with many parameters or the parameter object pattern for regular functions, or expose the default value separately, e.g. `const DEFAULT_KEY_LEN: usize = 256 / 8;`. All of these solutions make the API more difficult to learn and should be used with care. Finally, if a default value is necessary at all, it has to be documented well to make the API understandable and help the user switch to another value if necessary.

So in short, there should be very few defaults in a crypto library: on higher levels, the choice itself should be eliminated and be taken care of transparently by the library, and on lower levels, all parameters and algorithms should be exposed without defaults.

There are two general exception to this rule: Firstly, some parameters can be made *optional* without requiring a specific default value that represents a decision. For example, when an encryption function is called without an IV, the library should generate a random one in a secure way and return it. This prevents users from hard-coding IVs and saves them a call to the library's IV generation function. Secondly, default algorithm-parameter-combinations should be provided for high-level primitives, e.g., the default for authenticated encryption could be AES-GCM-256. This extends to hashing and HMAC, as they are widely used (see section 6.2). Section 8.4 discusses how default algorithms can be specified with respect to different library structures.

> **DO** choose a secure default algorithm with secure parameter values for *high-level* primitives and eliminate parameters or other choices where possible.

> **DON'T** provide any defaults in *low-level* APIs.

8.7 Strong types

Sequences of bytes (`Vec<u8>` or `[u8]` in Rust) represent most of the data in cryptographic algorithms: inputs, ciphertexts, different kinds of keys, nonces, tags and digests. But they are easily mixed up:

```
fn hmac(key: &[u8], data: &[u8]) -> Vec<u8> { ... }
```

For this function, `hmac(&data, &key)` is a perfectly valid call and it will only fail at runtime if the data exceeds the maximum key length (and the implementation does not just truncate it). A confusion like this rarely results in a vulnerability *and* is usually discovered with a simple functionality test because the mix-up breaks decryption or verification or causes other runtime failures. And yet, a well-designed API should prevent such mistakes at compile time to improve the API's usability.

To harness the type system, at least some of these data kinds need to be represented by types, which can be simple wrappers:

```
pub struct Key(pub [u8; 32]); // Known length
pub struct Tag(pub Vec<u8>); // Unknown length
```

This changes the wrong call to `hmac(&data, &Key(key))`, which fails. The user data (inputs and ciphertexts) do not need a separate type and are most intuitively represented by arrays or vectors, as before. Ideally, the types serve other purposes than just the type safety of the function call. `sodium` offers functions like `fn gen_key() -> Key` as type-safe and expressive variants of its RNG function, for example, which encourages using the types throughout the application. The mere fact that there is also a `fn gen_nonce() -> Nonce`, which connects to the `nonce` parameter through the common type, tells the user that the nonce should also be generated (randomly), which is not clear to all users as the controlled experiment showed. According to Hertleif [Her16b], in an elegant API, "types are cleverly used to prevent logic errors, but don't get in your way too much."

Note that there should not be a single type `Key` for all kinds of keys used by the API. Using a single key for different cryptographic purposes is considered bad practice because it can lead to subtle vulnerabilities [YHR04, sec. 9.2]. Thus, at least every primitive should define its own key type.

> **DO** use type-safe wrapper structs for keys, nonces, tags and digests to avoid confusion.

Boolean, integer and string parameters are also easily mixed up when they represent certain fixed values rather than user data. Especially boolean parameters are frowned upon [Hid11; Leb11], as they can often be removed entirely and replaced with two versions of the function with telling names. Georgiev et al. [GIJ+12, sec. 4.2 and 7.1] report on a particularly severe case:

```
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, true);
```

This PHP code sets options for an SSL connection with the cURL library and runs without warnings in older PHP versions. The second call effectively *disables* host verification, making the application vulnerable to man-in-the-middle attacks because the peer can be impersonated by presenting any valid certificate for another host. The `CURLOPT_SSL_VERIFYHOST` parameter is an integer and defaults to `2`, which is the only secure option. While a value of `0` disables verification entirely, as one would expect, a value of `1` (or **`true`**) only enables it partly. In this example, various factors play a role: PHP is weakly typed and silently coerces **`true`** to `1`, the cURL library has a similar *boolean* parameter `CURLOPT_SSL_VERIFYPEER`, `1` happens to be an insecure option, and the mistake does not raise any warnings or cause failures.

Like many other languages, Rust provides enum types to prevent such mistakes, with the added benefit of more usable APIs and more readable code. Enum types and enum variants need expressive and specific names. They should be reused with care, as the use of a generic **`enum`** `State {Enabled, Disabled}`, for example, can lead to the same mix-up issues as plain booleans. For one parameter in particular, namely the chosen algorithm, there are alternative and more powerful solutions, though (see section 8.4).

> **DO** use enums for parameters with a small set of choices.

8.8 Keys, nonces and seeds

As discussed in the previous section, keys and nonces should be encapsulated in respective types. They also need to be generated randomly and securely by a CSPRNG (see section 2.2.1). The pure-Rust libraries do not implement an RNG because, unlike C, Rust has an officially endorsed `rand` crate that provides direct access to the operating system's CSPRNG through the `OsRng` type and explicitly advertises it for cryptographic purposes.

The wrapper libraries additionally hand through the RNG of the underlying C library (OpenSSL, BoringSSL or libsodium). Besides marginal differences (e.g. OpenSSL only seeds from the OS[18]), the major downside of the libraries' own RNGs are the less ergonomic interfaces, as the user has to preallocate and fill a vector/array in two separate steps. `ring` supports heap-less environments and the `rand` crate does not, which forces `ring` to offer its users an RNG.

[18] https://github.com/openssl/openssl/issues/898 (2017-03-23)

Users can easily overlook the library's own RNG and the `OsRng` and use the insecure `rand::random()` function instead. A similar situation in Java, where the insecure `java.util.Random` and the secure `java.security.SecureRandom` coexist in the JDK, has led to accidental misuse [EBFK13, sec. 6.2], see also [ANA+15; LCWZ14]. In my self-experiment with `rust-crypto`, even though I had found the correct `OsRng`, I was unsure about using it. The crypto libraries' documentation should therefore point to a secure RNG, whether it is included or from the `rand` crate. As mentioned in the previous section, `sodium` cleverly uses strong types and promotes its `fn gen_key() -> Key` function, so that users do not have to search for an RNG and the right key/nonce length at all.

> **DO** offer key and nonce generation functions with the appropriate type.

The controlled experiment and other research show that some users provide insecure keys or nonces, particularly zeros, hard-coded or otherwise reused values, when they cannot find a generation method for them in the library. Lazar et al. [LCWZ14, sec. 2.3.1] find hard-coded encryption keys to be a common mistake and hypothesize that developers believe that the keys cannot be read from the compiled binary. Unfortunately they can and an attacker in possession of the key can decrypt and forge any message. For the same reason, hard-coding other kinds of keys (e.g. MAC keys), nonces or seeds for RNGs is highly insecure. Lazar et al. [LCWZ14, sec. 3.1] name the Keyczar library as a positive example, where the user does not specify the key itself but the path to a key file, which discourages hard-coding. The API is obviously geared towards the use of key files, but other scenarios where the key needs to be read from memory or derived from another secret can be implemented, too. A solution like this is also possible and desirable in Rust because it helps the user load and store the keys properly.

> **DO** offer simple (default) ways to load and create key files.

To prevent nonce misuse entirely, Devlin [Dev14] proposes that encryption functions generate the nonce internally and return it, which can be done separately (returning a tuple) or concatenated with the ciphertext (see section 8.6). While the latter is desirable because it results in a higher-level primitive, the former only applies to low-level APIs, whose users should know how to generate a nonce anyway.

> **DO** generate nonces internally for *high-level* primitives and concatenate them with the ciphertext.

Egele et al. [EBFK13, sec. 5.2] use static analysis to detect hard-coded keys and seeds and non-random nonces in Android apps. Their detection can be executed by app stores as part of the approval process for uploaded apps or by developers as part of the build process to ensure that no bugs of this kind have been introduced. Rust programs are also amenable to static analysis, but priority should be given to the solutions discussed above, as they prevent these mistakes from happening in the first place.

Forler, Lucks and Wenzel [FLW12] use features of the Ada programming language to prevent the accidental or intentional reuse of nonces. Unfortunately, it is unclear how

their approach works in the usual symmetric encryption scenario and they have not responded via e-mail or on GitHub.[19] Using the language specification and thus the compiler to prevent nonce reuse generally sounds like a good approach that could also work in Rust, especially because Rust's memory models allows values to be "consumed" by functions so that they cannot be used anymore afterwards.

8.9 Constant-time comparisons

Digests like MAC values or the authentication tags used by the AEAD primitive have to be compared in constant time, but the runtime of Rust's default comparison function `PartialEq.eq()` and the `==` operator depends on the length of the common prefix, making these comparisons insecure (see section 2.2.8). All crypto libraries except `octavo` offer constant-time comparison functions, usually inside a utility module that contains nothing else. However, users will probably never search for a proper comparison function if they are not aware of the side-channel attacks enabled by variable-time comparisons. The documentation should definitely explain the risks, point to the right solution and use it in all code samples (see section 8.1.2), but not all users read the documentation. Since only expert users can be expected to know the danger, crypto libraries need additional strategies to *advertise* their constant-time comparison functions and to prevent accidental misuse of Rust's insecure/default ones.

In the case of AEAD encryption, the library's API can simply demand the tag as a parameter and refuse to return the decrypted plaintext if the tag does not match. The comparison can then be implemented securely as part of the decryption function, making it impossible to accidentally misuse the wrong comparison function. All crypto libraries do this properly (except `octavo`, which does not support AEAD yet).

The case of MAC (or hash) digests is more difficult because it really only consists of one operation, as opposed to the encrypt-decrypt pair of AEAD. The digest is generated to sign the message, and later it is generated again to verify (that is, compare) the signature. An exemplary strategy implemented by `rust-openssl`, `ring` and `sodium` is to expose two separate top-level methods (or respective other structures) that reflect the sign-verify pair—even if the verification function does nothing more than calling the signing function and doing a secure comparison. For instance, `ring` has a `sign()` and a `verify()` function right next to each other in its `hmac` module and uses separate key types. This makes the correct verification method much harder to miss.

> **DO** offer a dedicated signature/MAC verification function based on a constant-time comparison.

[19] https://github.com/cforler/Ada-Crypto-Library/issues/8 (2017-03-23)

`rust-crypto` does not implement this solution but has another way to advertise its comparison function: the digest is wrapped in a `MacResult` struct, which implements the `PartialEq` trait itself and redirects to the constant-time comparison function.[20] Thus, comparing two `MacResult` instances with `==` is secure. The `MacResult::code()` getter, which returns the digest as a plain byte slice, has an appropriate warning in its documentation. Unfortunately, this is not enough, as figure 6.5 on page 91 shows: 28% of users end up with an insecure comparison—significantly more often than users of the other libraries. In the third stage of my self-experiment with `rust-crypto` (see section 7.1.4), I also used an insecure `==` comparison on two `Vec<u8>` because I was unaware of the problem and had used regular hash values just before.

Nevertheless, `rust-crypto`'s solution is worth adopting, as it introduces a strong type for digests at the same time (see section 8.7). To improve the solution, retrieving the bare representation of the digest (`[u8]`, `Vec<u8>` or a reference to them) has to be discouraged. An off-putting name like `.raw_digest()` or even `.raw_less_secure()` cautions users about the bare digest and might prompt them to read the documentation. More importantly, the function should rarely have to be called. As the usage analysis in section 6.4 showed, most users need to send or to store the digest as a Base64 string. Instead of exposing the bare digest, the wrapper struct could interface directly with the functions for sending and storing, e.g. by implementing the `Encodable` and `Decodable` traits from the official `rustc_serialize` crate and advertising it through code samples.

> **DO** use a wrapper type for digests, which implements `PartialEq`, `Encodable`, `Decodable` and other useful traits to avoid the need for bare digest values.

8.10 &mut parameters

Instead of or in addition to returning a value, a function can write to a memory location specified by one of its parameters. These parameters are regular pointers in C, `out` parameters in C# and `&mut` parameters (so-called *mutable borrows*) in Rust. Their use is generally discouraged in all languages because it obfuscates the data flow. While it is necessary to use in some languages when a function absolutely needs to return two separate values, modern languages like Python and Rust natively support tuple types, which can also be returned from functions. The official Rust style guide recommends: "prefer compound return types to out-parameters."[21]

[20] https://docs.rs/rust-crypto/0.2.36/crypto/mac/struct.MacResult.html (2017-03-23)
[21] https://doc.rust-lang.org/1.12.0/style/features/functions-and-methods/input.html (2017-03-23)

However, returning values (be it a single value or a tuple) has a relevant downside: the stack size of the return type needs to be static (known at compile time). To work around this problem, functions allocate flexible space on the heap (e.g. with a `Vec<u8>` in Rust) and return a reference to the heap location. The reference itself has constant size and thus solves the problem. This fundamentally applies to all programming languages, though higher-level programming languages like Python or Java work with heap-allocated types most of the time so that the developer does not notice the problem. In the case of systems programming languages like Rust, some runtime environments do not have a heap at all, forbidding the use of vectors and strings entirely.

Crypto libraries often have to return plaintexts or ciphertexts whose size is not static. Using `&mut` parameters has the following advantages and disadvantages:

++ Compatibility: Works in heap-less environments.
+ Memory usage: Can be used in-place, that is, the en-/decrypted result is written in the memory location of the input. This saves half the memory usage and is especially relevant for huge plaintexts/ciphertexts.
+ Performance: Saves unnecessary allocations and copy operations. Compared to many other operations, including the cipher operations themselves, this benefit is negligible.
– Readability: The code does not look as intuitive, preallocating a target array/vector requires extra lines of code.
– Understandability: When a function returns a `Result<T, E>`, it is not inherently clear whether/which output parameters will be filled in the error case and what they contain.
–– Usability: Requires the caller to find out about the proper size and dynamically allocate a vector:

```
// Without preallocation:
let result = crypto_primitive.execute(...);

// With preallocation:
let mut result = vec![0u8; crypto_primitive.output_bytes()];
crypto_primitive.execute(..., &mut result);
```

In my self-experiment (see section 7.1.9), preallocation was often the most difficult aspect of an API, especially because I was not aware of the simple solution with the `vec!` macro shown above.

Fortunately, the most significant advantage (compatibility with heap-less environments) and the remedy for the most significant downside (usability) can be combined in a

single design, as implemented by `rust-openssl`: it simply offers both variants simultaneously. Its fine-grained API (see section 8.3.2) in the `symm::Crypter`[22] type works with `&mut` parameters and hence without heap allocations. The corresponding "one-shot" functions[23] `symm::encrypt()` and `symm::decrypt()` return a heap-allocated vector, making them easier to use. This pattern is not used consistently across the `rust-openssl` API (e.g. the fine-grained `hash::Hasher` type still returns a vector), but it could be applied everywhere. Similarly, `sodium` has a `stream_xor()` and a `stream_xor_inplace()` function next to each other in its `stream` module.[24]

> **DO** offer a high-level API variant with return values instead of `&mut` parameters. Consider designing an entirely allocation-free variant in addition, which can coincide with the fine-grained API (see section 8.3.2).

8.11 Conclusion

This chapter presented recommendations for crypto API design in Rust based on experiments and the literature. The most important recommendations pertain to rather non-technical matters, namely the documentation, code samples and the structure of the library. Offering and advertising functionality like RNGs, constant-time comparison functions and key management functions already improves misuse resistance. Higher-level primitives, which choose appropriate parameter values, generate nonces for the user and avoid `&mut` parameters, can be implemented alongside the existing lower-level primitives to improve usability and misuse resistance. In addition, the type system can be used to detect errors like mixed-up parameters.

For those libraries that do not want to dedicate their development time to improving their usability (like `rust-crypto` and `rust-openssl`) there is still one important recommendation: they should *state this non-goal explicitly* on the front pages of their repositories and documentations and recommend an alternative library (or a good overview of other libraries) for beginners and those who appreciate better usability for average use cases.

[22] https://docs.rs/openssl/0.9.1/openssl/symm/struct.Crypter.html (2017-03-23)
[23] https://docs.rs/openssl/0.9.1/openssl/symm/index.html (2017-03-23)
[24] https://docs.rs/rust_sodium/0.1.2/rust_sodium/crypto/stream/chacha20/index.html (2017-03-23)

9 Conclusion

Section 9.1 looks at the future of the Rust crypto ecosystem, section 9.2 discusses possible directions of future scientific research and section 9.3 retrospectively summarizes the entire thesis.

9.1 Future of crypto in Rust

The Rust crypto ecosystem obviously depends on the Rust language as a whole to become established over the coming years. Like any other programming language, Rust has a crypto ecosystem to provide cryptographic functionality to its users, and the current libraries for primitives analysed in this thesis already provide a good range of functionality. However, the crypto ecosystem of Rust has the potential to serve even more users because **Rust is a good language to implement cryptographic primitives in** and to make them available to other languages' ecosystems.

Currently, the **main obstacle is a lack of developers** or funding (see section 5.5.2). This is an important issue to address because any library can benefit from (more) reviews and design discussions—and for crypto libraries, fixing bugs and improving misuse resistance is crucial. Therefore, section 5.7.1 discussed various approaches to **recruit more developers**.

Before Rust sees a major increase in usage numbers, the crypto libraries will have to **polish and stabilize their APIs**. The Rust platform itself has a couple of open issues (see section 5.7.2) that will hopefully be fixed soon to enable better cryptographic implementations and APIs. As the analyses in this thesis showed, **not all crypto library developers care equally about usability and misuse resistance**. For those who do, this thesis contributes quantitative data and experimental observations to base decisions on, as well as discussions and recommendations regarding the most relevant API design topics. For those libraries that focus on other goals, there should be **corresponding warnings** from the Rust community in general and inside the libraries' own documentations **to prevent uninformed users from choosing them**.

There are good reasons to deprioritize a usable API design and focus on other aspects, especially for the libraries like `rust-crypto` and `octavo` which implement cryptographic

primitives in Rust. To make such libraries successful in the medium term, they need a **wrapper library** which provides higher-level primitives and APIs based on their low-level primitives. Such a high-level wrapper would compete with other high-level libraries like `sodiumoxide`, but it would not need bindings to foreign-language code. `ring` also approaches the same goal, but instead of bindings it directly includes the BoringSSL code and often ports it to Rust, and instead of merely wrapping a lower-level library it constantly evolves its interfaces to become more high-level and especially more misuse resistant. As the competition is still very much open, it will be interesting to see how these libraries develop in the future.

In the long term, once the crypto libraries have matured and a suitable candidate emerges, an **officially endorsed crypto library** is desirable—where "endorsing" means making it a rust-lang crate because the standard library itself is generally kept small. Such an official crypto library would spare Rust users some research and another security-relevant choice. And it would allow the Rust crypto developers to concentrate their implementation and review efforts on a single product.

9.2 Future work

While the analyses of the developers and processes in the Rust crypto ecosystem (the contributors survey and the GitHub analyses) could answer the corresponding research questions in a satisfactory manner, the analyses regarding the usage and usability of the current crypto libraries suggest several starting points for further research.

Firstly, the rather quantitative usage analysis presented in chapter 6 found that hashing is the most frequently used primitive and unauthenticated encryption has many more users than AE, which is in stark contrast to the common crypto tasks identified by Nadi et al. [NKMB16]. A **qualitative, open-coding analysis** similar to theirs could **identify the tasks** of those users, whether they actually need hashing and unauthenticated encryption or should really be using something else, and how well they cope with the existing APIs.

Secondly, a similar qualitative analysis could find out **why the completely broken algorithms MD5 and SHA-1 are so widespread** (see section 6.3.1). A qualitative analysis of these usages could find out *why* these users still need MD5 and SHA-1 and what they use hashing for in general. It can be suspected that these users do not need a cryptographically secure hash function but just any (fast) hash function for other purposes like comparing two files without any security requirements. They possibly turn to full-blown crypto libraries because they know that these algorithms, which are not in the standard library, can be found there. If this suspicion can be confirmed, the solution would be to exclude these algorithms from all crypto libraries and to offer them as separate crates (as the `RustCrypto` fork is already doing).

Thirdly, the usage analysis itself could be **repeated** as soon as more developers and especially beginners have started using the crypto libraries in Rust, to see how their usages differ and possibly adjust the crypto libraries accordingly. By the time a repetition of this analysis becomes interesting, the number of usages will have grown so much that only a small sample of the newest usages can be analysed manually. Alternatively, the analysis could be **automated with large-scale static analysis tools** (once they are available for Rust) to track the usage over time with less manual effort. Such static analysis tools can be helpful in general, as they can inform library developers about the number of users who would be affected by a certain, breaking API change, and possibly they can even post automated issues to affected GitHub repositories. The analysis could also allow to find all usages of a function on GitHub so that API designers can quickly check whether their current API is misunderstood or not. Being a clean, fairly simple and explicit language where everything needs to be imported with `use` explicitly, Rust seems relatively amenable to such static usage analyses.

Lastly, the controlled experiments presented in section 7.2 could be repeated in a number of ways: with a similar pool of participants (possibly from the same university) in order to **measure the progress** of the libraries over time, with other libraries to get experimental data and observations for them as well, or with a **think-aloud protocol** to get more detailed insights for API design.

9.3 Summary

Rust has set out to rival C and C++. If it succeeds, good Rust crypto libraries will be in high demand, especially since Rust promotes memory safety and thus suits security-critical applications. The ecosystem is growing and encompasses several crypto libraries already. Now is the right time to ensure that design weaknesses, which have made other crypto libraries difficult to use and have led to vulnerabilities in end-user applications, and other mistakes are not repeated in the emerging Rust crypto libraries. This thesis aims to analyse the growing crypto ecosystem and to highlight potentials for improvement, with a particular focus on usability and misuse resistance.

The general definition of a *crypto ecosystem* serves as a foundation to systematically analyse Rust's crypto ecosystem, as it describes the populations in the ecosystem (platform, libraries, contributors and users) and their interactions. Fifteen research questions based on this ecosystem structure were used to guide the analyses, which employed various research methods: a systematic search, which found 80 crypto libraries and identified 4 major ones, a survey with 20 Rust crypto contributors, several GitHub analyses including a manual analysis of all 710 issues and 1001 PRs of the major libraries as well as a self-experiment and a controlled experiment (in collaboration with my supervisor Kai Mindermann) to test the libraries' usability.

Overall, the contributors are typical open source developers, though they have above-average programming experience and there are a few crypto experts working in the Rust crypto ecosystem. They collaborate in typical ways on GitHub, where discussions take place at least for the larger projects. Because of a general lack of contributors and because most spend their unpaid free time to work on Rust crypto, almost all libraries only have a single main developer who contributes (much) more than all others combined. The most frequently used cryptographic primitives are hashing, HMAC and symmetric encryption. It would be interesting to find out why hashing is so popular, especially the cryptographically insecure MD5 and SHA-1, and possibly decide to move them to separate, non-cryptographic crates.

`ring` and `sodium` make usability and misuse resistance a top priority and there are regular discussions about these topics on GitHub. Accordingly, they make almost none of the mistakes that related usability studies have found in other crypto libraries and they proved the most usable in the self-experiment. `rust-crypto` and `rust-openssl` are more difficult to use (although `rust-crypto` helped more participants of the controlled experiment succeed with its fitting code sample) and especially less misuse resistant, which is to be expected as they explicitly prioritize other goals. Despite that and the difficult underlying OpenSSL library, `rust-openssl` is easier to use than `rust-crypto` because its interface is more high-level. `octavo` still lacks many features and does not prioritize usability, either.

Being the most downloaded libraries with relatively canonical names, `rust-crypto` and `rust-openssl` run the risk of attracting uninformed users who likely misuse their interfaces and produce insecure code. While the Rust crypto community recommends `sodiumoxide` and especially `ring` to beginners at every opportunity, these recommendations often get lost in long discussions and chat logs. To avoid cryptographic misuse, the Rust community and maybe even `rust-crypto` and `rust-openssl` themselves should actively inform users about easier-to-use alternatives. Apart from that, the thesis gathered suggestions to attract more contributors and to improve the Rust platform for cryptographic purposes, and it discussed several usability topics, which should be considered when developing new cryptographic APIs in Rust. One particular API that ought to be built in the future is a common, official interface for important cryptographic primitives. However, its development should be based on certain platform features that are to be built in the near future. Future academic work includes qualitative extensions of the quantitative usage analysis presented in this thesis, a repetition and possibly automation of this analysis to gain insights about API usage over time and new controlled experiments with other libraries or newer versions of the same libraries.

A Supplementary material

The following materials are provided for download:

1. all *figures* included in this thesis (as PDF files) and the corresponding numbers or raw data cited in the text (CSV), as well as further figures not used in the text
2. the *R scripts* and *PowerPoint files* used to generate these figures
3. the full list of identified Rust crypto *libraries* along with their *categorization* and number of downloads and dependent crates (Excel sheet)
4. the *survey structure* including all questions (LimeSurvey LSS file)
5. the raw *survey responses* except the part that would make participants personally identifiable (CSV)
6. the *R scripts* used to *download* GitHub issues, pull requests, contributor data and commits
7. the raw data of our *GitHub issues and pull requests* analysis with *topics* assigned to every issue, and the *type and length of discussion* for feature-relevant issues (multiple Excel sheets and RData files)
8. a list and categorization of all *usages of the major crypto libraries* (multiple Excel sheets)
9. the *code* that resulted from the self-experiment (multiple Rust projects)
10. all *code snippets* from chapters 2 and 8 in a runnable application
11. instructions on how to read and use these materials

Download URL: http://philippkeck.de/download?202
Mirror: https://osf.io/edjtb/

B Rust contributors survey questionnaire

This is the full questionnaire, which was presented to survey participants as a web page generated by LimeSurvey [LS16]. The few mandatory questions are marked with a star.

Thank you for participating in this survey, we appreciate your support! This should take you about 10-20 minutes (less if you are not a main contributor, more if you spend lots of time expressing many thoughts). Please feel free to skip any questions you cannot or do not want to answer. The results of this survey are used for a master thesis ("Analysis and improvement of the crypto ecosystem of Mozilla Rust") at the University of Stuttgart, and possibly in an academic research paper. If you have any questions, please do not hesitate to contact <e-mail address>.

B.1 Demographics

Q1.1 *What is your gender?* Male/Female/No answer
Q1.2 *How old are you?* 17 or younger; 18-20; 21-24; 25-29; 30-34; 35-39; 40-49; 50-59; 60 or older
Q1.3 *Which country (or US state) do you live in?* Free-text
Q1.4 *What is the highest level of school you have completed or the highest degree you have received?* Less than high school degree; High school degree or equivalent (e.g., GED); Some college but no degree; Associate degree; Bachelor degree; Graduate degree
Q1.5 *Which of the following categories best describes your employment status?* Student, not employed; Student, employed part-time; Employed, working 40 or more hours per week; Employed, working 1-39 hours per week; Not employed, looking for work; Not employed, NOT looking for work; Retired; Disabled, not able to work
Q1.6 *Counting all locations where your employer operates, what is the total number of persons who work there?* 1; 2-9; 10-24; 25-99; 100-499; 500-999; 1000-4,999; 5,000+
Q1.7 *Which industry does your employer belong to?* Software development; Other IT; Other technical; Anything else

B.2 Your involvement

Q2.1 *How many years of experience do you have in . . . ?* Numeric input
a. *. . . programming in general (in any language)?*
b. *. . . programming in Rust?*
c. *. . . cryptography (e.g. from hearing a lecture, working on a project, ...)?*
d. *. . . cryptography in Rust?*
Q2.2 *How would you rate your own cryptography skills?* Novice; Educated (heard a lecture or similar); Experienced (worked on crypto topics for a few weeks); Advanced (worked on crypto topics for at least a year); Expert (written blog/book or given talks or developed new crypto algorithms or similar)
Q2.3 *How would you rate your own Rust skills?* Novice; Educated (heard a lecture about Rust, done a tutorial or similar); Experienced (worked with Rust for a few weeks); Advanced (worked with Rust for at least a year; can give advice to others); Expert (written blog/book or given talks or Rust core developer or similar)
***Q2.4** *Do you implement or have you implemented cryptographic algorithms, interfaces, libraries, etc. in Rust?* Yes/No
Q2.5 *If you answered yes, what is your primary motivation for working on (crypto in) Rust?* Free-text
Q2.6 *How many hours per week do you currently spend on . . . ?* Numeric input
a. *implementing crypto in Rust*
b. *implementing crypto in other languages*
c. *other crypto-related tasks (documentation, testing, consulting) in Rust*
d. *other crypto-related tasks (documentation, testing, consulting) in other languages*
Q2.7 *How many hours per week did you spend on . . . when you were most involved?* Numeric input, same 4 sub questions as above
Q2.8 *Which of the following are you paid for (e.g. as part of your job, a project, a PhD, etc.)?* Multiple choice, same 4 items as above
Q2.9 *Which of the following were you paid for in the past?* Multiple choice, same 4 items as above
***Q2.10** *Are you a contributor to one or multiple published Rust crypto libraries?* Yes/No

B.3 Rust's crypto ecosystem

Q3.1 *In what stage is Rust's crypto ecosystem overall?* Infancy; Early stages; Maturing; Mature; Deprecated; + free-text comment box
Q3.2 *Are there any significant influences from people without cryptographic knowledge onto the crypto ecosystem of Rust? If so, from whom and in what way?* Free-text
Q3.3 *Which Rust crypto library would you spontaneously recommend for a Rust beginner who simply wants to encrypt a string, and why?* Free-text
Q3.4 *Which Rust (TLS) library would you spontaneously recommend for a Rust beginner who wants to securely query a server over HTTPS, and why?* Free-text

B.4 API design

Q4.1 *How important is crypto API design in relation to the security of the implementation itself, with respect to how many end-user applications are vulnerable?* Much less important; Less important; About the same; More important; Much more important
Q4.2 *If any come to mind, please name examples of problematic crypto APIs in Rust (otherwise just skip the question).* Free-text
Q4.3 *If any come to mind, please name examples of exceptionally good/safe/clever crypto APIs, both in Rust and other languages that do or could serve as a model (otherwise just skip the question).* Free-text
Q4.4 *What obstacles do you currently see (with respect to the Rust language and ecosystem) that prevent the implementation of usable (crypto) APIs?* Free-text

B.5 Identifiable part

If you do not work on any crypto library in Rust, please skip this page (just leave everything empty and click "Next" right away).

Otherwise: Because of the low total number of contributors, it is likely that you will be personally identifiable if you answer the following questions. However, we do guarantee that we will not publish any results from this survey that make you identifiable without your consent. That is, we might publish something like "The average developer. . . " or "Most developers...", but we will ask for your explicit consent before publishing something like "The developers of library X spend Y amount of time". You can (a) skip this page to avoid being identifiable entirely (just leave everything empty and click "Next" right away), (b) specify your e-mail address or #rust-crypto IRC name so that we can contact you or (c) fill in the answers without contact information, in which case we will not publish anything that might identify you.

Q5.1 *Optional contact address (e-mail / #rust-crypto IRC)* Free-text
Q5.2 *Which crypto library in Rust do/did you work on? (If there are multiple, you can specify them in a separate line/paragraph each and do the same for the following fields.)* Free-text
Q5.3 *Off the top of your head, how many main contributors (possibly including yourself) work on this library?* Free-text
Q5.4 *What are the goals of the library (most important ones first)?* Free-text
Q5.5 *Does the library guarantee API stability already or what is the priority of backward compatibility versus other goals such as performance, safety, usability, etc.?* Free-text
Q5.6 *Is the library modelled after another one (wrt. functionality, algorithms, API design, documentation)? Are there other sources that directly inspire the library? If so: Which ones?* Free-text
Q5.7 *If the library is a wrapper or fork of another implementation: What is the relationship between your library and the wrapped/forked one? Bug reports / fixes contributed (both ways, one way)?* Free-text
Q5.8 *How are design decisions made (e.g. which functions to include, how to structure the code, which default values to use, etc.)? (Where/How) Do discussions take place? Is a full consensus required or does a single developer have the authority to decide?* Free-text
Q5.9 *How do you and the other library contributors organise the development process?* Multiple choice: Ad-hoc; Code-and-fix; Systematic; Agile; Waterfall; Scrum; Chaos model; Test-driven development; Passive (bug fixes and pull requests only); Other (specify)

B.6 Additional comments

If you have anything else you want to say, please write it down here.

Q6.1 *About cryptography in Rust* Free-text
Q6.2 *About crypto API usability research* Free-text
Q6.3 *About the survey* Free-text

This is the end of the survey, thank you for your time! If you have any questions, please do not hesitate to contact <e-mail address>.

Acronyms

AAD additional associated data. 24, 93, 96
AD associated data. 24, 25, 102
AE authenticated encryption. 24, 25, 33, 43, 102, 108, 140
AEAD authenticated encryption with associated data. 25, 26, 81, 92, 93, 96, 97, 99–103, 109, 118, 124, 135
AES Advanced Encryption Standard. 24–26, 33, 43, 53, 54, 70, 76, 83, 94, 100, 102, 104, 107, 131
API application programming interface.
ASP.NET Active Server Pages .NET. 42
BDFL benevolent dictator for life. 65, 70
CBC Cipher Block Chaining (mode for block ciphers). 24, 33, 43, 70, 83, 92, 100, 102, 107, 130
CFB Cipher Feedback (mode for block ciphers). 24, 83
CLR Common Language Runtime. 42
CNG Cryptography API: Next Generation. 42
CNSA Commercial National Security Algorithm. 51
CSPRNG cryptographically secure pseudorandom number generator. 20, 43, 77, 108, 133
CTR Counter (mode for block ciphers). 24, 83, 100
CVE Common Vulnerabilities and Exposures. 31
DES Data Encryption Standard. 24
DoS Denial of Service. 73
ECB Electronic Codebook (mode for block ciphers). 24, 33, 83, 92, 102, 130
ECC elliptic curve cryptography. 83
FFI foreign function interface. 74
GCC GNU Compiler Collection. 20
GCM Galois/Counter Mode (for block ciphers). 25, 83, 102, 107, 131
HKDF HMAC-based key derivation function. 22, 83
HMAC keyed-hash message authentication code. 22, 26, 43, 81, 83, 85, 86, 90–93, 96–104, 112, 119, 120, 125, 131, 142
HTML Hypertext Markup Language. 19
HTTP Hypertext Transfer Protocol. 37, 93
HTTPS Hypertext Transfer Protocol Secure. 38, 56

IDE integrated development environment. 36, 114, 116
IMAP Internet Message Access Protocol. 37
IMAPS Internet Message Access Protocol Secure. 38
IRC Internet Relay Chat. 5, 28, 47, 52, 56, 62, 66, 67, 72
IV initialization vector. 21, 24, 81, 93, 102, 130, 131
JCA Java Cryptography Architecture. 34, 40, 41, 44, 73
JCE Java Cryptography Extension. 41
JDK Java Development Kit. 41, 134
JRE Java Runtime Environment. 41
KDF key derivation function. 22, 83
KISS Keep it simple, stupid (design principle). 81
MAC message authentication code. 22, 25, 26, 83, 85, 90, 96, 128, 129, 134, 135
MD5 Message-Digest Algorithm 5. 21, 54, 113, 116–118, 129, 142
MSVC Microsoft Visual C++. 20
NI New Instructions. 53, 76
OFB Output Feedback (mode for block ciphers). 24
OS operating system. 19, 20, 133
PBKDF2 Password-Based Key Derivation Function 2. 22, 83
PGP Pretty Good Privacy. 30
PHP PHP: Hypertext Preprocessor (recursive acronym; originally: Personal Home Page Tools). 118, 130, 133
PKCS Public-Key Cryptography Standards. 24, 92, 93
PR pull request. 52, 66–71, 141
PRNG pseudorandom number generator. 20, 31
RFC request for comments. 29, 76, 111
RNG random number generator. 21, 41, 72, 83, 115, 132–134, 138
RQ research question. 38, 45
RSA a cryptosystem named after Ron Rivest, Adi Shamir, and Leonard Adleman. 26, 33, 83, 91
SHA Secure Hash Algorithm. 21, 22, 43, 55, 89, 90, 99, 116, 129, 140, 142
SIMD single instruction, multiple data. 76
SMTP Simple Mail Transfer Protocol. 53
SPI service provider interface. 40, 41
SSL Secure Sockets Layer. 30, 133
TCP Transmission Control Protocol. 97, 98
TLS Transport Layer Security. 12, 26, 31, 37, 40, 52, 54, 102, 105, 117
UTF Unicode Transformation Format. 14, 92, 111
VM virtual machine. 106

References

[AB12] J.-P. Aumasson, D. J. Bernstein. "SipHash: A fast short-input PRF". In: *Progress in Cryptology - INDOCRYPT 2012: 13th International Conference on Cryptology, Kolkata, India, 9-12 December 2012. Proceedings*. Ed. by S. Galbraith, M. Nandi. LNCS 7668. Berlin, Heidelberg: Springer, 2012, pp. 489–508. ISBN: 978-3-642-34931-7. DOI: 10.1007/978-3-642-34931-7_28 (cit. on p. 73).

[ABF+16] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, C. Stransky. "You get where you're looking for: The impact of information sources on code security". In: *37th IEEE Symposium on Security and Privacy (S&P '16), San Jose, CA, USA, 23-25 May 2016. Proceedings*. 2016, pp. 289–305. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.25 (cit. on p. 110).

[ANA+15] S. Arzt, S. Nadi, K. Ali, E. Bodden, S. Erdweg, M. Mezini. "Towards secure integration of cryptographic software". In: *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Pittsburgh, PA, USA, 2015. Proceedings*. Onward! New York: ACM Press, 2015, pp. 1–13. ISBN: 978-1-4503-3688-8. DOI: 10.1145/2814228.2814229 (cit. on pp. 41, 132).

[ANN+16] S. Amani, S. Nadi, H. A. Nguyen, T. N. Nguyen, M. Mezini. "MUBench: A benchmark for API-misuse detectors". In: *13th International Workshop on Mining Software Repositories (MSR '16), Austin, TX, USA, 14-15 May 2016. Proceedings*. New York: ACM Press, 2016, pp. 464–467. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2903506 (cit. on p. 33).

[Arc13] T. Arcieri. *What's wrong with in-browser cryptography?* 2013. URL: https://tonyarcieri.com/whats-wrong-with-webcrypto (visited on 2016-11-07) (cit. on p. 39).

[Arc17] T. Arcieri. *Talk: Macaroons – a better kind of cookie*. 2017. URL: https://air.mozilla.org/rust-meetup-february-2017-02-09/ (visited on 2017-03-16) (cit. on p. 62).

[BB10] J. Bosch, P. Bosch-Sijtsema. "From integration to composition: On the impact of software product lines, global development and ecosystems". In: *Journal of Systems and Software* 83.1 (2010), pp. 67–76. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.051 (cit. on p. 36).

[Ber09] D. J. Bernstein. "Cryptography in NaCl". Chicago, 2009. URL: http://cr.yp.to/highspeed/naclcrypto-20090310.pdf (cit. on pp. 12, 33, 53).

[Bir17] J. Birr-Pixton. *Talk: rustls - a modern, pure-Rust TLS library*. 2017. URL: https://air.mozilla.org/rust-meetup-february-2017-02-09/ (visited on 2017-03-16) (cit. on p. 66).

[BU02] J. Black, H. Urtubia. "Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption". In: *11th USENIX Security Symposium, San Francisco, CA, USA, 5-9 August 2002. Proceedings*. Berkeley: USENIX Association, 2002, pp. 327–338. ISBN: 1-931971-00-5 (cit. on p. 24).

[CGJ16] T. Chełkowski, P. Gloor, D. Jemielniak. "Inequalities in open source software development: Analysis of contributor's commits in Apache Software Foundation projects". In: *PLoS ONE* 11.4 (2016), pp. 1–19. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0152976 (cit. on pp. 28, 64).

[Cua+16] J. Cuadra et al. *Why is a trusted, feature-complete crypto library not a top priority for the Rust community?* 2016. URL: https://internals.rust-lang.org/t/3125 (visited on 2017-03-23) (cit. on pp. 52, 61).

[CWHW12] K. Crowston, K. Wei, J. Howison, A. Wiggins. "Free/Libre open-source software development: What we know and what we don't know". In: *ACM Computing Surveys* 44.2 (2012), pp. 1–35. ISSN: 03600300. DOI: 10.1145/2089125.2089127 (cit. on p. 28).

[Den] F. Denis. *libsodium*. URL: https://github.com/jedisct1/libsodium (cit. on pp. 12, 33, 39, 53).

[Dev14] S. Devlin. *Talk: Crypto interface pitfalls and how to avoid them*. 2014. URL: https://air.mozilla.org/bay-area-rust-meetup-december-2014/ (visited on 2017-01-01) (cit. on pp. 77, 108, 116, 128, 132).

[DK14] S. Das, K. King. "IV = 0 security: Cryptographic misuse of libraries". 2014. URL: https://courses.csail.mit.edu/6.857/2014/files/18-das-gopal-king-venkatraman-IV-equals-zero-security.pdf (cit. on pp. 12, 32, 33, 40, 41, 43, 108, 110–112, 116, 117, 128).

[DR11] T. Duong, J. Rizzo. "Cryptography in the web: The case of cryptographic design flaws in ASP.NET". In: *32nd IEEE Symposium on Security and Privacy (S&P '11), Berkeley, CA, USA, 22-25 May 2011. Proceedings*. IEEE, 2011, pp. 481–489. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.42 (cit. on pp. 42, 115, 116).

[DS08] P. A. David, J. S. Shapiro. "Community-based production of open-source software: What do we know about the developers who participate?" In: *Information Economics and Policy* 20.4 (2008), pp. 364–398. ISSN: 0167-6245. DOI: 10.1016/j.infoecopol.2008.10.001 (cit. on p. 27).

[DW08] A. Dey, S. Weis. *Keyczar: A cryptographic toolkit*. 2008. URL: http://www.keyczar.org (cit. on p. 12).

[EBFK13] M. Egele, D. Brumley, Y. Fratantonio, C. Kruegel. "An empirical study of cryptographic misuse in Android applications". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS '13), Berlin, Germany, 2013. Proceedings*. New York: ACM Press, 2013, pp. 73–84. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516693 (cit. on pp. 11, 31, 32, 41, 108, 128, 132).

[FHM+12] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, B. Freisleben. "Why Eve and Mallory love Android: An analysis of Android SSL (in)security". In: *ACM Conference on Computer and Communications Security (CCS '12), Raleigh, NC, USA, 2012. Proceedings*. New York: ACM Press, 2012, pp. 50–61. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382205 (cit. on pp. 11, 31).

[FHP+13] S. Fahl, M. Harbach, H. Perl, M. Koetter, M. Smith. "Rethinking SSL development in an appified world". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS '13), Berlin, Germany, 2013. Proceedings*. New York: ACM Press, 2013, pp. 49–60. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516655 (cit. on pp. 12, 31).

[FLW12] C. Forler, S. Lucks, J. Wenzel. "Designing the API for a cryptographic library: A misuse-resistant application programming interface". In: *17th Ada-Europe International Conference on Reliable Software Technologies, Stockholm, Sweden, June 11-15, 2012. Proceedings*. Ed. by M. Brorsson, L. M. Pinho. LNCS 7308. Berlin, Heidelberg: Springer, 2012, pp. 75–88. ISBN: 978-3-642-30598-6. DOI: 10.1007/978-3-642-30598-6_6 (cit. on pp. 12, 31, 108, 132).

[FSK10] N. Ferguson, B. Schneier, T. Kohno. *Cryptography engineering: Design principles and practical applications*. Indianapolis: Wiley, 2010. ISBN: 978-0-470-47424-2 (cit. on p. 20).

[GAH13] D. M. German, B. Adams, A. E. Hassan. "The evolution of the R software ecosystem". In: *17th European Conference on Software Maintenance and Reengineering (CSMR '13), Genova, Italy, 5-8 March 2013. Proceedings*. IEEE, 2013, pp. 243–252. ISBN: 978-0-7695-4948-4. DOI: 10.1109/CSMR.2013.33 (cit. on p. 36).

[GGKR02] R. A. Ghosh, R. Glott, B. Krieger, G. Robles. *Free/libre and open source software: Survey and study (FLOSS Final Report Part 4)*. Tech. rep. University of Maastricht, 2002. URL: http://flossproject.merit.unu.edu/report/FLOSS_Final4.pdf (cit. on p. 27).

[GIJ+12] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, V. Shmatikov. "The most dangerous code in the world: Validating SSL certificates in non-browser software". In: *ACM Conference on Computer and Communications Security (CCS '12), Raleigh, NC, USA, 2012. Proceedings*. New York: ACM

Press, 2012, pp. 38–49. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382204 (cit. on pp. 11, 30, 39, 52, 109, 111, 128, 130).

[GS16] M. Green, M. Smith. "Developers are not the enemy! The need for usable security APIs". In: *IEEE Security & Privacy* 14.5 (2016), pp. 40–46. ISSN: 1540-7993. DOI: 10.1109/MSP.2016.111 (cit. on pp. 12, 32, 77, 108).

[GSB16] G. Gousios, M.-A. Storey, A. Bacchelli. "Work practices and challenges in pull-based development: The contributor's perspective". In: *38th International Conference on Software Engineering (ICSE '16), Austin, TX, USA, 14-22 May 2016. Proceedings*. New York: ACM, 2016, pp. 285–296. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884826 (cit. on p. 28).

[Her16a] D. Herman. *Shipping Rust in Firefox*. 2016. URL: https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox/ (visited on 2016-11-08) (cit. on p. 11).

[Her16b] P. Hertleif. *Elegant library APIs in Rust*. 2016. URL: https://scribbles.pascalhertleif.de/elegant-apis-in-rust.html (visited on 2016-03-16) (cit. on pp. 29, 112, 121, 130).

[Hid11] A. Hidayat. *Hall of API shame: Boolean trap*. 2011. URL: https://ariya.io/2011/08/hall-of-api-shame-boolean-trap (visited on 2017-03-23) (cit. on p. 130).

[HNH03] G. Hertel, S. Niedner, S. Herrmann. "Motivation of software developers in open source projects: An Internet-based survey of contributors to the Linux kernel". In: *Research Policy* 32.7 (2003), pp. 1159–1177. ISSN: 0048-7333. DOI: 10.1016/S0048-7333(03)00047-7 (cit. on pp. 27, 60, 61).

[IKND16] S. Indela, M. Kulkarni, K. Nayak, T. Dumitraş. "Helping Johnny encrypt: Toward semantic interfaces for cryptographic frameworks". In: *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Amsterdam, Netherlands, 2-4 November 2016. Proceedings*. Onward! New York, 2016, pp. 180–196. ISBN: 978-1-4503-4076-2. DOI: 10.1145/2986012.2986024 (cit. on p. 77).

[Inf15] Information Assurance Directorate at the NSA. *Commercial National Security Algorithm Suite (CNSA Suite)*. 2015. URL: https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm (visited on 2017-03-14) (cit. on p. 51).

[JC09] K. H. Jamieson, J. N. Cappella. *Echo chamber: Rush Limbaugh and the conservative media establishment*. New York: Oxford University Press, 2009. ISBN: 9780195366822 (cit. on p. 94).

[Kau11] C. Kaula. *The 80% use case*. 2011. URL: https://christiankaula.com/80-percent-use-case.html (visited on 2017-03-23) (cit. on p. 81).

[KG+] P. Kehrer, A. Gaynor et al. *cryptography*. URL: https://cryptography.io/ (cit. on pp. 43, 116).

[KGB+16] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071. ISSN: 1573-7616. DOI: 10.1007/s10664-015-9393-5 (cit. on pp. 28, 67, 72).

[Kob12] M. Koby. *The 80/20 rule*. 2012. URL: https://mkoby.com/the-8020-rule-2/ (visited on 2017-03-23) (cit. on p. 81).

[LCWZ14] D. Lazar, H. Chen, X. Wang, N. Zeldovich. "Why does cryptographic software fail? A case study and open problems". In: *5th Asia-Pacific Workshop on Systems (APSys '14), Beijing, China, 25-26 June 2014. Proceedings*. New York: ACM Press, 2014, 7:1–7:7. ISBN: 978-1-4503-3024-4. DOI: 10.1145/2637166.2637237 (cit. on pp. 11, 12, 31, 32, 108, 115, 132).

[Leb11] J. Lebar. *Boolean parameters to API functions considered harmful*. 2011. URL: http://jlebar.com/2011/12/16/Boolean_parameters_to_API_functions_considered_harmful..html (visited on 2017-03-23) (cit. on p. 130).

[LK12] P. Loyola, I.-Y. Ko. "Biological mutualistic models applied to study open source software development". In: *IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT '12), Macau, China, 4-7 December 2012. Proceedings*. IEEE, 2012, pp. 248–253. ISBN: 978-1-4673-6057-9. DOI: 10.1109/WI-IAT.2012.228 (cit. on p. 36).

[LK14] P. Loyola, I.-Y. Ko. "Population dynamics in open source communities: An ecological approach applied to Github". In: *23rd International Conference on World Wide Web (WWW '14 Companion), Seoul, Korea, 7-11 April 2014. Proceedings*. New York: ACM, 2014, pp. 993–998. ISBN: 978-1-4503-2745-9. DOI: 10.1145/2567948.2578843 (cit. on p. 36).

[LRM14] A. Lima, L. Rossi, M. Musolesi. "Coding together at scale: GitHub as a collaborative social network". In: *8th International Conference on Weblogs and Social Media (ICWSM '14), Ann Arbor, MI, USA, 1-4 June 2014. Proceedings*. AAAI Press, 2014, pp. 295–304. ISBN: 978-1-57735-659-2. arXiv: 1407.2535 (cit. on p. 28).

[LS16] LimeSurvey Project Team, C. Schmitz. *LimeSurvey: An open source survey tool*. Hamburg, Germany, 2016. URL: http://www.limesurvey.org/ (cit. on pp. 56, 142).

[LW03] K. R. Lakhani, R. G. Wolf. "Why hackers do what they do: Understanding motivation and effort in free/open source software projects". 2003. URL: http://www.ssrn.com/abstract=443040 (cit. on p. 27).

[Mar16] R. Martins. *Convenient and idiomatic conversions in Rust*. 2016. URL: https://ricardomartins.cc/2016/08/03/convenient_and_idiomatic_conversions_in_rust (visited on 2017-03-23) (cit. on pp. 29, 109).

[Mat+16] N. Matsakis et al. *Rust roadmap 2017: productivity: learning curve and expressiveness*. 2016. URL: https://internals.rust-lang.org/t/4097 (visited on 2017-03-14) (cit. on p. 11).

[Mer15] L. Mergen. *On the state of cryptography in Haskell*. 2015. URL: https://leonmergen.com/c272fb0b6478 (visited on 2017-03-23) (cit. on p. 39).

[Min16] K. Mindermann. "Are easily usable security libraries possible and how should experts work together to create them?" In: *9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '16), Austin, TX, USA, 16 May 2016. Proceedings*. New York: ACM Press, 2016, pp. 62–63. ISBN: 978-1-4503-4155-4. DOI: 10.1145/2897586.2897610 (cit. on p. 12).

[Moz16] Mozilla. *The Rust programming language*. 2016. URL: https://doc.rust-lang.org/book/ (cit. on pp. 11, 13).

[Nic16] C. Nichols. *Six easy ways to make your crate awesome*. 2016. URL: http://www.integer32.com/2016/12/27/how-to-make-your-crate-awesome.html (visited on 2017-03-16) (cit. on p. 110).

[NIS16] NIST. *Annex A: Approved security functions for FIPS PUB 140-2, Security requirements for cryptographic modules*. 2016. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf (cit. on p. 51).

[NKMB16] S. Nadi, S. Krüger, M. Mezini, E. Bodden. ""Jumping through hoops": Why do Java developers struggle with cryptography APIs?" In: *38th International Conference on Software Engineering (ICSE '16), Austin, TX, USA, 14-22 May 2016. Proceedings*. New York: ACM Press, 2016, pp. 935–946. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884790 (cit. on pp. 12, 32, 41, 96, 108, 115, 116, 138).

[Oec03] P. Oechslin. "Making a faster cryptanalytic time-memory trade-off". In: *Advances in Cryptology - CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, CA, USA, 17-21 August 2003. Proceedings*. LNCS 2729. Springer, 2003, pp. 617–630. ISBN: 978-3-540-40674-7. DOI: 10.1007/978-3-540-45146-4_36 (cit. on p. 22).

[Ora16] Oracle. *Java Cryptography Architecture (JCA) Reference Guide*. 2016. URL: http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Design (visited on 2016-11-25) (cit. on p. 40).

[Par15] Paragon Initiative Enterprises. *Using encryption and authentication correctly (for PHP developers)*. 2015. URL: https://paragonie.com/blog/2015/05/using-encryption-and-authentication-correctly (visited on 2017-03-23) (cit. on p. 24).

[Pil04] D. Piliptchouk. *Java vs. .NET security*. O'Reilly Media, 2004. ISBN: 978-0-596-55668-6 (cit. on p. 42).

[Rar13] K. Rarick. *Fernet spec*. 2013. URL: https://github.com/fernet/spec/blob/0250c59/Spec.md (visited on 2017-03-23) (cit. on p. 128).

[Rob09] M. P. Robillard. "What makes APIs hard to learn? Answers from developers". In: *IEEE Software* 26.6 (2009), pp. 27–34. ISSN: 0740-7459. DOI: 10.1109/MS.2009.193 (cit. on p. 29).

[Rog02] P. Rogaway. "Authenticated-encryption with associated-data". In: *9th ACM Conference on Computer and Communications Security (CCS '02), Washington, DC, USA, 18-22 November 2002. Proceedings*. New York: ACM, 2002, pp. 98–107. ISBN: 1-58113-612-9. DOI: 10.1145/586123.586125 (cit. on p. 24).

[Sap+16] S. Sapin et al. *What should go into the standard library?* 2016. URL: https://internals.rust-lang.org/t/2158 (visited on 2017-03-23) (cit. on pp. 72, 123).

[SBK+17] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov. "The first collision for full SHA-1". 2017. URL: https://shattered.io/static/shattered.pdf (cit. on p. 114).

[Sta06] W. Stallings. *Cryptography and network security: Principles and practices*. Vol. 4301. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006. ISBN: 978-3-540-49462-1. DOI: 10.1007/11935070 (cit. on p. 20).

[Sta16] Stack Overflow. *Developer survey results*. 2016. URL: http://stackoverflow.com/insights/survey/2016 (visited on 2016-03-22) (cit. on pp. 28, 57, 58).

[Sta17] Stack Overflow. *Developer survey results*. 2017. URL: http://stackoverflow.com/insights/survey/2017 (visited on 2017-03-22) (cit. on pp. 11, 28).

[TC14] A. Turon, A. Crichton. *Rust RFC 236: Error conventions*. 2014. URL: https://github.com/rust-lang/rfcs/blob/master/text/0236-error-conventions.md (visited on 2017-03-23) (cit. on p. 109).

[TC15] A. Turon, A. Crichton. *Rust RFC 1242: rust-lang crates*. 2015. URL: https://github.com/rust-lang/rfcs/blob/master/text/1242-rust-lang-crates.md (visited on 2017-03-16) (cit. on p. 72).

[Tro14] J. Troutman. *People want safe communications, not usable cryptography*. 2014. URL: https://www.technologyreview.com/s/533456/ (visited on 2016-11-22) (cit. on p. 30).

[Tul08] J. Tulach. *Practical API design: Confessions of a Java framework architect*. Apress, 2008. ISBN: 978-1-4302-0973-7. DOI: 10.1007/978-1-4302-0974-4 (cit. on pp. 11, 28, 29, 31).

[Wik16] Wikipedia. *Ecosystem*. 2016. URL: https://en.wikipedia.org/w/index.php?title=Ecosystem&oldid=751960746 (visited on 2017-03-23) (cit. on p. 35).

[WO08] G. Wurster, P. C. van Oorschot. "The developer is the enemy". In: *Workshop on New security paradigms (NSPW '08), Lake Tahoe, CA, USA, 22-25*

September 2008. Proceedings. New York: ACM Press, 2008, pp. 89–97. ISBN: 978-1-60558-341-9. DOI: 10.1145/1595676.1595691 (cit. on pp. 29, 32).

[WT99] A. Whitten, J. Tygar. "Why Johnny can't encrypt: A usability evaluation of PGP 5.0". In: *8th Conference on USENIX Security Symposium (SSYM '99), Washington, DC, USA, 23-26 August 1999. Proceedings*. Berkeley: USENIX Association, 1999, pp. 169–183 (cit. on p. 30).

[Wu05] H. Wu. "The misuse of RC4 in Microsoft Word and Excel". 2005. URL: http://ia.cr/2005/007 (cit. on p. 29).

[YHR04] T. Yu, S. Hartman, K. Raeburn. "The perils of unauthenticated encryption: Kerberos version 4". In: *11th Network and Distributed System Security Symposium, (NDSS '04), San Diego, CA, USA, 5-6 February 2004. Proceedings*. The Internet Society, 2004. ISBN: 1-891562-18-5 (cit. on pp. 24, 130).

[YMK+15] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, N. Ubayashi. "Revisiting the applicability of the Pareto principle to core development teams in open source software projects". In: *14th International Workshop on Principles of Software Evolution (IWPSE '15), Bergamo, Italy, 30 August 2015. Proceedings*. New York: ACM, 2015, pp. 46–55. ISBN: 978-1-4503-3816-5. DOI: 10.1145/2804360.2804366 (cit. on p. 28).

[ZP08] M. E. Zurko, A. S. Patrick. "Panel: Usable cryptography: Manifest destiny or oxymoron?" In: *12th International Conference on Financial Cryptography and Data Security (FC '08), Cozumel, Mexico, 28-31 January 2008. Revised Selected Papers*. Ed. by G. Tsudik. LNCS 5143. Berlin, Heidelberg: Springer, 2008, pp. 302–306. ISBN: 978-3-540-85230-8. DOI: 10.1007/978-3-540-85230-8_27 (cit. on p. 30).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature