Herausragende Masterarbeiten

Autor*in

Ingo Budde

Studiengang

Software Engineering for Embedded Systems, M.Eng.

Masterarbeitstitel

Exploring the Use of Static Data Flow Analysis for Automatic Vulnerability Audits of Rust Code

RPTU
Distance and Independent Studies Center
DISC

Rheinland-Pfälzische Technische Universität
Kaiserslautern-Landau

Distance Study Program
Software Engineering for Embedded Systems

Master's Thesis

Exploring the Use of Static Data Flow Analysis for Automatic Vulnerability Audits of Rust Code

---

Provided by
B. Sc. Ingo Budde

First supervisor: Prof. Dr.-Ing. Liggesmeyer
Second supervisor: Prof. Dr. Bodden

Declaration

Ich versichere, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Place, date	Signature
Paderborn, 1.12.2023	Ingo Budde

Abstract

Software systems are ubiquitous in our society and in everyday life. However, bugs make them insecure and vulnerable to attacks. Rust is a novel programming language that uses static code analysis to prevent memory corruption bugs and thread-safety bugs at compile time. This can reduce certain vulnerabilities, but Rust programs are still insecure when using vulnerable external dependencies.

The tool cargo-audit scans Rust projects for external dependencies with known vulnerabilities using the RustSec Database as backend (a Rust-specific vulnerability database). However, cargo-audit does not verify if these vulnerabilities are triggered in the code. Therefore, manual work is still necessary to verify if the code is actually vulnerable. Other tools like MirChecker, Rudra or CRUST perform automated vulnerability audits at the source code level but focus on specific types of bugs and cannot be used to detect all vulnerabilities reported in the RustSec Database.

This thesis introduces a hybrid code analysis tool that queries the RustSec Database to identify vulnerabilities in external dependencies on the project level and then verifies if these vulnerable libraries are used in vulnerable ways. The tool checks if vulnerable functions are actually called and, where applicable, if parameters are actually in the vulnerable range of values. This code analysis tool leverages an algorithm for conditional data-flow analysis, which was developed as part of the thesis. The thesis furthermore shows that extending the RustSec Database to include ranges of vulnerable parameter values for applicable vulnerabilities increases the precision of detecting these vulnerabilities.

The development of the tool is grounded in a set of requirements that were derived from comparing several program representations for Rust code regarding their applicability for data-flow analysis and from studying real-world vulnerabilities with a high reach in the Rust ecosystem. These vulnerabilities were selected based on their frequency in a data set that was produced in the context of this thesis using a structured dependency analysis on all 817.417 package versions published in the Rust Package Registry.

The feasibility of the hybrid approach is demonstrated in the evaluation, which shows that the developed tool works as designed and can be used to find real vulnerabilities in real-world applications in a reasonable time frame. Still, exotic code patterns were identified that result in long analysis times and require future work. Furthermore, many characteristics of the Rust language are currently not supported by the tool, as has been identified by a microbenchmark developed in this thesis to test support for analyzing the Rust language.

Acknowledgements

I would like to thank my advisors, Florian Wege and Jan-Martin Persch, for their invaluable guidance and support throughout this thesis. Special thanks to Prof. Dr.-Ing. Liggesmeyer and Prof. Dr. Bodden for the insightful feedback. I also want to express gratitude to my family for their constant support. Thanks to Fabian Schiebel, Hagen Tarner, Thomas Spengler, Stefan Dziwok, Thorsten Koch, David Schubert and Arno Haase for the helpful feedback. Thanks to the RPTU Kaiserslautern-Landau and Fraunhofer IESE for providing the necessary resources.

Table of Contents

Abbreviations

1. **AST** Abstract Syntax Tree. 13, 14
2. **HIR** High-Level Intermediate Representation. 13, 16
3. **LLVM IR** LLVM Intermediate Representation. 13, 19
4. **MIR** Mid-Level Intermediate Representation. 13, 16, 18
5. **SMIR** Stable Mid-Level Intermediate Representation. 13, 18
6. **THIR** Typed High-Level Intermediate Representation. 13, 16
7. **TOCTOU** Time of Check to Time of Use. 24

List of Tables

List of Figures

List of Listings

1 Introduction

Software systems are ubiquitous in our society and in everyday life. As software systems become more complex, it is increasingly challenging to understand and control their behavior and to ensure their security.
Insecure systems can be attacked and abused for unintentional purposes, threatening individuals, businesses, and society as a whole. For example, the global economic impact of cybercrime was estimated to be greater than US $400 billion in annual costs (McAfee & CSIS, 2014).
For software-intensive systems, it is therefore of the utmost importance to include security-related considerations throughout the software development lifecycle to create systems that are "secure by design" (Bodden, 2018b).
Static code analysis, a major component of software analysis, is a method of reasoning about program code without the need for actual program execution. Static code analysis can detect potential problems early in the software development lifecycle, improving code quality, reducing vulnerabilities, and increasing robustness.
As most software systems use many external dependencies, any bugs in those external dependencies directly influence the security of the entire system. Therefore, external dependencies are highly relevant in the context of software security. The OWASP Top 10 List (The OWASP Foundation, 2021) represents industry consensus on critical security risks and lists "Vulnerable and Outdated Components" in the sixth position.

1.1 Rust – A New, Safer Language

The Rust programming language has experienced massive adoption during the past 10 years and addresses problems that native languages usually have with a unique approach. It provides a strong type system that helps to prevent memory and thread safety issues at the time of compilation. This is achieved by enforcing a strict ownership model through static analysis performed in the compiler. In this way, the code is rejected by the compiler if it contains such bugs. The runtime performance of Rust code is comparable to that of C++, making it a viable choice even for embedded software development.
Still, Rust programs can be vulnerable, especially when relying on vulnerable external dependencies. Rust offers a unified approach to managing software dependencies using the official Rust Package Registry[1] and allows users to easily integrate external dependencies through the Package Manager Cargo. Cargo enforces a project layout, which increases compatibility between projects. This has resulted in Rust projects relying heavily on external dependencies. Kikas, Gousios, Dumas, and Pfahl (2017) reported an average

[1] https://crates.io

of 9.6 transitive dependencies per project. Therefore, it can be concluded that Rust projects often have many external dependencies, making the problem of using vulnerable or outdated dependencies highly relevant. Moreover, dealing with extensive dependency trees makes it difficult to manually assess security vulnerabilities.

1.2 Automated Vulnerability Audits in Rust

To address the issue of vulnerable dependencies in Rust at scale, automated vulnerability audits are needed. The RustSec Database[2] provides information on vulnerable packages in the Rust Package Registry, and is used by the static analysis tool *cargo-audit* to detect if any external dependencies are listed as vulnerable using dependency analysis. However, this approach identifies complete software projects as vulnerable, as opposed to individual sections of the project's source code. Furthermore, it is not detected if vulnerable parts of the code are ever executed. Rejecting vulnerable packages altogether can lead to the use of outdated versions, which themselves might cause other security risks and provide limited functionality. In addition, security experts may need to manually assess whether the software system is actually vulnerable at the code level, which can be costly.
Automated vulnerability audits at the code level are beneficial in reducing this manual verification work. Previous research has employed data-flow analysis and constraint solving techniques; however, these have been limited to detecting only specific kinds of vulnerabilities. For example, Rudra (Bae, Kim, Askar, Lim, & Kim, 2021) has used taint analysis to detect memory safety bugs, undefined behavior, and security threats in Rust code. MirChecker (Li, Wang, Sun, & Lui, 2021) has used Abstract Interpretation to identify denial-of-service attacks and memory safety bugs in Rust code. CRUST (Toman, Pernsteiner, & Torlak, 2015) uses the model checker CBMC to identify memory safety bugs. The advantage of these approaches is that they can uncover previously unreported vulnerabilities. The drawback is that they only detect certain types of vulnerabilities that are explicitly supported. Also, most related approaches do not monitor data flows into all external dependencies in the dependency tree, which is a problem as the exploitability is increased by the many external dependencies used. For example, the library `time`, which provides access to the system time, is one of the most transitively dependent libraries in the Rust ecosystem (Kikas et al., 2017) and is vulnerable to `RUSTSEC-2020-0071`[3].
In conclusion, automated vulnerability audits are necessary to identify security problems at scale, but existing tools do not solve the problem of vulnerable dependencies completely. They either do not utilize the RustSec Database and thereby only focus on certain vulnerability types, or they utilize the database allowing them to identify any

[2] https://rustsec.org
[3] https://rustsec.org/advisories/RUSTSEC-2020-0071.html

reported vulnerability but only operate on the project level and do not verify if the code actually triggers the vulnerability causing additional manual effort.

1.3 Hybrid Approach: Dependency Analysis and Subsequent Data-Flow Analysis

This thesis proposes a hybrid approach as shown in Figure 1-1 that performs both a dependency analysis and a data-flow analysis to automatically audit the source code to identify if the vulnerable parts of the code are ever used.
The approach of cargo-audit is reused on a conceptual level to perform the dependency analysis by querying the RustSec Database for vulnerable dependencies. This part is shown on the left side of Figure 1-1.
When vulnerable dependencies are found, a subsequent data-flow analysis is performed to identify whether the vulnerability can be confirmed at the code level, thereby increasing the precision of the automated vulnerability audit. This part is shown on the right side of Figure 1-1.

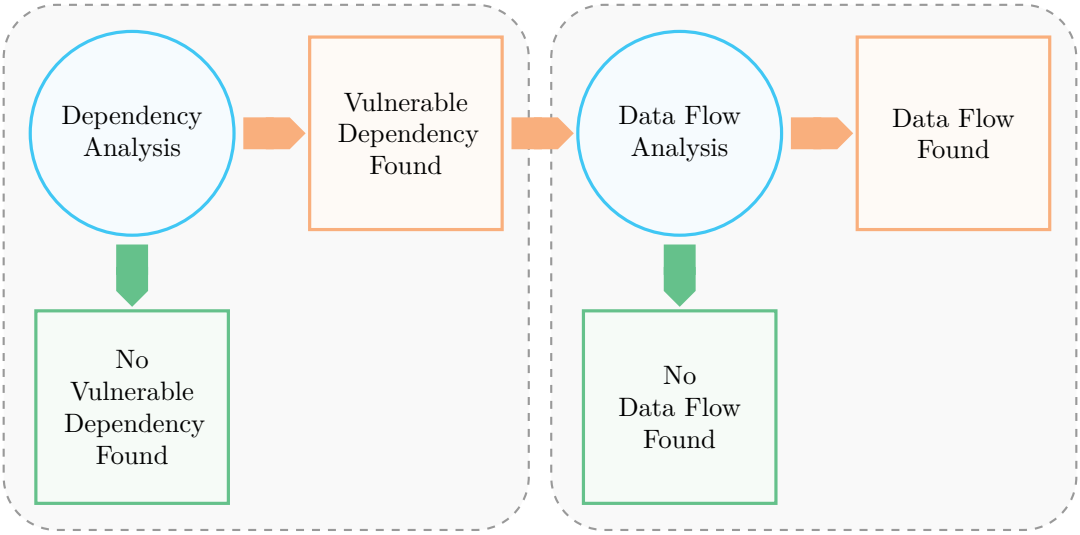


Figure 1-1: Hybrid Approach Combining Dependency Analysis With Subsequent Data-Flow Analysis

1.4 Research Questions

The following questions guide through this thesis and drove the research behind it:

RQ1 How can the RustSec Database be utilized to support data-flow analysis?

RQ2 How can data-flows across external dependencies be analyzed?

RQ3 What program representations of Rust code are suitable for data-flow analysis?

Research Question *RQ1* *(How can the RustSec Database be utilized to support data-flow analysis)* is defined. The RustSec Database is an established repository of known security vulnerabilities and forms the back-end of the tool cargo-audit. The vulnerability definitions in this database have the granularity of individual Rust crates. However, as data-flow analysis relies on more fine-grained information, the applicability of this database for data-flow analysis needs to be explored.
Research Question *RQ2* *(How can data-flows across external dependencies be analyzed)* is defined. In the context of this thesis, a data-flow analysis is proposed to analyze the Rust code including their external dependencies. It has to be identified if the data-flow analysis can be extended to multiple projects and in particular how data-flows at the boundary between two dependencies can be maintained.
Research Question *RQ3* *(What program representations of Rust code are suitable for data-flow analysis)* is defined. In the Rust ecosystem, specific program representations are employed to represent the Rust code. These representations are used to perform analyses as well as to compile the Rust program. Therefore, the applicability of these representations for data-flow analysis needs to be explored.

1.5 Contributions of This Thesis

This thesis introduces a hybrid code analysis tool that queries the RustSec Database to identify vulnerabilities in external dependencies on the project level and then verifies if these vulnerable libraries are used in vulnerable ways. The tool checks if vulnerable functions are actually called and, where applicable, if parameters are actually in the vulnerable range of values.
The thesis contributes the underlying algorithm for conditional data-flow analysis that evaluates if conditions to increase precision.
The thesis furthermore contributes an extension to the RustSec Database schema to include ranges of vulnerable parameter values for applicable vulnerabilities and extends the respective vulnerability entries in the database to include the corresponding data.
The thesis contributes a comparison of several program representations that can be used to perform static analysis on Rust code.

Furthermore, the thesis contributes a data set that describes the frequency of vulnerabilities in the Rust ecosystem. This data set is produced using a structured dependency analysis on all 817.417 package versions published in the Rust Package Registry.
The evaluation demonstrates that the developed tool works as designed and can be used to find real vulnerabilities in real-world applications in a reasonable time frame, but for exotic code patterns analysis is currently slow and the coverage of the Rust language characteristics is currently low.

1.6 Thesis Structure

This thesis is organized as follows. Chapter 2 introduces the foundational concepts used in this thesis, including used terms for describing the static code analysis, the Rust programming language, the Rust ecosystem, and prevalent program representations of Rust code. Chapter 3 outlines the requirements for the developed static analysis tool. This chapter examines the particularities of significant vulnerabilities in the Rust environment and chooses a program representation to work with. Chapter 4 conceptualizes a high-level architecture for the tool based on the defined requirements and also describes the design of the tool by presenting the used data-structures and algorithms. Chapter 5 outlines the implementation of the tool in the Rust language. The implementation interfaces with existing components, such as the Rust compiler and Cargo, the Rust build tool. The implementation also uses optimized data-structures to increase run-time performance and decrease memory usage. Chapter 6 evaluates the tool on both synthetic benchmarks and real-world programs and discusses the quality of the findings and the measured run-time performance. Chapter 7 compares the approach of this thesis with related approaches in the field, including a highly related approach using abstract interpretation. Chapter 8 concludes the thesis, provides answers to the research questions, and highlights key insights, as well as possible further research.

2 Background

This chapter introduces the background concepts and the theory used in this thesis. Section 2.1 covers fundamental concepts of static analysis. Section 2.2 highlights the distinctive features of the Rust programming language. Section 2.3 describes the Package Manager Cargo and the Rust Package Registry. Section 2.4 introduces the RustSec Database[1], which is a Rust-specific vulnerability database. Section 2.5 introduces and analyzes program representations of Rust code that are prevalent in the Rust ecosystem and applicable for Rust code analysis.

2.1 Static Code Analysis

Static code analysis (Møller & Schwartzbach, 2018) is a technique used in software security to examine the source code of a program without executing it. It can be used to identify potential vulnerabilities, bugs, and security issues by analyzing the code structure, control flow, and data-flow. In static code analysis, *entry points* refer to specific locations in a program where the analysis starts. *Field sensitivity* in static code analysis describes the tracking of individual fields or data members within data structures and allows the analysis to capture how data is manipulated within these structures, enhancing the precision of the analysis. *Alias sensitivity* aims to differentiate between different pointers, references, or variables and determine whether they may point to the same or distinct memory locations. In security analysis, understanding aliasing relationships helps uncover potential vulnerabilities that originate from unintended data sharing or manipulation.

2.1.1 Intraprocedural Analysis

Intraprocedural analysis is a type of static code analysis that focuses exclusively on the code within an individual function and is used to understand its behavior and properties in isolation. It usually involves analyzing control flow, data-flow, and local variables within that function. For analyzing the control flow of an application and understanding control flow relationships, the subject of analysis can be represented as a directed graph (Allen, 1970).

A control flow graph is a representation of a program's control flow. It uses nodes to represent basic blocks of code and edges to represent how control flows between these blocks. In security analysis, it helps identify potential vulnerabilities related to branching and program execution flow.

[1] https://rustsec.org

2.1.2 Data-Flow Analysis and Abstract Interpretation

Data-flow analysis (Møller & Schwartzbach, 2018) is a technique used to track how data propagates through a program by examining how data is read, modified, and passed between variables and functions through the application of flow functions. A *flow function* defines how a value is transformed into another value when evaluating an individual statement that is part of the analyzed program. *Constant propagation* is a specialized form of data-flow analysis that focuses on constant values of variables within a program by propagating known values at each statement in the program and reasoning how a statement reads and updates these variable values.
Due to Rice's theorem (Rice, 1953), which states that any non-trivial semantic property of the program is undecidable, it is not feasible to create a general algorithm that can accurately reason about all potential states or properties of a program at run-time.
To cope with this fact, the analysis can be performed on *abstract values* representing multiple states at once to reduce the problem space for the analysis. This way of analyzing the program is called Abstract Interpretation (Cousot & Cousot, 1977).

2.1.3 Lattice Theory

Lattice Theory is a mathematical model grounded in order theory. In the context of program analysis, lattices are used to describe the set of possible *abstract values* that a variable can have at any given point in the program. The lattice can be defined for a specific analysis, and the elements and operations performed on the lattice depend on the concrete analysis performed.

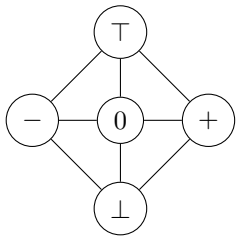


Fig. 2-1: Lattice Used for Sign Analysis

Figure 2-1 shows an exemplary lattice that can be used to represent the sign of a numeric variable, which can be negative, zero, or positive. As multiple numeric values are represented by one lattice element, the lattice describes abstract values. In this example, the element $\bot$ (bottom) represents the initial state before a variable has a known value and the sign of the value is still unknown. The lattice element `+` represents positive numbers, and the element `-` represents negative values. The element $\top$ (top) represents the state that occurs if during analysis multiple signs are possible for one single variable, which can happen when considering multiple paths that the program can take.
The following set of definitions are based on Møller and Schwartzbach (2018) and Kam and Ullman (1977).

Definition 1 (Join Semilattice)**.** A join semilattice is a set L with two partial orders $\sqsubseteq$, $\sqsubset$ and a join operation $\sqcup$ such that:

1. Regarding $\sqsubseteq$ it holds that: $\forall x, y \in L : x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.
2. Regarding $\sqsubset$ it holds that: $\forall x, y \in L : x \sqsubset y \Leftrightarrow x \sqcup y = y \wedge x \neq y$.

Definition 2 (Bounded Chain)**.** A chain $x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n$ of lattice elements is bounded if for each x in the chain, there is a constant b_x, such that each chain beginning with x has length at most b_x.

Definition 3 (Bounded Semilattice)**.** A bounded semilattice L is a semilattice, where:

1. The element $\top \in L$ (top) is the unique maximum element with respect to $\sqsubseteq$.
2. The element $\bot \in L$ (bottom) is the unique minimum element with respect to $\sqsubseteq$.
3. Every chain in the lattice is bounded.

2.1.4 Monotone Framework

The Monotone Framework (Kam & Ullman, 1977) is based on lattice theory and ensures that as information flows through the analysis, it never decreases, providing a foundation for maintaining consistent results and for proving correctness and termination.

Definition 4. A Monotone data-flow analysis framework is a triple $D = (L, \sqcup, F)$, where

1. L is a bounded semilattice with join operation $\sqcup$.
2. F are monotone functions that operate on elements of L.

Definition 5. A function $f \in F$ is monotone if: $\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Fixed-point computation is a method used in data-flow analysis which iteratively refines the results until a stable solution is achieved. For monotone functions, this fixed-point exists.
Therefore, to guarantee a sound data-flow analysis, the analysis described in this thesis will be designed to perform a fixed-point computation of monotone functions as required by the Monotone Framework.

2.1.5 Interprocedural Analysis

Interprocedural analysis (Møller & Schwartzbach, 2018) extends the analysis beyond the boundaries of individual functions. It considers how functions or procedures interact with one another, including how data is passed between them and how control flows between function calls to provide a more comprehensive view of a program's behavior by considering the effects of function calls on data and control flow. This is important because security issues can span multiple functions. For interprocedural analysis, understanding the interaction of function calls can be aided by a directed, acyclic graph (Ryder, 1979). A *call graph* is a representation that encodes functions as nodes and function calls as edges.

2.1.6 Context Sensitivity

Context Sensitivity (Møller & Schwartzbach, 2018) in static code analysis determines how the analysis considers the calling context of a function, which is the sequence of function calls and their parameters that lead to the execution of a specific code segment. A context-sensitive analysis is used to capture more precise and detailed information about the program behavior by considering the specific context the functions are called in. In this thesis, the focus is put on context-sensitive, interprocedural static analysis, as they are by their very nature more suitable for the here conducted work.
Sharir and Pnueli (1978) describe two approaches to achieve a context-sensitive analysis: The *call-strings approach* and the *functional approach*. The call-strings approach keeps track of the call stack, enabling precise tracking of the function call history. In contrast, the functional approach works by creating reusable summaries that abstract function behavior and focus on the essence of what functions do. When handling a function call and a reusable summary already exists for that function, the existing summary is applied, and the function is not analyzed another time.
The choice between these approaches affects the precision and run-time of the analysis. Bodden (2018a) mentions that the call-strings approach can lead to an exponential blow-up in the number of contexts that the analysis handles, which may make it impractical for realistic programs as it might require a massive amount of resources. In the context of this thesis, the focus will therefore be on the functional approach to achieve context sensitivity.
Function summaries can be created using either a top-down summarization approach or a bottom-up summarization approach. The top-down approach propagates information from function callers to callees, while the bottom-up approach propagates information from the callees to the callers.

Zhang, Mangal, Naik, and Yang (2014) describe that "top-down analyses only analyze procedures under contexts in which they are called in a program", leading to summarizes that track details to individual calling contexts, while "bottom-up analyses analyze procedures under all contexts", making them highly reusable and easier to parallelize. In the context of this thesis, the focus will be on the bottom-up summarization approach, as it allows to analyze external dependencies compositionally and comprehensively before analyzing their dependents.

2.2 Rust Programming Language

Rust is a strongly-typed programming language with a strong emphasis on safety, performance, and concurrency. It offers a set of features that are enforced by the Rust compiler and collectively contribute to preventing common classes of program errors, such as memory corruption bugs and thread safety violations. The Rust Book[2] mentions that Rust enforces an ownership model that requires each value to have one single and unique owner. When the owner is deallocated, the owned value is deallocated as well. This ownership model is more effective than manual memory management in avoiding heap corruption, double frees, and undefined behavior. Rust also tracks the lifetime of every value and reference to a value, ensuring that every value lives at least as long as any reference to it. Additionally, Rust does not allow the use of raw pointers or pointer arithmetic by default to prevent similar classes of bugs. Furthermore, Rust differentiates between mutable and immutable references, preventing two mutable references to one value from existing at the same time. This prevents race conditions in multithreaded code and allows performance optimizations. Finally, Rust tracks which values can be safely transferred into another thread and which values are safe to be referenced by another thread by inductively reasoning about the type of the value.

The Rust language rules, as mentioned in the preceding section, are very stringent, and thus it is possible that the compiler enforcing those regulations may reject a program that does not actually contain any memory-related or thread-safety bugs. This also leads to the situation that programs might not be expressable in the Rust language at all. Rust provides a language superset, referred to as unsafe code. This unsafe code can be mixed with regular Rust code and allows for manual memory management using raw pointers, as well as bypassing the lifetime checks, mutable aliasing checks, and thread-safety checks of the compiler.

Bae et al. (2021) studied the consequences of using unsafe code at the ecosystem level. Rust code must adhere to the ownership rules, and for safe Rust, the compiler checks those rules. Unsafe code can escape compiler checks, but if unsafe code violates these

[2] https://doc.rust-lang.org/book/

rules, undefined behavior can occur, which is the source of many reported vulnerabilities. When an application only uses safe Rust, the chances of introducing vulnerabilities related to memory management or thread management are significantly reduced. However, any external dependencies used by the application may contain unsafe code, which in turn can contribute undefined behavior that affects the complete application.

2.3 Rust Crates and Cargo

Rust provides encapsulation of the code by using so-called *crates*. These crates are either applications or libraries. An application crate contains a function `main`, which is the entry point for the program and is usually defined in the file `main.rs` itself or in a submodule of it. A library crate provides common functionality that other crates can import. The exported items of a library crate have public visibility and are usually defined in the file `lib.rs` or in a submodule of it.
Rust allows composing multiple crates into so-called *packages*. In Rust, a package includes metadata such as package name, version, and any dependencies with a valid version range. The Rust language has an official package manager, Cargo, which simplifies the creation and compilation of Rust packages. Package metadata is defined in the configuration file `Cargo.toml`. A lock file, `Cargo.lock`, is used to ensure reproducible builds for multiple developers. Cargo is integrated with the Rust Package Registry[3] to allow the downloading and publishing of Rust packages.

2.4 RustSec Database

The RustSec Database is a collection of vulnerabilities that affect the Rust ecosystem. It is managed as a git repository with markdown files that provide textual descriptions of each vulnerability. Each markdown file contains a section that adheres to the TOML syntax (Preston-Werner, Gedam, et al., 2019) and provides structured information about the vulnerability. The structure of this section is shown in Figure 2-2. A vulnerability record is called an `Advisory` and includes details of the affected functions, architectures, and operating systems. It also outlines the range of affected versions of the crate. Rust follows the Semantic Versioning Standard (Preston-Werner, 2013).
As an example, Listing 2-1 shows the definition for vulnerability `RUSTSEC-2023-0044`.
The RustSec Database was accessed on June 22, 2023 (git revision `9cf72357c8`) and contained 553 vulnerabilities at that time. These 553 entries can be subdivided into 533 entries for crates published in the official Rust Package Registry, 18 vulnerabilities in the standard library, one vulnerability in the Rust Package Manager Cargo and one in the

[3] https://crates.io

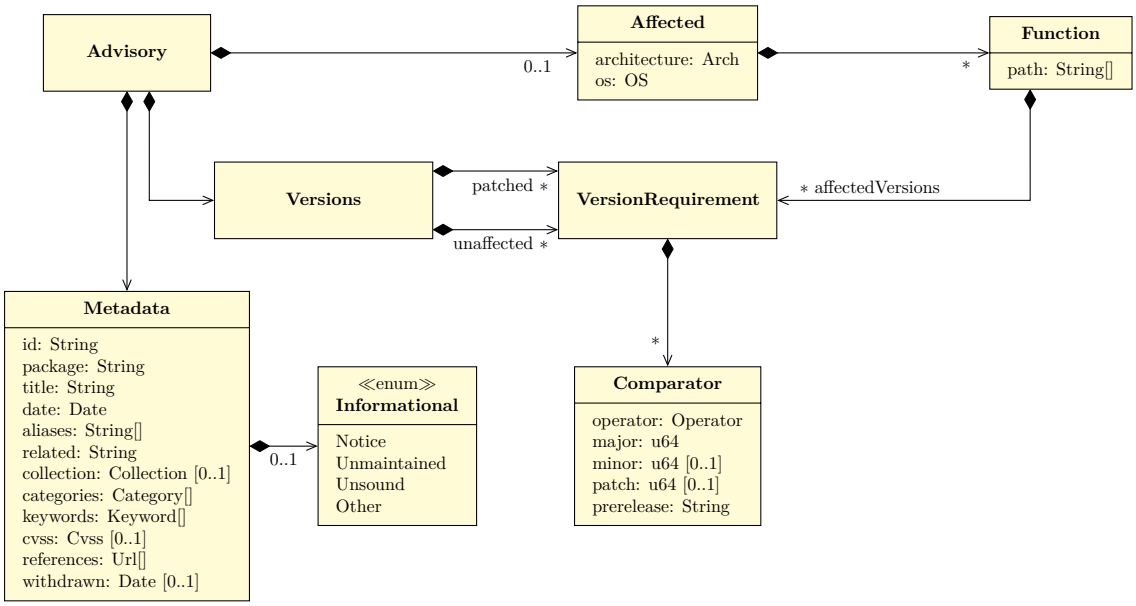


Figure 2-2: RustSec Database Schema

```
[advisory]
id = "RUSTSEC-2023-0044"
package = "openssl"
date = "2023-06-20"
url = "https://github.com/sfackler/rust-openssl/issues/1965"
categories = ["memory-exposure"]
aliases = ["GHSA-xcf7-rvmh-g6q4"]

[affected]
functions = { "openssl::x509::verify::X509VerifyParamRef::set_host" = ["< 0.10.55,
    ↪ >=0.10.0"] }

[versions]
patched = [">= 0.10.55"]
```

Listing 2-1: Vulnerability Definition RUSTSEC-2023-0044

automatic documentation generator Rustdoc.

The 533 vulnerability entries related to published crates can be further subdivided into 104 informational entries, which either inform about an unmaintained package (101 cases) or report a package that only exists to present a vulnerability (3 cases). The remaining 429 entries refer to actual, code-related vulnerabilities in published crates.

Vulnerability entries can also refer to the part of the source code, which is affected by the vulnerability by defining *affected functions*, which represent the functions in the source code that contain the vulnerability. This can be helpful information for the data-flow analysis that is designed in this thesis because it can be used to configure the analysis to identify the invocation of such affected functions.

In the RustSec Database, 101 vulnerability entries of the 429 relevant entries define the affected functions. Therefore, this thesis focuses on vulnerabilities for which the affected functions are known.

2.5 Rust Program Representations

The Rust Compiler Development Guide (The Rust Project Developers, 2018) mentions several program representations that are used within the Rust compiler and are shown in Figure 2-3. These representations model the state of the source code as it is being compiled. The compilation process is subdivided into multiple transformations. After each major transformation step, the code is represented in a respective Intermediate Representation (IR) that explicitly models the properties and invariants that hold at the respective compilation stage.

The compiler accepts textual source code and then successively transforms it into more low-level representations until it eventually emits machine code. The first transformation parses the textual source code into an AST (Abstract Syntax Tree), which is then successively lowered into specialized representations called HIR (High-Level Intermediate Representation), THIR (Typed High-Level Intermediate Representation), MIR (Mid-Level Intermediate Representation) and LLVM IR (LLVM Intermediate Representation). The SMIR (Stable Mid-Level Intermediate Representation) is a proposed representation, which is not usable in practice currently.

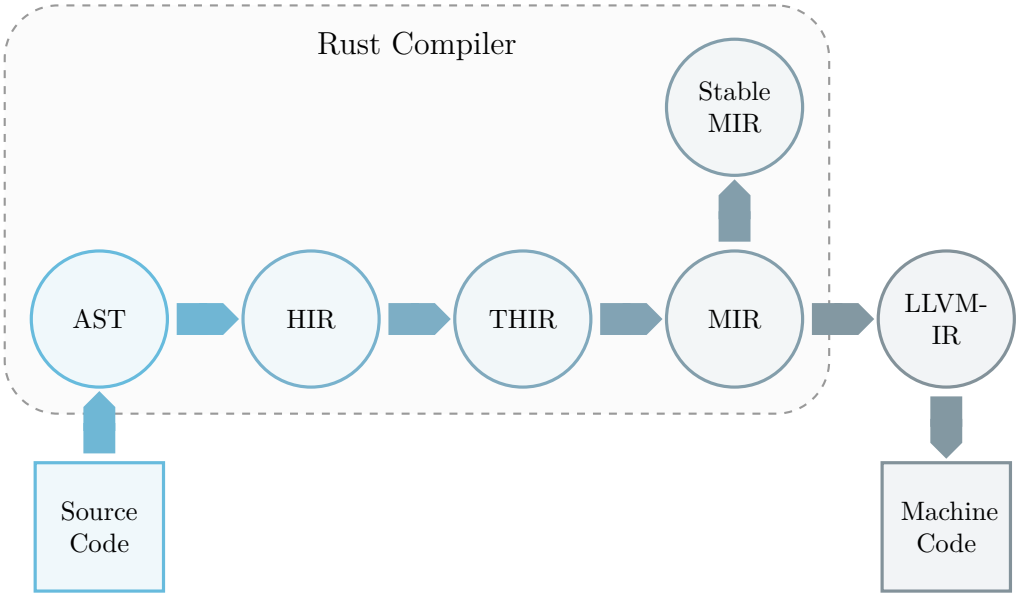


Figure 2-3: Rust Compilation Process and Intermediate Representations

2.5.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is one-to-one representation of the textual source code, which is obtained by parsing the textual representation into a tree form describing the syntax elements that are part of the Rust source code. Figure 2-4 shows an excerpt of the AST representation that has been deduced from the Rust compiler source code.[4]
The AST distinguishes between items, expressions and statements. Items as represented by the type `Item` denote structural language items, like the definition of a type, a module or a function while expressions and statements appear within the context of a function's body. Statements, as represented by the type `Stmt` and represent an instruction that does not necessarily provide a return value. Expressions, as represented by the type `Expr`, inherently represent a value or the calculation of a value and can be nested to form an expression tree that can be evaluated to retrieve the calculated expression value.
The Rust language provides advanced concepts like Macros[5] and Pattern Matching[6]. Macro definitions are represented in the AST via the item kind `MacroDef` and a macro call is represented by the expression kind `MacCall` as well as the statement kind `MacCallStmt`. The Rust keyword `match`, which enables pattern matching, is represented by the expression kind `Match`. Rust supports three kinds of loops: The for loop, while loop and the unconditional loop, which are represented by the expression types `ForLoop`, `While` and `Loop`. In the AST, name lookup is not yet performed and thus elements reference each other only via their name.

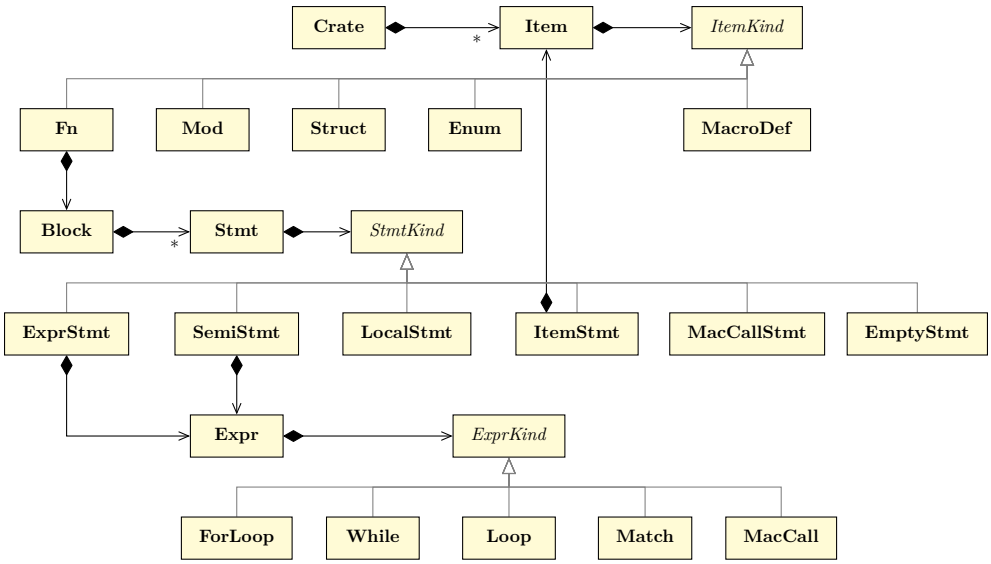


Figure 2-4: Excerpt of Abstract Syntax Tree

[4] https://github.com/rust-lang/rust/tree/1.70.0/compiler/rustc_ast
[5] https://doc.rust-lang.org/book/ch19-06-macros.html
[6] https://doc.rust-lang.org/book/ch18-00-patterns.html

2.5.2 High-Level IR

The HIR is created by transforming the AST into a format that is better qualified for the algorithmic analyses and transformations performed in the Rust compiler. Figure 2-5 shows an excerpt of the internal structure of the HIR that has been deduced from the Rust compiler source code.[7]

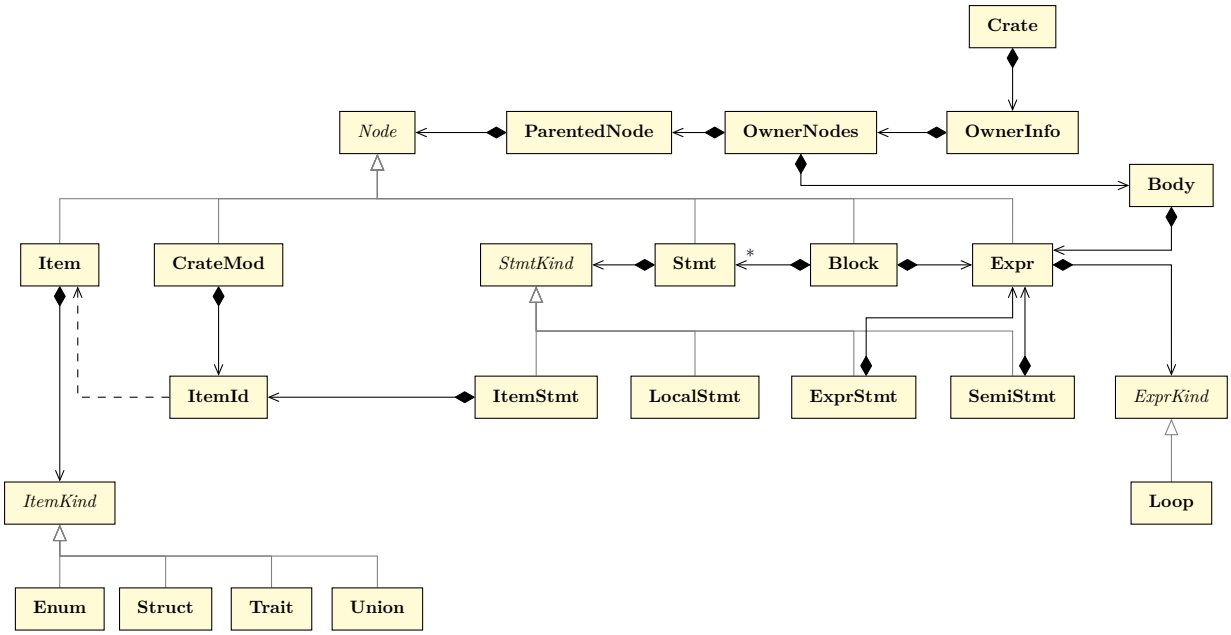


Figure 2-5: Excerpt of the High-Level Intermediate Representation

In the HIR, many of the advanced language features that Rust provides are transformed into a more unified form. Similarly to the AST, the HIR also contains items that represent structural language items (e.g. enumerations, structs, traits, unions, among others). While these items refer to each other by name in the AST, on the HIR level these references are resolved, and therefore these data structures are traversable.

The HIR introduces the concept of so-called *body owners*, which represent items that contain code. Functions and constant initializers are both body owners. The code that is contained in the body owners consists of statements and expressions.

In HIR, macros are expanded by inlining the generated macro code at the call-site of the macro, and the different loop constructs part of the Rust Language provides are unified as a single loop element `Loop`, which contains the exit conditions necessary to emulate the other loop types as part of the loop body.

In Rust, items can be re-exported to other modules using the keyword `pub use`[8]. The information about these re-exports is available in the HIR and is relevant for the inter-crate

[7] https://github.com/rust-lang/rust/tree/1.70.0/compiler/rustc_hir
[8] https://doc.rust-lang.org/beta/std/keyword.pub.html

analysis that is explored in relation to the Research Question *RQ2* (How can data-flows across external dependencies be analyzed).
In summary, the HIR represents structural parts of the code as so-called items, the references between items are resolved, macros are expanded, re-exports are known and the concept of body owners is introduced that represents elements containing code and are identified by an ID, which allows subsequent IRs to refine the code of these body owners.

2.5.3 Typed High-Level IR

The THIR[9] is generated from HIR after type checking. Unlike the HIR, the THIR provides a modified version of the code a with filled-in types after the completion of type checking. It exclusively represents executable code bodies, such as function bodies and const initializers, excluding items like structs or traits.
The THIR is used to generate MIR. In particular, THIR bodies are temporary and discarded when no longer needed, contrasting with the HIR, which persists until the compilation process concludes. Additionally, the THIR incorporates desugaring beyond type information, explicitly representing automatic references and dereferences. Rust allows the program to omit the reference and dereference operators when enough context information is known to add implicit reference and dereference operations as necessary. THIR makes this explicit by adding the dereference expressions. THIR converts method calls and overloaded operators into regular function calls and makes destruction scopes explicit. In summary, THIR refines the code as represented by body owners by adding type information and preparing the IR to generate MIR.

2.5.4 Mid-Level IR

The MIR was introduced to the Rust language in the Rust RFC1211 (Matsakis, 2015) and refines the function bodies for the body owners as first introduced in HIR and later enriched by type info from THIR. The MIR provides a control flow graph (CFG) to make the control flow explicit. Figure 2-6 shows an excerpt from MIR that was deduced from the Rust compiler source code.[10] In MIR, a basic block contains arbitrarily many statements followed by a single terminator statement. This terminator statement is located at the end of the basic block and describes how to terminate the basic block (e.g. by defining the successor basic block). In this way, MIR models the control flow of the function explicitly. The assignment statement as represented by the type `Assign` assigns an Rvalue to a place. A place describes an access path to a variable and is used on the left-hand side of an

[9] https://rustc-dev-guide.rust-lang.org/thir.html
[10] https://github.com/rust-lang/rust/blob/1.70.0/compiler/rustc_middle/src/mir

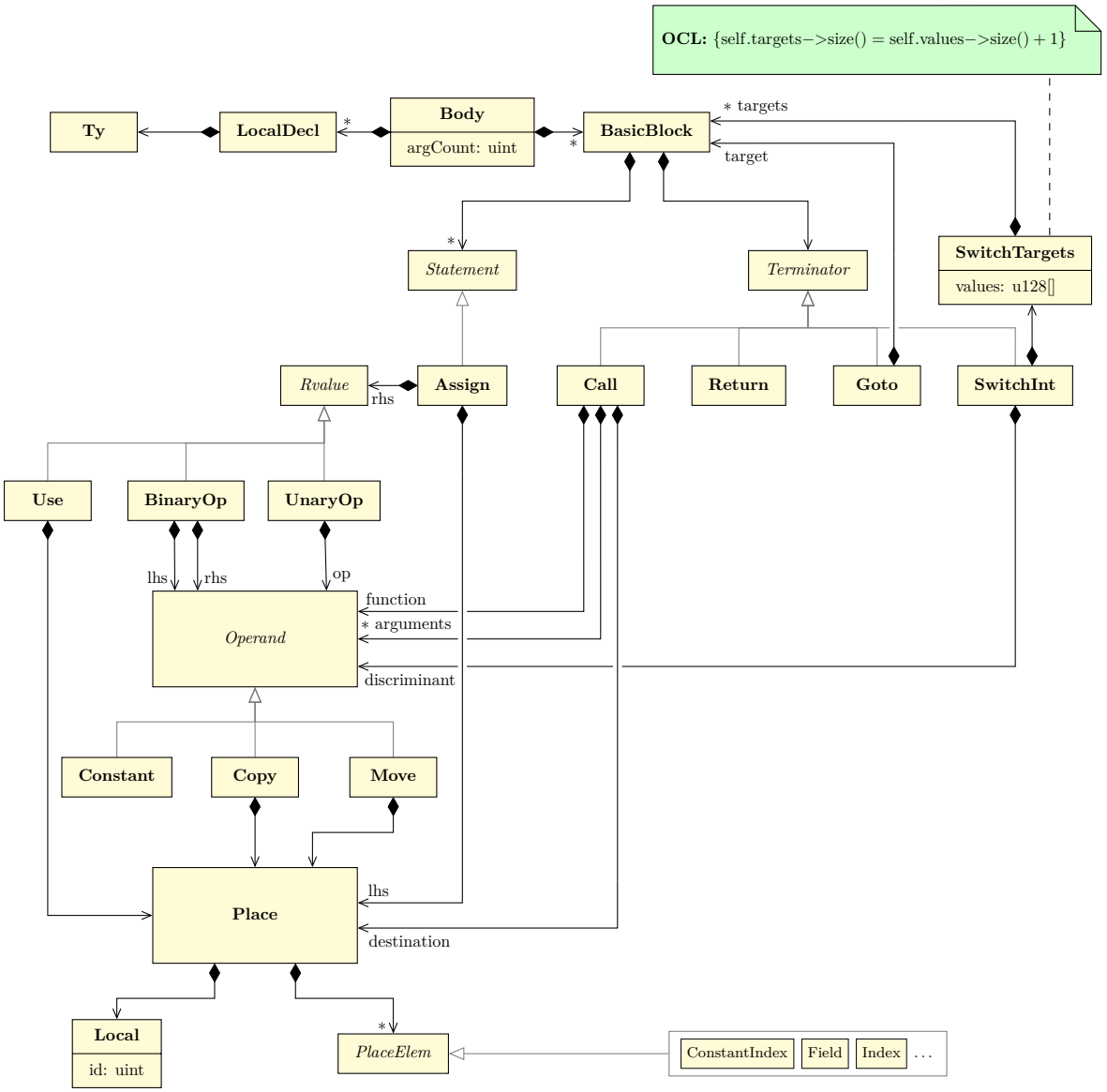


Figure 2-6: Excerpt of the Mid Level Intermediate Representation

assignment. The access path starts at a local variable and furthermore consists of several segments, called `PlaceElem`. One such segment is the `Field`, which describes a field access using the dot operator, and the `ConstantIndex` and `Index` segments, which represent an array access using square bracket notation. Other segments are the `Deref` operator, which dereferences a pointer using the $*$ operator, as well as several operators for casting and accessing subslices. The right-hand side of an assignment is represented by the type `Rvalue` and represents either a used place, a binary expression, or a unary expression. The binary expression refers to two operands, a left-hand side and right-hand side, while the unary expression refers to a single operand. The statement `SwitchInt` compares a discriminant operand to multiple constant values of type `u128` and performs a jump to a target basic block depending on which number is equal to the operand. An additional

target represents the default target to jump to if no value matches.
A data-flow analysis needs access to the control flow graph, and the MIR explicitly models the control flow of the program, which makes it a relevant candidate for the use in this thesis.

2.5.5 Stable Mid-Level IR

The SMIR is proposed by the Stable MIR Librarification Project Group (2023) to provide a stable API for external consumption by clients like static analysis tools. This is relevant because in general, the IRs in the compiler are not meant to be used outside of the Rust compiler and therefore do not define a stable API and no serialization format. The Rust internal IRs are tightly coupled to the Rust compiler, change frequently and are therefore highly unstable and are generally considered an implementation detail of the Rust compiler. When using the internal data structures of the Rust compiler as a library in a separate application, the application can only be executed if a compatible version of the Rust toolchain is installed. This forces users of such an application to install a compatible Rust toolchain on their system, which might not be the most recent version of Rust, if applications are not kept up-to-date with the latest changes in Rust. While it is possible to install multiple Rust toolchains on the same system, relying on old software versions is generally not advisable.
SMIR is a new IR that is still in the draft phase. The idea of SMIR is that it can solve this problem by proposing a stable serialization format for MIR that can be used safely by external analysis tools without losing compatibility to other Rust toolchain versions. However, as SMIR is still in the draft phase, it cannot be used in this thesis.

2.5.6 LLVM IR

The LLVM IR is part of the LLVM compiler framework published by Lattner and Adve (2004). The Rust compiler uses the LLVM framework to generate efficient machine code for a multitude of platforms. The architecture of the LLVM Framework is shown in Figure 2-7.
The LLVM Framework is language-agnostic supporting multiple source code languages via dedicated frontends that transform language-specific code to language-agnostic LLVM IR code. The LLVM Framework performs code optimizations on the IR. Finally, the IR can be transformed to native code for a multitude of target architectures via dedicated LLVM backends.
This architecture enables one to transform n source code languages into m compilation targets, leading to $n \times m$ supported configurations. Furthermore, new languages can

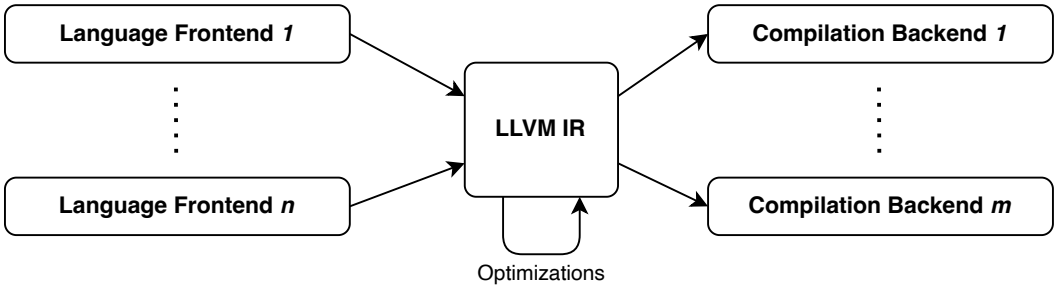


Figure 2-7: LLVM Architecture

easily utilize the LLVM compilation and optimization capabilities by providing a dedicated LLVM frontend, which was also done for the Rust language.
Figure 2-8 shows an excerpt of internal structure of the LLVM IR, which was deduced from the LLVM 17 Source Code.[11] The IR consists of modules, which represent one single

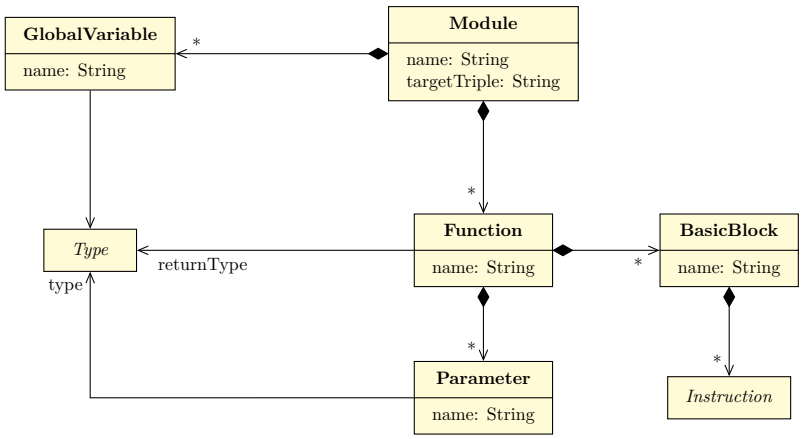


Figure 2-8: Excerpt of the LLVM Intermediate Representation

unit of compilation (also called translation unit). A module contains global variables and named functions. The body of a function is described using a control flow graph (CFG) that consists of basic blocks. A basic block consists of instructions. The values of the LLVM IR are SSA (Single Static Assign) values, which means that they can only be assigned once in the program.
Overall, the LLVM IR is language-agnostic and represents code in a control flow graph, as needed by a data-flow analysis. This makes the LLVM IR a relevant candidate for use in this thesis.

[11] https://github.com/llvm/llvm-project/tree/llvmorg-17.0.4

3 Definition of Requirements

This chapter defines the requirements for the static code analysis tool developed in this thesis based on background information about the Rust ecosystem and the program representations, as outlined in Chapter 2. In Section 3.1, seven vulnerabilities are selected for further inspection. In Section 3.2, a well-defined scope for the data-flow analysis is defined. In Section 3.3, the selected vulnerabilities are analyzed and requirements are derived from them. In Section 3.4, the differences between program representations are analyzed and IRs are selected for the approach. Section 3.5 concludes the chapter with a summary of the requirements defined for the static code analysis tool.

3.1 Selected Rust Vulnerabilities for Further Analysis

In this section, vulnerabilities are selected for later analysis based on the entries in the RustSec Database, as introduced in Section 2.4. The RustSec Database contains vulnerabilities in Rust crates published in the Rust Package Registry.
To select vulnerabilities, the number of transitively affected crates is used. Although the dependents of vulnerable crates are potentially vulnerable themselves, these dependents are not listed explicitly in the RustSec Database. The database only lists the top-level crate that defines the vulnerable code. To acquire the number of transitively affected crates, a pre-evaluation was performed. The underlying data set has been produced by automated download of all crates from the Rust Package Registry and a subsequent dependency analysis on each of them, as described in Section A2.2.
The CVSS score (Mell, Scarfone, & Romanosky, 2006) is a metric that defines the impact of a vulnerability. It was deliberately decided not to order the vulnerabilities by their impact, as defined by the CVSS score, but rather by the number of transitively affected crates because ordering by CVSS score would have put vulnerabilities in the center of attention that only affect a very limited number of potentially rather obscure crates. For instance, `RUSTSEC-2021-0021` *(nb-connect invalidly assumes the memory layout of std::net::SocketAddr)* has been assigned a CVSS score of 9.8 and 381 crates were found to be affected. In contrast, the vulnerability `RUSTSEC-2020-0071` *(potential segfault in the time crate)* affects 24,525 crates and has been assigned a CVSS score of 6.2.
Table 3-1 shows important Rust vulnerabilities that are prevalent in the Rust ecosystem and will be further analyzed in this chapter to derive requirements for the approach.
The table column *Count* (T) lists the number of different crates that are affected by the vulnerability. The table column *Count* ($T \times V$) lists the number of distinct versions of the crate that are vulnerable (if multiple versions of the same crate are affected, they are counted separately).

The first four vulnerabilities are, consequently, selected because a large number of crates are vulnerable to them. These four vulnerabilities are chosen by ordering the vulnerability definitions by their *Count* ($T \times V$) value in descending order, rejecting vulnerabilities that do not define affected functions (as required by the proposed approach), and then selecting those vulnerabilities with a value of $T > 10.000$.
The last three vulnerabilities in the table have been found to occur only when a certain parameter value is passed to the affected function. This type of vulnerability is of particular interest, as a data-flow analysis can be used to determine whether the value of the vulnerable parameter reaches the affected function or not, potentially leading to more precise results. The actual values that cause the vulnerability were manually deduced from the affected source code. These derived information are marked with an asterisk in the table. Although only three vulnerabilities of this kind were identified with reasonable effort, there might be more vulnerabilities occurring only for specific parameter values.

Vulnerability	**Crate**	**Count (T)**	**Count (T×V)**	**Functions**	**Parameters**
High Reach					

RUSTSEC-2020-0071	time	24525	179415	`time::OffsetDateTime::now_local`
				`time::OffsetDateTime::try_now_local`,
				`time::UtcOffset::current_local_offset`,
				`time::UtcOffset::local_offset_a`,
				`time::UtcOffset::try_current_local_offset`,
				`time::UtcOffset::try_local_offset_at`,
				`time::at`,
				`time::at_utc`,
				`time::now`
					∅

RUSTSEC-2023-0018	remove_dir_all	12321	85713	`remove_dir_all::ensure_empty_dir`
				`remove_dir_all::remove_dir_all`,
				`remove_dir_all::remove_dir_contents`
					∅

RUSTSEC-2021-0124	tokio	11208	68598	`tokio::sync::oneshot::Receiver::close`	∅
RUSTSEC-2022-0078	bumpalo	10091	64676	`bumpalo::collections::vec::Vec::into_iter`	∅
Known Parameter Values					
RUSTSEC-2023-0044	openssl	8328	53657	`openssl::X509VerifyParamRef::set_host`	P_1 = "" (*)
RUSTSEC-2023-0024	openssl	7646	49244	`openssl::X509Extension::new`, `openssl::X509Extension::new_nid`	P_2 = None (*)
RUSTSEC-2022-0051	lz4-sys	177	1261	`lz4_sys::LZ4_decompress_safe` (*) `lz4_sys::LZ4_decompress_safe_continue` (*)	$P_4 < 0 \wedge P_3 \neq 0$ (*) $P_5 < 0 \wedge P_4 \neq 0$ (*)

(*) = Information is not part of the RustSec Database
P_i = Parameter number i of respective function(s)

Table 3-1: Selected Rust Vulnerabilities for Further Analysis

3.2 Scope for the Approach

It is acknowledged that commercial static code analysis products should be able to support alias sensitivity, field sensitivity, array sensitivity, and dynamic function calls. However, in the scope of this thesis, it is not anticipated that these characteristics will have a major impact on the research questions as outlined in Section 1.3. Therefore, Requirement *R1 (No support for alias sensitivity, field sensitivity or array sensitivity, and no support for dynamic function calls)* is defined.

3.3 Analysis of Selected Vulnerabilities

In this section, the seven selected vulnerabilities as shown in Table 3-1 are analyzed. For each vulnerability, the underlying problem is discussed, and vulnerable code is presented and analyzed. The insights are then used to define the requirements for the approach.

3.3.1 RUSTSEC-2020-0071 – Potential Segfault in the Time Crate

RUSTSEC:	RUSTSEC-2020-0071
CVE:	CVE-2021-45710
CVSS:	5.3
CVE Vector:	CVSS:3.1/AV:N/AC:H/PR:L/UI:N/S:U/C:N/I:N/A:H
CWE:	CWE-825 – Expired Pointer Dereference

The vulnerability `RUSTSEC-2020-0071`[1] is associated with the crate `time`[2], which allows retrieving the system time and calculating time durations. The vulnerability describes a possible segmentation fault on Unix systems if an environment variable is set on one thread and an affected function is called on another thread, as they access the environment variables in a way that is not thread-safe. The crate `utc-0.1.0`[3] has been identified as a directly affected crate and will be investigated next. The crate `utc` defines an application that only invokes the function `time::now_utc()` and prints the time in a formatted way on the console. The file `main.rs` contains the source code of the crate and is shown in Listing 3-1. The main function invokes the exported function `time::now_utc()`. The definition of the called function is shown in Listing 3-2.

[1] https://rustsec.org/advisories/RUSTSEC-2020-0071
[2] https://crates.io/crates/time
[3] https://crates.io/crates/utc

```
1  extern crate time;
2
3  fn main() {
4    let utc = time::now_utc();
5    let format = "%Y-%m-%d %H:%M:%S";
6    println!("{}", utc.strftime(format).expect("format string error"));
7  }
```

Listing 3-1: Source Code of Crate utf-0.1.0

The function `time::now_utc()` invokes the function `time::at_utc()`, which is listed as an affected function in the vulnerability definition. Note that the `time::now_utc()` function itself is not marked as affected in the vulnerability definition. In conclusion, the function `time::now_utc()` is transitively affected, but is not marked as affected itself.

```
412  /// Returns the current time in UTC
413  pub fn now_utc() -> Tm {
414    at_utc(get_time())
415  }
```

Listing 3-2: Function now_utc in Crate time-0.1.45

The fact that a transitively affected function is omitted in the vulnerability definition has consequences for the approach. It can be argued that listing transitively affected functions in the vulnerability definition is superfluous because they describe redundant information that can be obtained by analyzing the code. Moreover, it requires the submitter of the vulnerability definition to find all the calling functions, which is a technical task that can be prone to human errors when done manually. This task is much better suited for a data-flow analysis. So, to obtain this redundant information, the data-flow analysis will need to analyze into the vulnerable crate itself to find transitively affected functions and must not rely entirely on the information provided in the vulnerability definition. Requirement *R2 (Infer transitively affected functions in vulnerable crate)* represents this requirement.

The software project analyzed by the data-flow analysis can be an application program (in this case the `utc` program) or it can be a software library (in this case the `time` library). Software libraries provide functionality for other software projects to use by exporting functions that can be called by dependents. Therefore, in the context of software security, every function that is exported by a software library can be considered an entry point for security threats. Application programs usually define a single point of entry, which is the *main* method of the program. In order for the data-flow analysis to support both application programs and software libraries, Requirement *R3 (Support application programs and software libraries)* is defined.

3.3.2 RUSTSEC-2023-0018 – Race Condition Enabling Link Following and Time-Of-Check Time-Of-Use (Toctou)

RUSTSEC:	RUSTSEC-2023-0018
CVE:	*Not assigned*
CVSS:	4.9
CVE Vector:	CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:U/C:N/I:N/A:H
CWE:	CWE-367 – Time-of-check Time-of-use (TOCTOU) Race Condition

The vulnerability `RUSTSEC-2023-0018`[4] is associated with the crate `remove_dir_all`[5], which provides functionality to delete a directory and its contents recursively. This functionality is also provided by the function `std::fs::remove_dir_all` in the Rust standard library, but the crate implements a specialized version for the Windows platform. For all other platforms, the crate delegates to the `std::fs::remove_dir_all` function in the Rust standard library. This is done using conditional compilation, which selects one of multiple `pub use` statements in the `lib.rs` file of the `remove_dir_all` crate as shown in Listing 3-3.

```
1 #[cfg(windows)]
2 pub use self::fs::remove_dir_all;
3
4 #[cfg(not(windows))]
5 pub use std::fs::remove_dir_all;
```

Listing 3-3: Reexport of Different Implementations Based on Selected Platform

The vulnerability describes a possible race condition in the Windows-specific implementation of the crate `remove_dir_all`, which involves deleting unexpected parts of the filesystem if a symbolic link is created at the right time. This can occur because the function `remove_dir_call` checks for symbolic links initially, but if the operating system performs a context switch to a different thread after the function already checked for symbolic links, an attacker could interfere at that time and create a symbolic link in the file system, which will not be recognized by the function. This is a TOCTOU (Time of Check to Time of Use) problem.

[4] https://rustsec.org/advisories/RUSTSEC-2023-0018
[5] https://crates.io/crates/remove_dir_all

The crate `uu_rm-0.0.1`[6] was identified to be affected by the vulnerability, as it invokes the affected function `remove_dir_all` in an internal function `handle_dir` via several calls:

1. `uu_rm::main` (entry point, generated by macro)
2. `uu_rm::uumain`[7]
3. `uu_rm::remove`[8]
4. `uu_rm::handle_dir`[9]
5. `remove_dir_all::remove_dir_all`[10] (vulnerable)

In this crate, the `main` function automatically generated by the macro call `uucore_procs::main!(uu_rm)`[11]. For the data-flow analysis to find this flow, it needs to support Rust macros. Therefore, Requirement *R4 (Support Rust macros)* is defined and in Section 3.4, the program representations in the Rust compiler are analyzed according to the way they represent macros.
The vulnerability `RUSTSEC-2023-0018` can only be reproduced on the Windows platform because the vulnerable code is only conditionally compiled if the Windows platform is targeted. Therefore, Requirement *R5 (Support analyzing platform-specific Rust code)* is defined to support analyzing code that uses conditional compilation to provide functionality for a specific platform. The keyword `pub use` is used to re-export different implementations based on the selected platform, as shown in Listing 3-3.
The Rust Book[12] recommends using this language feature when creating libraries that export items from several modules, as this allows one to hide the internal module structure of the library and allows clients of the library to import the items from one single place without specifying the internal module names in the library.
Therefore, it is expected that this feature is prevalent in Rust libraries and thus the static code analysis should support it. Consequently, Requirement *R6 (Support re-exported functions)* is defined.

[6] https://crates.io/crates/uu_rm
[7] https://github.com/uutils/coreutils/blob/0.0.1/src/rm/rm.rs#L48
[8] https://github.com/uutils/coreutils/blob/0.0.1/src/rm/rm.rs#L157
[9] https://github.com/uutils/coreutils/blob/0.0.1/src/rm/rm.rs#L191
[10] https://github.com/XAMPPRocky/remove_dir_all/blob/v0.5.1/src/fs.rs#L33
[11] https://crates.io/crates/uucore_procs
[12] https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html#re-exporting-names-with-pub-use

3.3.3 RUSTSEC-2021-0124 – Data Race When Sending and Receiving After Closing a Oneshot Channel

RUSTSEC:	RUSTSEC-2021-0124
CVE:	CVE-2021-45710
CVSS:	8.1
CVE Vector:	CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H
CWE:	CWE-362 – Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

The vulnerability `RUSTSEC-2021-0124`[13] is associated with the crate `tokio`[14], which provides a platform for writing asynchronous applications. The vulnerability describes a race condition that can occur after a one-shot channel is closed using the method `close` on the type `tokio::sync::oneshot::Receiver`. The following code in Listing 3-4 invokes the vulnerable function.

```
1 fn main() {
2   let (tx, mut rx) = tokio::sync::oneshot::channel::<usize>();
3   rx.close();
4   let _ = rx.try_recv();
5 }
```

Listing 3-4: Reproducer for Vulnerability RUSTSEC-2023-0044

Race condition bugs can be hard to spot during software testing, as they may not always materialize when the program is executed. Testing may not uncover these issues. However, the static analysis approach discussed in this thesis can detect them reliably. For this vulnerability, no further requirements need to be defined.

[13] https://rustsec.org/advisories/RUSTSEC-2021-0124
[14] https://crates.io/crates/tokio

3.3.4 RUSTSEC-2022-0078 – Use-After-Free Due to a Lifetime Error in Vec::into_iter()

RUSTSEC:	RUSTSEC-2022-0078
CVE:	*Not assigned*
CVSS:	8.2
CVE Vector:	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:H
CWE:	CWE-416 – Use After Free

The vulnerability `RUSTSEC-2022-0078`[15] is associated with the crate `bumpalo`[16], which provides functionality to allocate memory efficiently in an arena. An arena is a memory pool that allows clients to allocate data in it and allows to deallocate all data at once by dropping the arena.
The vulnerability definition describes a possible use-after-free bug when a vector iterator is created by the function `bumpalo::collections::vec::Vec::into_iter` and is used after the arena providing the underlying memory for the vector has already been dropped. An example code is provided in description text, which demonstrates a possible memory corruption that can arise from executing that code. This reproducer code is shown in Listing 3-5.

```
1  fn main() {
2    let bump = bumpalo::Bump::new();
3    let mut vec = bumpalo::collections::Vec::new_in(&bump);
4    vec.extend([0x01u8; 32]);
5    let into_iter = vec.into_iter();
6    drop(bump);
7
8    for _ in 0..100 {
9      let reuse_bump = bumpalo::Bump::new();
10     let _reuse_alloc = reuse_bump.alloc([0x41u8; 10]);
11   }
12
13   for x in into_iter {
14     print!("0x{:02x} ", x);
15   }
16   println!();
17 }
```

Listing 3-5: Reproducer for Vulnerability RUSTSEC-2022-0078

In line 2 of the reproducer code, a new bump arena is allocated, and in line 3 a vector is allocated, which is backed by the arena's memory. The vector is filled with 32 values and an iterator over the vector is created using the affected function `into_iter` in line 5.

[15] https://rustsec.org/advisories/RUSTSEC-2022-0078
[16] https://crates.io/crates/bumpalo

In line 6, the arena is dropped, but in line 13 the iterator is still used to read from the deallocated arena. Lines 8 to 11 allocate new arenas and fill them with data to increase the chance that memory of the old arena is reused by a new arena and the data of a new arena is printed at runtime while reading from the old iterator. Overall, the affected function is invoked in the main function in line 5 and the data-flow analysis needs to be able to identify it as such.
Rust enforces that the lifetime of a reference does not exceed the lifetime of the referenced value using the borrow checker. The implementation of the bump arena escapes this check by using unsafe code. Using the iterator after dropping the underlying arena is a *use-after-free*, which is undefined behavior. For this vulnerability, no further requirements need to be defined.

3.3.5 RUSTSEC-2023-0044 – Openssl X509VerifyParamRef::set_host Buffer Over-Read

RUSTSEC:	RUSTSEC-2023-0044
CVE:	*Not assigned*
CVSS:	7.3
CVE Vector:	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:L
CWE:	CWE-416 – Use After Free

The vulnerability `RUSTSEC-2023-0044`[17] is associated with the crate `openssl`[18], which provides access to the cryptography library OpenSSL. The vulnerability description refers to a corresponding issue in the Github Repository that mentions a reproducing example code, as shown in Listing 3-6.

```
1  extern crate native_tls;
2
3  use native_tls::TlsConnector;
4  use std::io::{Read, Write};
5  use std::net::TcpStream;
6
7  fn main() {
8    let connector = TlsConnector::builder().use_sni(false).build().unwrap();
9    let host = ""; // <-- Here, an empty string is used as host (!)
10   let stream = TcpStream::connect("google.com:443").unwrap();
11   let mut stream = connector.connect(host, stream).unwrap();
12   stream.write_all(b"GET / HTTP/1.0\r\n\r\n").unwrap();
13   let mut res = vec![];
14   stream.read_to_end(&mut res).unwrap();
15   println!("{}", String::from_utf8_lossy(&res));
16 }
```

[17] https://rustsec.org/advisories/RUSTSEC-2022-0044
[18] https://crates.io/crates/openssl

Listing 3-6: Reproducer for Vulnerability RUSTSEC-2023-0044

The vulnerability describes a possible buffer overread when passing an empty string to the affected function `set_host` on the type `openssl::X509VerifyParamRef`, which then allows an attacker to read arbitrary memory.
The reproducer code does not directly interact with the crate `openssl` but opens a TLS connection using the crate `native_tls`, which itself delegates to OpenSSL on certain platforms. This TLS connection is established in the code by invoking the function `native_tls::TlsConnector::connect`, which internally calls the affected function `X509VerifyParamRef::set_host` via several calls:

1. `reproducer::main` (entry point)
2. `native_tls::TlsConnector::connect`
3. `native_tls::imp::TlsConnector::connect`
4. `openssl::ssl::connector::ConnectConfiguration::connect`
5. `openssl::ssl::connector::ConnectConfiguration::into_ssl`
6. `openssl::ssl::connector::setup_verify_hostname`
7. `openssl::x509::verify::X509VerifyParamRef::set_host` (vulnerable)

The functions that are part of this call chain originate from three different crates: The top-level crate containing the reproducer code, the crate `native_tls` and the crate `openssl`. To identify vulnerabilities that span multiple crates, the data-flow analysis must scan across multiple levels of dependencies. To explore the possibilities of supporting a data-flow analysis that can operate across external dependencies, the Requirement *R7 (Analyze across dependency boundaries)* is defined.
This type of vulnerability is of particular interest, as a data-flow analysis can be used to determine if an empty string of the vulnerable parameter reaches the affected function or not, potentially leading to more precise results. To support this, the RustSec Database must contain information that the empty string is vulnerable for the method `set_host`. As the database schema does not support this information, it needs to be extended, which is represented by Requirement *R8 (Expand the structure of the RustSec Database to incorporate security-related parameter values and identify these values in the data-flow analysis)*.

3.3.6 RUSTSEC-2023-0024 – Openssl X509Extension::new and X509Extension::new_nid Null Pointer Dereference

RUSTSEC:	RUSTSEC-2023-0024
CVE:	*Not assigned*
CVSS:	7.5
CVE Vector:	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
CWE:	CWE-476 – NULL Pointer Dereference

The vulnerability `RUSTSEC-2023-0024`[19] is associated with the crate `openssl`[20] and describes a possible segmentation fault if the method `new` or the method `new_nid` of type `X509Extension` is executed with a context argument of `None`. Both methods accept the context argument at their fourth parameter. The artificial reproducer code shown in Listing 3-7 was created to intentionally invoke both affected functions with a context argument of `None`. This code should be recognized as vulnerable by the data-flow analysis.
This type of vulnerability is of particular interest, as a data-flow analysis can be used to determine if the value `None` of the vulnerable parameter reaches the affected function or not, potentially leading to more precise results. To support this, the RustSec Database must contain the special value `None` for the second parameter so that the analysis can access them. Requirement *R8 (Expand the structure of the RustSec Database to incorporate security-related parameter values and identify these values in the data-flow analysis)* is already defined and therefore, no further requirements are being defined.

```
1  fn main() {
2    openssl::x509::X509Extension::new(
3      None,
4      None, // <-- context is None here (!)
5      "",
6      ""
7    );
8    openssl::x509::X509Extension::new_nid(
9      None,
10     None, // <-- context is None here (!)
11     openssl::nid::Nid::from_raw(0),
12     ""
13   );
14 }
```

Listing 3-7: Artificial Reproducer Code for Vulnerability RUSTSEC-2023-0024

[19] https://rustsec.org/advisories/RUSTSEC-2023-0024
[20] https://crates.io/crates/openssl

3.3.7 RUSTSEC-2022-0051 – Memory Corruption in liblz4

RUSTSEC:	RUSTSEC-2023-0051
CVE:	*Not assigned*
CVSS:	9.8
CVE Vector:	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
CWE:	CWE-787 – Out-of-bounds Write, CWE-190 – Integer Overflow or Wraparound

The vulnerability `RUSTSEC-2022-0051`[21] is associated with the crate `lz4-sys`[22], which bundles the LZ4 system library to enable Rust programs to compress and decompress data. System packages usually provide the necessary system functionality to Rust programs in a safe manner. If this is not possible, the API is marked with the keyword `unsafe`. Therefore, functions that are not labeled as unsafe guarantee that the observable behavior of the function is safe. In this particular vulnerability, the problematic code is located within the `liblz4` library itself. The crate `lz4-sys` provides access to this vulnerable code through a safe function, but the observable behavior of that function is actually unsafe, and therefore the crate `lz4-sys` itself is vulnerable.
The original vulnerability in the used C library `liblz4`[23] is located in its private helper function `LZ4_decompress_generic`[24], which provides functionality to decompress an LZ4 archive. The function `LZ4_decompress_generic` receives the size of the compressed input data and the size of the output buffer through its parameters, as can be seen in the signature of the function shown in Listing 3-8.

```
1  LZ4_decompress_generic(
2    const char* const src,
3    char* const dst,
4    int srcSize,
5    int outputSize,
6    earlyEnd_directive partialDecoding,
7    dict_directive dict,
8    const BYTE* const lowPrefix,
9    const BYTE* const dictStart,
10   const size_t dictSize
11 );
```

Listing 3-8: Function Signature of C-Function LZ4_decompress_generic

In general, buffer sizes are never negative, but the function `LZ4_decompress_generic` declares the corresponding parameters as signed integers (using the C keyword `int`). This technically allows negative numbers to be passed to it. Internally, the function does not

[21] https://rustsec.org/advisories/RUSTSEC-2023-0051
[22] https://crates.io/crates/lz4-sys
[23] https://github.com/lz4/lz4/tree/8301a21773ef
[24] https://github.com/lz4/lz4/blob/8301a21773ef/lib/lz4.c#L1738

check for this case but unconditionally adds the output size to the output base pointer to calculate a pointer to the end of the output data. In the case of a negative size argument, this leads to a wrong calculation of the end pointer, which can then point to arbitrary data.

The vulnerable function `LZ4_decompress_generic` is not exported to Rust and, therefore, is not mentioned as affected function in the vulnerability definition. However, upon further inspection, it was identified that the functions `LZ4_decompress_safe`[25] and `LZ4_decompress_safe_continue`[26] are accessible from Rust and actually call the vulnerable function `LZ4_decompress_generic` internally. Therefore, these two functions should be marked as affected in the vulnerability definition.

Furthermore, it became obvious that the vulnerability is prevented when the number zero is passed as the input buffer size because the C function checks for this case and then leaves the function early. Therefore, the data-flow analysis can have increased precision when it does not report a vulnerability in this case.

The artificial reproducer code shown in Listing 3-9 was created to invoke the affected function with a negative output size and a non-zero input size. This code should be recognized as vulnerable by the designed data-flow analysis.

```
1  fn main() {
2    unsafe {
3      let source = [0u8; 8];
4      let mut target = [0u8; 64];
5      lz4_sys::LZ4_decompress_safe(
6        source.as_ptr() as *const std::ffi::c_char,
7        target.as_mut_ptr() as *mut std::ffi::c_char,
8        8,   // <-- inputSize is not equal to 0 (!)
9        -1,  // <-- outputSize is negative (!)
10     );
11   }
12 }
```

Listing 3-9: Artificial Reproducer Code for Vulnerability RUSTSEC-2022-0051

This type of vulnerability is of particular interest, as a data-flow analysis can be used to determine whether vulnerable values reach the corresponding parameters of the affected function or not. By checking for vulnerable parameter ranges, more precise results can be achieved. To support this, the RustSec Database must mark negative values as vulnerable for the parameter representing the output size as well as mark the value zero as not vulnerable for the parameter representing the input size. Requirement *R8 (Expand the structure of the RustSec Database to incorporate security-related parameter values and identify these values in the data-flow analysis)* is already defined and therefore, no further requirements are being defined.

[25] https://github.com/lz4/lz4/blob/8301a21773ef/lib/lz4.c#L2171
[26] https://github.com/lz4/lz4/blob/8301a21773ef/lib/lz4.c#L2322

For the function `LZ4_decompress_safe`, the third parameter represents the input size, and the fourth parameter represents the output size and for the function `LZ4_decompress_safe_continue`, the fourth parameter represents the input size, and the fifth parameter represents the output size. In Table 3-1, the respective parameter conditions are encoded accordingly.

3.4 Selection of Program Representation

In this section, the program representations used in the Rust compiler as introduced in Section 2.5 are compared in terms of their applicability in the approach.
The AST, HIR and THIR do not explicitly define the control flow but rather describe implicit control flow that is not formally encoded in the IR. Therefore, they are not selected as candidates to perform a data-flow analysis on them. The HIR provides access to the structural items of the Rust code. In the context of a code analysis, this information are helpful to reason about the type hierarchy and to generate a callgraph. The MIR describes code of function bodies that has been lowered into a control flow graph and therefore contains only explicit control flow. In MIR, macros are expanded, types are resolved, and functions are resolved. Therefore, the MIR is considered as a candidate in the context of the designed static data-flow analysis. LLVM IR is generated from the MIR and thus has comparable properties (macros are expanded). The LLVM IR models the control flow of functions explicitly using a control flow graph. Therefore, the LLVM IR is considered as a candidate in the context of the designed static data-flow analysis.
As both the MIR and LLVM IR enable data-flow analysis, the applicability of these IRs in the context of this thesis is explored next. The LLVM IR is language-agnostic, which allows to write analyses that operate on code originating from different source code languages like Rust, C, C++ and others. As the LLVM IR code generated by the Rust compiler can originate from multiple software projects that are part of the compilation process, the LLVM IR represents the code of the complete dependency tree, while the MIR represents code originating from one single software project and only lives during compilation of one software project. While generally the interaction between software projects is modeled in MIR by referring to external crates, it is expected that a data-flow analysis on the MIR needs to perform additional work to correlate code fragments that originate from different software projects and interact with each other.
In the context of this thesis, the applicability of tracking data flows across external dependencies of the IRs is explored next in response to Requirement *R7 (Analyze across dependency boundaries)*. The database-based approach described in this thesis utilizes the RustSec Database to query for vulnerable functions in published software libraries. The database stores the name of the software project in which the vulnerability originates

and the version of that software project. It is therefore necessary that the selected IR contains the function names and the names of the software project to query the database for results. In the presence of external dependencies, there are multiple software projects that contribute to the code that is being compiled into a single executable. Therefore, the approach requires detailed information for each function in the program, from which software project they originated and in which version that software project is present to query the RustSec Database for the corresponding vulnerability information.
The Rust Compiler has access to this information when used with the Package Manager Cargo. The MIR contains complete type information of the Rust types using provenance data that refers back to the corresponding items in the HIR. The MIR provides fully qualified function names as used in the RustSec Database and provides access to the version of the compiled software via data passed from Cargo to the Rust Compiler.
This makes MIR the representation with the highest level of abstraction where the needed information for an inter-crate analysis is still present and can be used to query the RustSec Database for specific functions and project versions.
The LLVM IR does not explicitly model the concept of software projects, crate version numbers, or fully qualified names of Rust functions, as it is a language-agnostic IR and Rust is only one language that transforms into it. Still, a name-mangling scheme[27] is used by the MIR to LLVM transformation step that encodes additional information in the function names. While name manglings are intended to support the linking process, it is acknowledged that they can be used to reverse engineer Rust-specific information about the function.
To explore which Rust-specific information is included in this mangled name, a small Rust program is shown in Figure 3-1a that executes the function `dep::Widget::do_work()`, which is defined in the external software library and Figure 3-1b shows the definition of that software library.

```rust
pub fn main() {

  let w = dep::Widget;

  w.do_work();
}
```

(a) Application "app"

```rust
pub struct Widget;
impl Widget {
  pub fn do_work(&self) {
    // empty
  }
}
```

(b) Software Library "dep"

Figure 3-1: Rust Code Involving Application and Library

[27] https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html

The generated LLVM IR code for this Rust program is shown in Listing 3-10. The function name `_ZN3dep6Widget7do_work17he2591b8ef31ebe04E` is mangled and refers to the method `do_work` on the struct `Widget` in the crate `dep`. Additionally, a comment in line 9 hints at the fully qualified function name as well. While this representation is considered useful for debugging purposes, relying on them for a data-flow analysis seems fragile, as these names are implementation details of the MIR to LLVM transformation that could easily change in the future in subtle ways, because the MIR-to-LLVM transformation evolves quickly.

```
1  ; app::main
2  ; Function Attrs: uwtable
3  define internal void @_ZN6app4main17ha969bc26af2b8f99E() unnamed_addr #1 !dbg !174
4  {
5  start:
6    %_1 = alloca %"dep::Widget", align 1
7    %x.dbg.spill = alloca %"dep::Widget", align 1
8    call void @llvm.dbg.declare(metadata ptr %x.dbg.spill, metadata !178, metadata !
       ↪ DIExpression()), !dbg !182
9  ; call dep::Widget::do_work
10   call void @_ZN3dep6Widget7do_work17he2591b8ef31ebe04E(ptr align 1 %_1), !dbg !183
11   ret void, !dbg !184
12 }
```

Listing 3-10: Generated LLVM IR

Since the recently released version LLVM 17[28], the use of Typed Pointers is no longer supported as part of a transition[29] to Opaque Pointers, which carry no type information. As a consequence of this change, it is expected that structs with the same memory layout are merged into a single struct to enable low-level optimizations.[30] This makes it harder to correlate them with a RustSec Database or to query high-level type information on the level of structs, traits and crates. Being language-agnostic, it gives up Rust-specific information, and supporting low-level optimizations, it gives up high-level type information. Moreover, the development of Stable MIR (SMIR) suggests that the designers of Rust assume that this is the right place to introduce a stable API for the interaction with static code analyses.
Table 3-2 summarizes the discussed properties of the compared program representations.

[28] https://releases.llvm.org/17.0.1/docs/ReleaseNotes.html
[29] https://llvm.org/devmtg/2022-04-03/slides/keynote.Opaque.Pointers.Are.Coming.pdf
[30] https://llvm.org/docs/OpaquePointers.html#issues-with-explicit-pointee-types

	AST	**HIR**	THIR	**MIR**	LLVM IR
Language Agnostic	NO	NO	NO	NO	✓
Macros Expanded	NO	✓	✓	✓	✓
Explicit Control Flow	NO	NO	NO	✓	✓
Resolved Rust Type Information	NO	✓	✓	✓	NO

Table 3-2: Comparison of Program Representations for Rust Code

Overall, it is estimated that using the Rust IRs provides the most Rust-specific information for a data-flow analysis across software projects and integrates best with the RustSec Database. Therefore, in the context of this thesis, the MIR is selected as the IR on which the data-flow analysis is performed. Additionally, the HIR is used to gain access to higher-level type information on the level of structs, traits, and crates. Consequently, Requirement *R9 (Operate on the MIR and HIR)* is defined.

3.5 Conclusion

This chapter discussed important security vulnerabilities in the Rust ecosystem based on the number of crates affected by them and established requirements for the proposed solution. The following chapters present an approach to detect these vulnerabilities in actual Rust code. This will be done by performing a data-flow analysis after a dependency analysis, which is expected to give more accurate results than those obtained from the dependency analysis alone. The identified requirements for the analysis tool are listed below.

R1 No support for alias sensitivity, field sensitivity, or array sensitivity, and no support for dynamic function calls.

R2 Infer transitively affected functions in vulnerable crate.

R3 Support application programs and software libraries.

R4 Support Rust macros.

R5 Support analyzing platform-specific Rust code.

R6 Support re-exported functions.

R7 Analyze across dependency boundaries.

R8 Expand the structure of the RustSec Database to incorporate security-related parameter values and identify these values in the data-flow analysis.

R9 Operate on the MIR and HIR.

4 Concept of the Static Code Analysis

This chapter outlines the concept and design of a static analysis tool that implements a hybrid approach combining a dependency analysis of the RustSec Database, followed by a subsequent data-flow analysis that scans external dependencies to automatically audit the Rust code for vulnerabilities. The design process is based on the requirements defined in Section 3.5. The High-Level Architecture is shown in Section 4.1 and outlines the components involved in the approach. The behavior of these components are discussed in individual sections. Section 4.2 discusses an extension to the RustSec Database, as required by the approach. Section 4.3 outlines process performed in the Data-Flow Analyzer component and discusses the data structures and algorithms used for the data-flow analysis. The Data-Flow Analyzer operates on a single crate. Section 4.4 outlines the so-called inter-crate analysis that extends the capabilities of the tool to analyze across crate boundaries and scan into the complete dependency tree of the analyzed crate. Figure 6.4.1 denotes the criteria that define whether a vulnerability is reported to the tool user based on the type of analyzed crate and the information collected during analysis.

4.1 High-Level Architecture

The high-level architecture depicted in Figure 4-1 enables a data-flow analysis that covers all external dependencies. The *Cargo Subcommand* is the user interface of the developed tool, which extends the Package Manager Cargo and takes Rust Source Code and the Extended RustSec Database as input.

The first step of the *Cargo Subcommand* is to build a dependency tree. This tree exists because Cargo forbids cyclic dependencies. The root of this tree is the crate that the tool is executed on. The transitive dependencies of this top-level crate contained as nodes in the tree. The leaf nodes of the tree are crates that do not depend on any other crates.

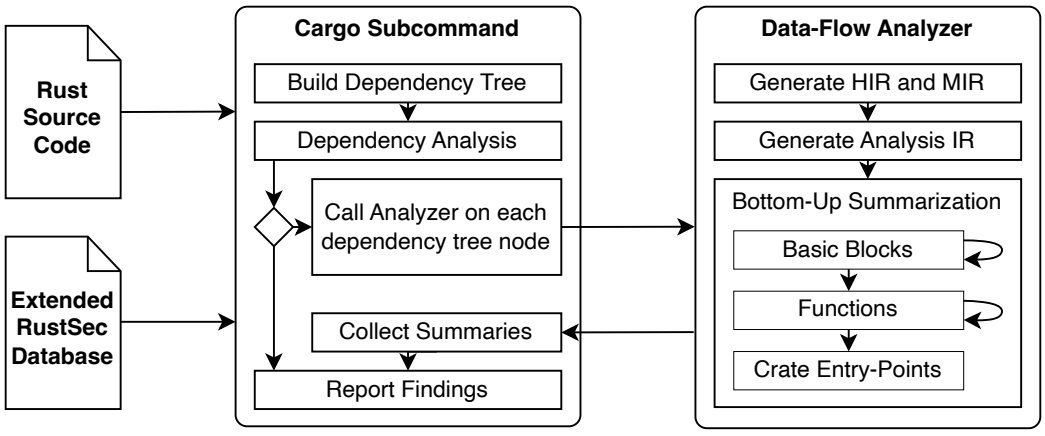


Figure 4-1: High-Level Architecture

The second step of the *Cargo Subcommand* is to perform a dependency analysis on the dependency tree to identify dependencies that are associated with known vulnerabilities in the RustSec Database. This is done using the `rustsec`[1] client library that allows issuing queries on the RustSec Database. Depending on the results of the dependency analysis, the next step is decided.
In case no potentially vulnerable dependencies are identified, the tool stops further analysis and reports to the user that no vulnerabilities were found.
If potentially vulnerable dependencies are identified, the tool continues to perform a data-flow analysis that operates on the whole dependency tree.
For this, the *Data-Flow Analyzer* is called on each tree node in the dependency tree so that dependencies are analyzed before their dependents (bottom-up).
The *Data-Flow Analyzer* assesses the security of a single crate and the resulting summaries for each crate are persisted on the filesystem. After all crates are summarized, the *Cargo Subcommand* collects the persisted summaries, and reports the contained results of the data-flow analysis for the top-level crate as well as the results of the dependency analysis to the user.

4.2 Extended RustSec Database

The High-Level Architecture shown in Figure 4-1 shows that the *Cargo Subcommand* requires the Extended RustSec Database as input. In this chapter, this extension to the RustSec Database is outlined. In Chapter 3 it has been identified that there are vulnerabilities that are only triggered when a function is called with specific parameter values passed as arguments. Still, the RustSec Database as introduced in Section 2.4, does not define specific parameter values that trigger a vulnerability. In response to Requirement *R8 (Expand the structure of the RustSec Database to incorporate security-related parameter values and identify these values in the data-flow analysis)*, the schema is now extended to include such parameter values as outlined in Figure 4-2.
Affected functions are represented in the schema by the type `Function`. The extension allows to define parameter specifications for an affected function. The parameter specification is represented by the type `AffectedParameterSpec` and describes a parameter that is vulnerable to a specific value or range of values. The 1-based index of the affected parameter is stored in the attribute `parameter`. The literal value is stored in the attribute `rustValue` and the type of the literal value is stored in the attribute `rustType`. The supported comparison operators are represented by the type `AffectedParameterComparisonOperator`.

[1] https://crates.io/crates/rustsec

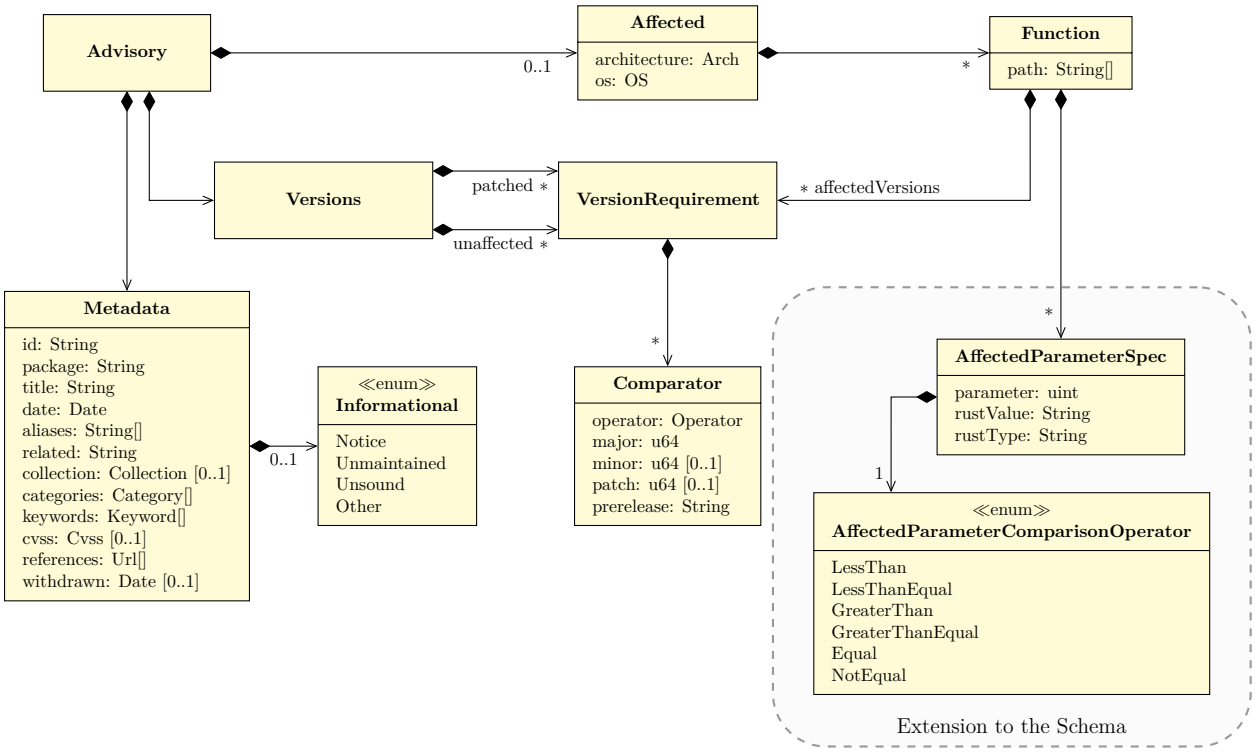


Figure 4-2: Schema of Rustsec Database and the Proposed Extensions

Table 4-1 shows the Rust types and Rust values supported by the schema extension. The tool validates that no unsupported combinations of types and values are used.

Rust Types	Supported Rust Values
Type `Option`	`None`
Type `String`	`""` (empty string)
Type `bool`	`true`, `false`
`i8`, `i16`, `i32`, `i64`, `i128`, `isize`	(all conforming signed integer values)
`u8`, `u16`, `u32`, `u64`, `u128`, `usize`	(all conforming unsigned integer values)
`f32`, `f64`	(all conforming floating point values)

Table 4-1: Rust Types and Rust Values Supported by the Schema Extension

If multiple parameter specifications exist for the same function, then they are combined using *Boolean-and* semantics, which means that *all* parameter specifications must be fulfilled for the function to be considered vulnerable. It is not possible to express *Boolean-or* semantics for a single vulnerability so that *any* parameter specification triggers the vulnerability. However, it is possible to define multiple vulnerability definitions for these cases since the tool inherently reports whether *any* defined vulnerability is identified.
The RustSec Database has been forked[2] and the vulnerability definitions are modified to

[2]Folder `masterthesis-advisory-db-fork` in submitted zip file

include the parameter specifications as identified in Chapter 3.
The changes to the vulnerability definition `RUSTSEC-2023-0044` are shown in Listing 4-1. The first parameter of the method `set_host` is vulnerable to an empty string.

```
[affected.parameters]
"openssl::x509::verify::X509VerifyParamRef::set_host" = [{parameter = 1, operator = "==", rust_type = "String", rust_value = ""}]
```

Listing 4-1: Extension to Vulnerability Definition RUSTSEC-2023-0044

The changes to the vulnerability definition `RUSTSEC-2023-0024` are shown in Listing 4-2. The second parameter of the methods `new` and `new_nid` are vulnerable to the value `Option::None`.

```
[affected.parameters]
"openssl::x509::X509Extension::new" = [{parameter = 2, operator = "==", rust_type = "Option", rust_value = "None"}]
"openssl::x509::X509Extension::new_nid" = [{parameter = 2, operator = "==", rust_type = "Option", rust_value = "None"}]
```

Listing 4-2: Extension to Vulnerability Definition RUSTSEC-2023-0024

The changes to the vulnerability definition `RUSTSEC-2022-0051` are shown in Listing 4-3. The functions `LZ4_decompress_safe` and `LZ4_decompress_continue` were identified as affected functions. The fourth parameter of these functions is vulnerable to negative numbers if the third parameter is not equal to zero as identified in Section 3.3.7.

```
[affected.functions]
"lz4_sys::LZ4_decompress_safe" = ["*"]
"lz4_sys::LZ4_decompress_safe_continue" = ["*"]

[affected.parameters]
"lz4_sys::LZ4_decompress_safe" = [{parameter = 4, operator = "<", rust_type = "i32", rust_value = "0"}, {parameter = 3, operator = "!=", rust_type = "i32", rust_value = "0"}]
"lz4_sys::LZ4_decompress_safe_continue" = [{parameter = 5, operator = "<", rust_type = "i32", rust_value = "0"}, {parameter = 4, operator = "!=", rust_type = "i32", rust_value = "0"}]
```

Listing 4-3: Extension to Vulnerability Definition RUSTSEC-2022-0051

4.3 Data-Flow Analyzer

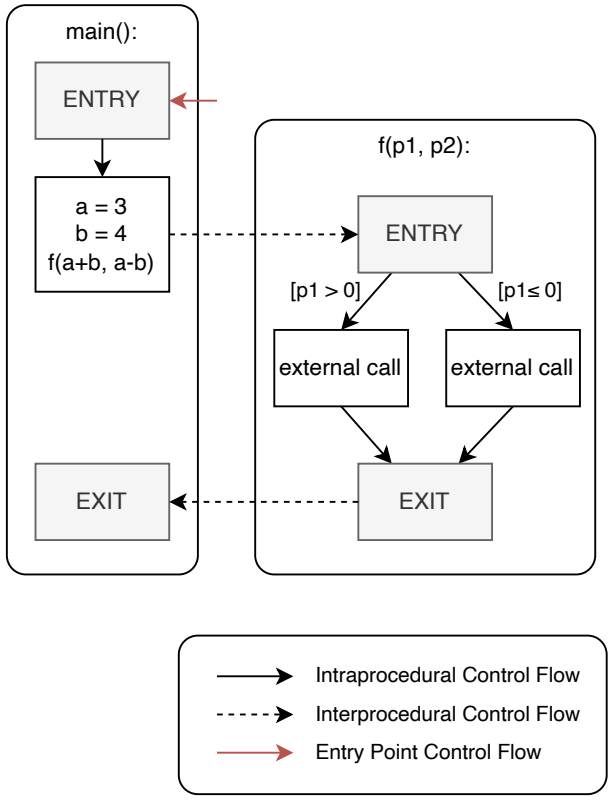
The *Data-Flow Analyzer* (as depicted in the High-Level Architecture in Figure 4-1) performs the data-flow analysis and Section 4.3.1 illustrates this analysis using an example. The analyzer is invoked on a single crate in the dependency tree such that dependencies are analyzed before their dependents. The analyzer assesses the code of the respective crate and accesses information present in the RustSec Database as well as analysis results for its crate dependencies that are already available.
The first step in the analyzer is to generate the High-Level Representation (HIR) and Mid-Level Representation (MIR) for the analyzed crate. This is done by integrating the *Data-Flow Analyzer* with the Rust Compiler, which provides access to these representations.
The next step in the analyzer is to generate the Analysis IR, which represents a supported subset of MIR and HIR, as described in Section 4.3.2.
Then, the Analysis IR is examined using a data-flow analysis that performs Abstract Interpretation, as introduced in Section 2.1.2. The abstract domain that the analysis operates on, is formally introduced in Section 4.3.3.
The data-flow analysis is performed at various levels of abstraction, from individual basic blocks, to functions, and finally all entry points of the crate. Section 4.3.4 shows an algorithm for analyzing an individual basic block by analyzing each contained statement in control flow order. Section 4.3.5 shows an algorithm for analyzing a function consisting by first the contained basic blocks in control flow order. Section 4.3.6 shows an algorithm for analyzing multiple functions that are contained in a Rust crate, starting at the entry points. Functions are analyzed before their callers in a bottom-up way. The results of each analysis are represented by summaries that describe whether the code is vulnerable at the respective level of abstraction.

4.3.1 Example of Data-Flow Analysis

This section illustrates the analysis algorithm on a simple example, to introduce the underlying ideas before the following subsections go into details.
For this, an example program is introduced along with its interprocedural control flow graph, as depicted in Figure 4-3. This example program is vulnerable, and the algorithm shows that it is vulnerable.
The algorithm works by analyzing basic blocks and functions and creating summaries for them. These summaries describe the circumstances under which the summarized code is vulnerable and, in case of a function summary, the return value of the function. In the example program presented in this section, the functions do not return values, so this part of the summary is not relevant and is omitted here.

```
1
2  fn main() {
3
4    let a = 3;
5
6    let b = 4;
7
8    f(a + b, a - b);
9
10 }
11
12 fn f(p: i32, q: i32) {
13
14   if p > 0 {
15
16     vulnerable_if_negative(q);
17
18   } else {
19
20     vulnerable();
21
22   }
23
24 }
```

(a) Rust Source Code



(b) Interprocedural Control Flow Graph

Figure 4-3: Example Program Containing Two Distinct Vulnerabilities

The example program calls the vulnerable function `vulnerable` and the vulnerable function `vulnerable_if_negative`. Let us assume that the RustSec Database contains vulnerability definitions that define these functions as vulnerable, as follows:

- The function `vulnerable` is always vulnerable.
- The function `vulnerable_if_negative(param)` is only vulnerable if its parameter is negative.

Consequently, the algorithm creates two function summaries for these functions that describe exactly these conditions under which each function is vulnerable.
The function `main` calls function `f`. Due to the bottom-up analysis order, `f` is analyzed first. Analysis starts at the level of basic blocks. The basic blocks of the example program are shown in the interprocedural control flowgraph depicted in Figure 4-3b.
Step ① of the exemplary analysis: The entry basic block of `f` does not contain any vulnerable code and thus its summary contains the vulnerability condition `false`.
Step ② of the exemplary analysis: The `if` statement produces a branch in the control flow, and the `then` block invokes the external function `vulnerable_if_negative(q)` in line 16, which has a summary of `size < 0`. The summary is applied, and `size` is replaced

by the argument `q` producing `q < 0`. The precondition for entering the basic block is `p > 0` and thus the complete basic block summary is `q < 0 && p > 0`.
Step ③ of the exemplary analysis: The basic block representing the else-block has the inverted precondition `p <= 0` and calls the function `vulnerable()` which has the summary `true`. Therefore, the basic block summary is `p <= 0 && true`, which can be shortened to `p <= 0`.
Step ④ of the exemplary analysis: In the `EXIT` basic block of function `f`, the control flow merges, and it is vulnerable if any of the two incoming basic blocks are vulnerable. The summaries are combined using the *Boolean-or* operator that produces `(p > 0 && q < 0)) || (p <= 0)`.
Step ⑤ of the exemplary analysis: The function `main` evaluates the assignments to the variable `a` in line 4 and to variable `b` in line 6. In line 8, function `f` is called. Before the call, it evaluates the first argument `a + b` to the value of `7` and the second argument `a - b` to the value of `-1`. It then calls `f(7, -1)` using the previously calculated vulnerability condition of the function `f`. Inserting the calculated argument values into the vulnerability condition of `f` produces `(7 > 0 && -1 < 0) || (7 <= 0)`, which evaluates to `true` and consequently, the call of `f` produces a vulnerability condition of `true`. As the call is located in the main function, the function `main` also gets a vulnerability condition of `true`. This is evidence that a vulnerability exists whenever the main function is called and consequently a security violation is reported for the main function.
To add more context, `time::now()` is a real-world function with a parameterless signature like the exemplary function `vulnerable()` and is vulnerable to `RUSTSEC-2020-0071`[3]. Likewise, the function `LZ4_decompress_generic` is vulnerable to `RUSTSEC-2022-0051`[4] only if a negative value is passed as a parameter, similarly to the exemplary function `vulnerable_if_negative`. Both of these vulnerabilities were discussed in Chapter 3.

[3] https://rustsec.org/advisories/RUSTSEC-2020-0071
[4] https://rustsec.org/advisories/RUSTSEC-2022-0051

4.3.2 Analysis IR

The tool operates on a subset of MIR and HIR. The subset of MIR is a result of Requirement *R1 (No support for alias sensitivity, field sensitivity or array sensitivity, and no support for dynamic function calls)*. A subset of HIR is used to support re-exported functions, as a result of Requirement *R6* (*Support re-exported functions*). For further elaborations, the combined subset will be referred to as Analysis IR and is shown in Figure 4-4.

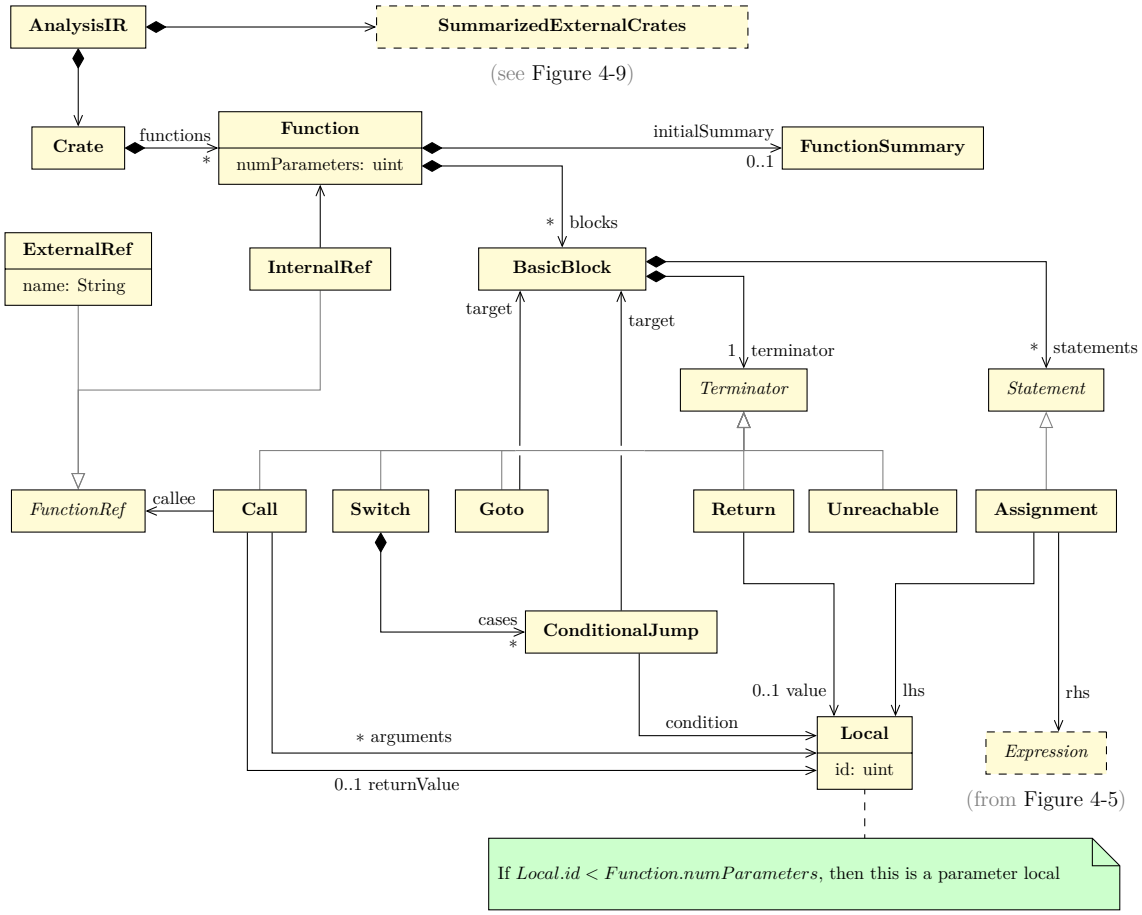


Figure 4-4: Analysis IR

Analysis IR describes the part of the code that is available to the data-flow analysis when analyzing a single crate. In order to support flows into external dependencies, the type `SummarizedExternalCrates` contains summarized data-flow information of external crates.

The program code of the analyzed crate is represented by the type `Crate`. It consists of functions, and the control flow inside these functions is modeled using a control flow graph. A basic block contains arbitrarily many regular statements and one terminator statement at its end. Analysis IR supports five kinds of terminator statements: Switch, Call, Goto, Return, and Unreachable.

The `Switch` statement in Analysis IR represents a conditional jump to one successor basic blocks. It defines one condition for each potential successor. At the MIR level, this is referred to as the `SwitchInt` instruction, which involves a discriminant operand and several immediate values to compare the operand to. In contrast, Analysis IR creates explicit comparisons for each case, allowing data-flow analysis to use these comparisons as preconditions at the start of a basic block.
The `Call` instruction is used to call an internal or external function. An internal function call refers to a function within the same crate, while an external function call refers to a function defined in an external crate or the Rust standard library via a fully qualified function name. For external functions, a summary lookup is performed based on the external summaries provided by the type `SummarizedExternalCrates`.
The `Goto` instruction is an unconditional jump to another basic block, while the `Return` instruction leaves the current function and either returns the value of a local variable or does not return a value in the case of a void function. The `Unreachable` statement represents a program point that should not normally be reached, usually terminating the program. For instance, this statement is generated by the Rust Macro `panic`.
The only regular statement supported by Analysis IR is the assignment statement. AnalysisIR does not support assignments to or from fields, as a result of Requirement *R1 (No support for alias sensitivity, field sensitivity or array sensitivity, and no support for dynamic function calls).* The assigned value is an expression that can be calculated with no side-effects. The expression model is shown in Figure 4-5. The expression model consists

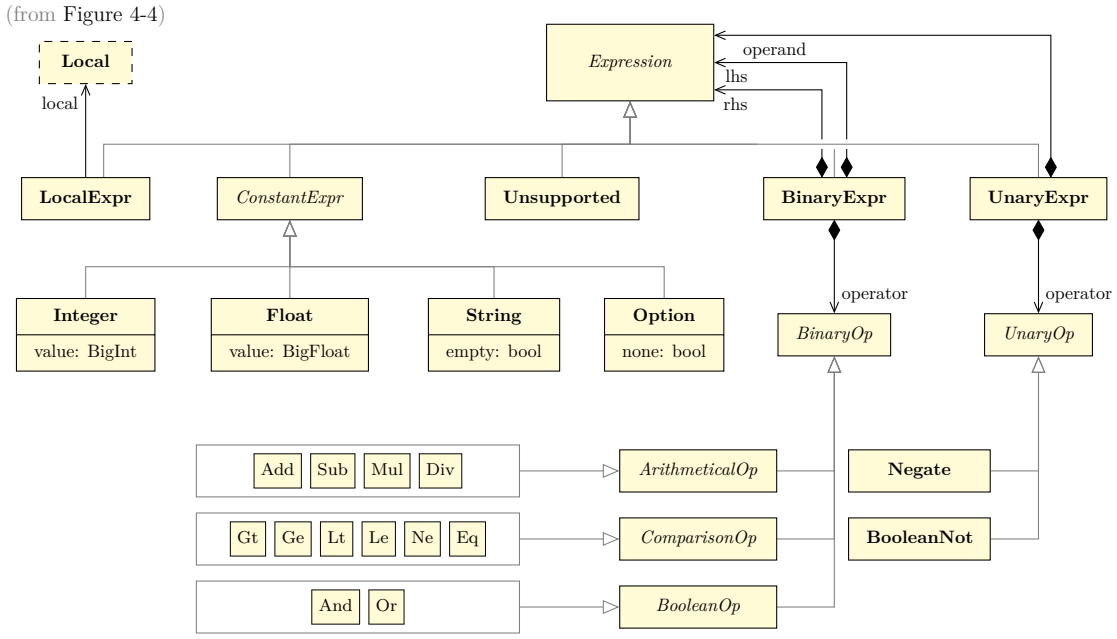


Figure 4-5: Analysis IR: Expression Model

of five kinds of expressions: a local expression, a constant, an unsupported expression, a

binary expression and a unary expression. A binary expression has two operands and a binary operator, which is either an arithmetical operator, comparison operator or boolean operator. A unary expression has one operand and a unary operator, which is either a negation or a *Boolean-not* operation. The constant expression refers to an integer, floating point, String or Option value. Boolean values are encoded as integer values with `false` being represented by the number zero and `true` by the number one, and the local expression refers to the value of a local variable. For strings, the analysis tracks if the string is empty and for the Option type, the analysis tracks if the Option represents a missing or present value (`Option::None` and `Option::Some`, respectively).

4.3.3 Abstract Domain

This subsection formally introduces the abstract domain that the analysis uses, showing that it is a bounded semi-lattice, as introduced in Section 2.1.3. It starts by introducing abstract values as tracked by the analysis and shows that they are a bounded semi-lattice. The actual abstract domain of the analysis is a tuple of abstract values with a per-element join operation, inheriting the bounded semilattice properties.
To represent these abstract value types, an expression model is introduced as shown in Figure 4-6. The abstract domain contains the lattice elements `Bottom` ($\bot$) and `Top` ($\top$).

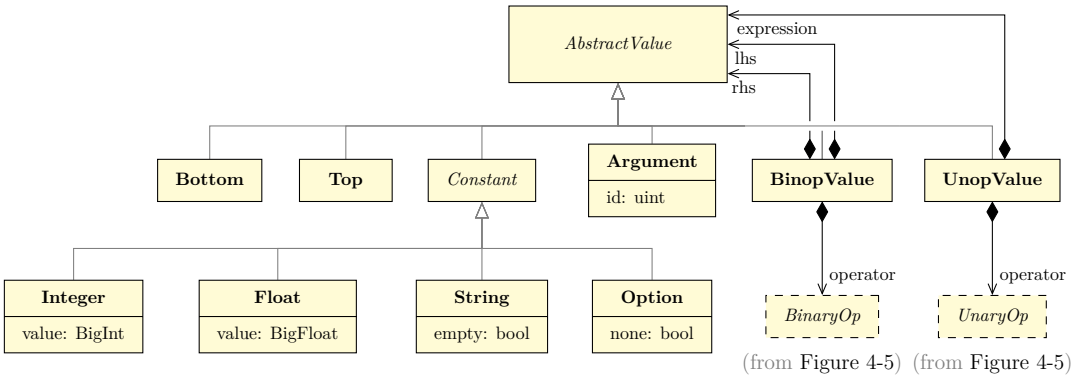


Figure 4-6: Abstract Values

A numeric value is represented by the type `Constant` supporting the same values as in Analysis IR. A formula to calculate values based on other values is represented by the expressions `UnopValue` and `BinopValue` supporting one and two operands, respectively. The operands are abstract values again to enable nesting of these expressions and expressing more complicated formulas. Parameter values are represented by the type `ArgumentExpr`, which refers to the numeric position of the parameter in the function signature. The actual value for the parameter is not known during analysis of the function, and therefore these expressions serve as a marker representing the value of the parameter.

Any expression that contains an `ArgumentExpr`, is called *symbolic* and can only be fully evaluated when concrete values are provided for all `ArgumentExpr` in the expression.
Figure 4-7 shows an excerpt of the bounded semilattice that includes all possible instances of the expression tree. The lattice is bounded by the Bottom and Top elements and the middle area of the lattice contains all constants, all argument values, all binary operations and all unary values. As there are infinitely many of these values, the diagram only shows one instance of each type. The element Bottom ($\bot$) represents the initial value for a

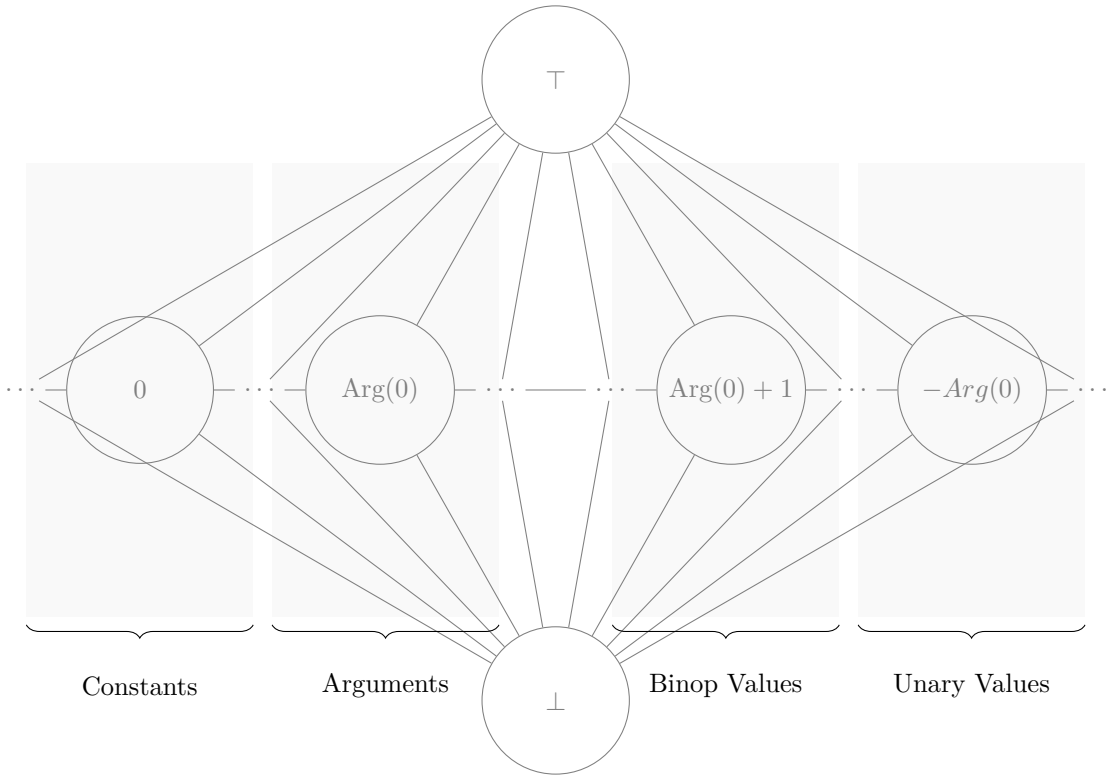


Figure 4-7: Excerpt of Bounded Semilattice

variable before a value is assigned to that variable in the program code. The element Top ($\top$) represents the state of a variable that is obtained when multiple possible values are joined. This can happen if multiple basic blocks with conflicting values for a single variable enter the same successor basic block. This situation is called a join in the control-flow graph and executes the `Join` operation on the set of values contributed by all predecessor basic blocks to retrieve a single value for that variable to use going forward.
Listing 4-4 shows the code for the functions `Join`, `Limit` and `Evaluate` that interact with abstract values. The `Join` operation combines the abstract values and returns the resulting joined abstract value. The join operation was formally introduced in Section 2.1.3 and calculates the least upper bound of the lattice shown in Figure 4-7.

For this lattice, the following semantics are sufficient:

- The value $\bot$ is the minimum value of the lattice and therefore joining any value with $\bot$ keeps the respective other value as the upper bound.
- The value $\top$ is the maximum value in the lattice, and therefore joining any value with $\top$ must keep $\top$ as the upper bound.
- In general, the join of distinct elements that are unordered with respect to the partial order $\sqsubseteq$ (in the middle area of the lattice) returns $\top$.

The `Limit` function limits the number of nodes in the expression tree to not exceed the constant value K. This is necessary to prevent expressions from becoming too complex for the analysis to handle efficiently and to prevent too much memory consumption at run-time. If the limit is exceeded, then $\top$ is returned. The function `Evaluate` evaluates parts of an expression that are not symbolic.

```
1  function Join(expressions)
2    expressions := expressions \ {⊥}
3    if expressions = ∅ then return ⊥ end if
4    if all expressions are equal then return expressions(0) end if
5    return ⊤
6  end function

8  function Limit(expression)
9    return { expression   if number of tree nodes in expression ≤ K
           { ⊤            else
10 end function

12 function Evaluate(expression)
13   evaluate the operations in each sub-expression of expression recursively
14   return Limit(⟨evaluated expression⟩)
15 end function
```

Listing 4-4: Operations on Abstract Values

Next, an algorithm for analyzing the program code on different levels of granularity is designed, starting with one basic block and finishing with analyzing a complete dependency tree.

4.3.4 Basic Block Analysis

This subsection explains the AnalyzeBasicBlock algorithm which analyzes a single basic block in Analysis IR. This algorithm gathers data for each basic block which can be used to create a function summary. For example, the function summary includes an expression tree that outlines the function return value based on the arguments and the

condition for which the function is vulnerable. The AnalyzeBasicBlock algorithm collects this information on the basic block level, as demonstrated in Listing 4-5.

```
1  function AnalyzeBasicBlock(b)
2    State ← deep clone Incoming(b)
3    L ← Locals(State)
4    for each statement in ControlFlowOrder(Statements(b)) do
5      switch statement
6        case Assignment(lhs, rhs) ⇒ L(lhs) ← ReplaceLocals(rhs, L)
7      end switch
8    end for
9    switch Terminator(b)
10     case Call(f', Args, ReturnVar) ⇒
11       L(ReturnVar) ← Evaluate(ReplaceArgs(Returns(f'), Args))
12       V ← Evaluate(ReplaceArgs(VulnerableIf(f'), Args) ∧ Precondition(State)))
13       VulnerableIf(State) ← Evaluate(VulnerableIf(State) ∨ V)
14     case Return(Var) ⇒
15       Returns(f) ← Join(Returns(f), L(Var))
16       VulnerableIf(f) ← Evaluate(VulnerableIf(f) ∨ VulnerableIf(State))
17   end switch
18   for each succ ∈ Successors(b) do
19     Outgoing(b, succ) ← { SwitchTransition(State, b, succ)   if Terminator(b) is Switch
                           { State                              else
20   end for
21 end function

23 function ReplaceArgs(Expression, Arguments)
24   switch Expression
25     case ArgumentExpr(arg) ⇒ return Arguments(arg)
26     else recursive descend
27   end switch
28 end function

30 function ReplaceLocals(Expression, L)
31   switch Expression
32     case LocalExpr(local) ⇒ return L(local)
33     else recursive descend
34   end switch
35 end function
```

Listing 4-5: Summarizing one Basic Block

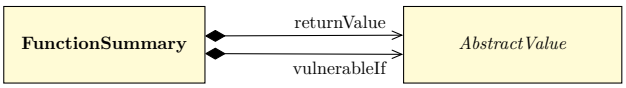
The variable *State* is used to represent the state of the basic block b, which is initialized using the incoming state $Incoming(b)$ that holds before the basic block. The algorithm then updates the state by first evaluating the assignment statements and then the terminator statement at the end of the basic block. This produces the outgoing state *Post*, which holds after the basic block and will be the incoming state for its successor basic blocks. If the terminator statement is a Switch statement, then the outgoing state can differ for each successor basic block and is thus stored for each successor basic block $succ \in Successors(b)$ as $Outgoing(b, succ)$.

The state of a basic block consists of the abstract values of the local variables represented

by $Locals(State)$ and L, the precondition $Precondition(State)$ of the basic block and the condition $VulnerableIf(State)$ that describes under which circumstances the state is vulnerable.
The values of local variables, represented by $Locals(State)$ and L, are initially set based on the incoming state, which represents the abstract values available from the preceding basic blocks. In the first `for`-loop, these values are then modified for each assignment within the basic block. In Analysis IR, an assignment to a local variable uses an expression on the right-hand side to define the value to assign. This expression can refer to other local variables. The algorithm replaces any such references to local variables in the expression with the actual value already known for them in L and evaluates the expression using the `Evaluate` function, as explained in Section 4.3.3.
The terminator statement is the last statement in the basic block and is therefore handled last. The call statement calls a function and assigns its return value to a local variable, and this assignment affects the state of the current basic block. The return statement adds any returned values to the set of values returned by the function $Returns(f)$.
The Goto and Unreachable terminators, which only affect the control flow but do not have any side effects on the current basic block; and the `Switch` terminator, which adds new preconditions to the successor basic blocks.

4.3.5 Intraprocedural Analysis

This subsection describes the algorithm AnalyzeFunction that analyzes one function in Analysis IR and creates a function summary for it.
As shown in Figure 4-8, a function summary contains two expressions. The first expression represents the formula to calculate the return value and the second expression represents the formula to calculate the condition for a vulnerability to be triggered when the function is invoked.



(from Figure 4-6)

Figure 4-8: Function Summary Model

The algorithm AnalyzeFunction is shown in Listing 4-6. It invokes the algorithm AnalyzeBasicBlock as shown in Listing 4-5 to analyze the basic blocks contained in the function. The algorithm starts by setting the incoming state for the entry basic block $Incoming(b_{entry})$ to the initial state of the function $Initial$. The initial state of the function is defined as follows: The vulnerable condition $VulnerableIf(Initial)$ is initialized with `false`, as initially no vulnerable calls are known. The $Precondition(Initial)$ is ini-

tialized with `true`, because the initial basic block is entered unconditionally. The local variables are initialized using the following convention: The first local variables represent the parameter values of the function and are therefore initialized using an ArgumentExpr refering to their respective parameter index. All other locals are initialized with $\bot$. This convention stems from the Mid-Level Representation and is used in Analysis IR as well.

```
1  function AnalyzeFunction(f)
2    p ← number of parameters of function f
3    Initial ← new empty state
4    Locals(Initial) ← (ArgumentExpr(0), ..., ArgumentExpr(p), ⊥, ..., ⊥)
5    VulnerableIf(Initial) ← false
6    Precondition(Initial) ← true
7    Incoming(b_entry) ← Initial
8    AnalyzeBasicBlock(b_entry)
9    worklist ← Successors(b_entry)
10   while worklist ≠ ∅ do
11     Remove basic block b from worklist
12     Incoming(b) ← JoinStates({Outgoing(p, b) ∀p ∈ Predecessors(b)})
13     AnalyzeBasicBlock(b)
14     worklist ← worklist ∪ {succ ∈ Successors(b) where Outgoing(b, succ) has changed }
15   end while
16   return (Returns(f), VulnerableIf(f));
17 end function

19 function SwitchTransition(State, b succ)
20   JumpCondition ← find jump condition from b to succ in Switch
21   Precondition(State) ← Evaluate(Precondition(State) ∧ JumpCondition)
22   return State
23 end function

25 function JoinStates(states)
26   State = new empty state
27   Precondition(State) ← Evaluate(Precondition(states(0)) ∨ ... ∨ Precondition(states(n))
28   VulnerableIf(State) ← Evaluate(VulnerableIf(states(0)) ∨ ... ∨ VulnerableIf(states(n))
29   Locals(State) ← JoinLocals(Locals(states(0)), ..., Locals(states(n))
30   return State
31 end function

33 function JoinLocals(locals)
34   return (Join(locals(0)(0), locals(1)(0), ...), Join(locals(0)(1), locals(1)(1), ...), ...)
35 end function
```

Listing 4-6: Analyzing one Function

The algorithm uses a worklist to refer to functions that need analysis.
Initially, the analysis starts at the entry basic block and adds all successor basic blocks to the empty worklist.
After that, the algorithm iterates over the worklist and each loop iteration takes one item from the worklist which is the next basic block b to analyze. The algorithm computes the incoming state $Incoming(b)$ for the basic block by merging the outgoing states $Outgoing(p, b)$ of all predecessors of b as assigned by the last execution of the function An-

alyzeBasicBlock. The loop continues until no successor state has to be updated anymore and a fixed-point is reached. The algorithm returns the constructed function summary consisting of the possible return values $Returns(f)$ and the vulnerable condition for the function $VulnerableIf(f)$.

4.3.6 Interprocedural Analysis

This subsection describes the algorithm AnalyzeProgram that analyzes in Analysis IR. The algorithm is shown in Listing 4-7 and analyzes a crate by analyzing the functions in the crate using the previously described algorithm AnalyzeFunction.
The analysis starts at the entry point functions of the crate. In response to Requirement *R3 (Support application programs and software libraries)*, both application programs and software libraries are supported. For an application crate the entry point function is the function `main` and for library crates, all public functions are entry points, as they are exported by the library and can be used by clients of the library.
The algorithm starts analyzing the entry point functions F_E of the analyzed crate. If a function f calls unsummarized functions F_U, those functions are summarized before f. This makes sure that the bottom-up order is adhered to. Otherwise, f is summarized using the intra-procedural algorithm described in the previous subsection.
Recursive calls (detected through a pre-calculated call graph) receive special treatment. If function f calls function g as part of a recursive cycle, then an empty function summary is stored initially, and f is analyzed based on it, and f's (preliminary) function summary is stored. Then g is analyzed based on this preliminary summary and so on, joining each iteration's summary with the previous one. The domain's bounded semi-lattice properties guarantee that this iteration reaches a fixed-point (i.e. the joined summary is eventually identical to the previous iteration's summary) in a finite amount of steps. The iteration's fixed-point is the algorithm's final summary for a given function.

```
1  function AnalyzeProgram(F_E)
2    worklist ← F_E
3    while worklist ≠ ∅ do
4      f ← remove function from worklist
5      if f calls unsummarized functions F_U without recursive cycles with f then
6        Append functions F_U to the worklist
7      else
8        Summary(f) ← AnalyzeFunction(f)
9        if Summary(f) has changed then worklist := worklist ∪ Callers(f) end if
10     end if
11   end while
12   return (Summary(f_e), ∀f_e ∈ F_E)
13 end function
```

Listing 4-7: Analyzing one Crate

4.4 Inter-Crate Analysis

As shown in the High-Level Architecture in Figure 4-1, the *Cargo Subcommand* calls the *Data-Flow Analyzer* on each dependency tree node to perform a data-flow analysis across the boundaries of single crates in response to Requirement *R7 (Analyze across dependency boundaries)* and is referred to as *inter-crate analysis* in further elaborations.
The inter-crate analysis uses summaries for external crates as represented by the type `SummarizedExternalCrates` shown in Figure 4-9. These external summaries are used to

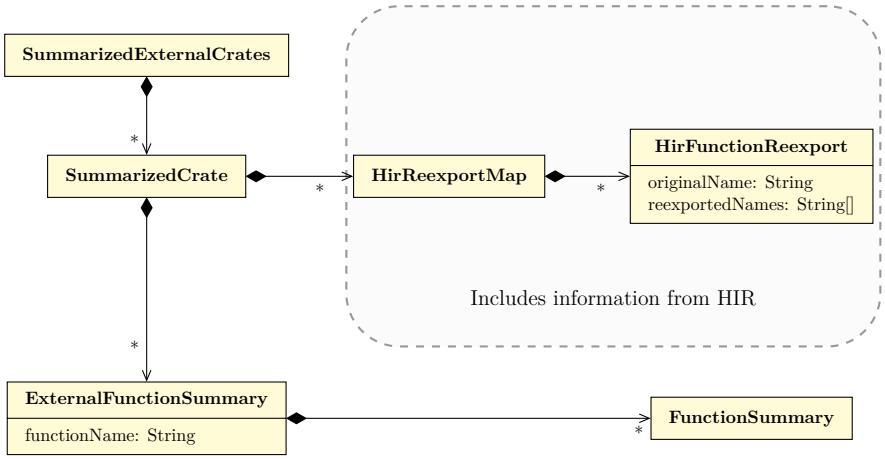


Figure 4-9: AnalysisIR: Summarized External Crates

resolve calls into external crates, as mentioned in Section 4.3.4.
The result of analyzing one crate is represented by the type `SummarizedCrate` which contains function summaries identified by their fully qualified function names. These results are persisted and are available to be used in the analysis of dependent crates.
In Rust, functions can be re-exported for other modules using the keyword `pub use`. In response to Requirement *R6 (Support re-exported functions)* all instances of `pub use` in the code are identified in the HIR and the `HirReexportMap` is generated, which lists re-exported functions as identified by their fully qualified name. A re-exported function is represented by the type `HirFunctionReexport`, which contains the original name and the reexported name of the function.
The algorithm AnalyzeDependencyTree as shown in Listing 4-8 calls the *Data-Flow Analyzer* for each crate in the dependency tree.

```
1 function AnalyzeDependencyTree(crates)
2   for each crate in ReverseTopologicalOrder(crates) do
3     load summaries ∀dep ∈ Dependencies(crate)
4     AnalyzeProgram(Entrypoints(crate))
5     persist generated summaries
6   end for
7 end function
```

Listing 4-8: Analyzing Dependency Tree

This algorithm iterates over all crates in the dependency tree and creates summaries for their entry point functions. The iteration happens such that dependencies are analyzed before dependants. Such an ordering is guaranteed to exist because Cargo prevents recursive dependencies between crates, and the partial order defined by dependencies is therefore guaranteed to be cycle-free.
The crate that is associated with the vulnerability is part of the dependency tree, and therefore it is also analyzed. This allows to identify other affected functions in the vulnerable crate that were left out in an incomplete vulnerability specification, which is required to satisfy Requirement *R2 (Infer transitively affected functions in vulnerable crate)*.
The loop can be parallelized as long as it is guaranteed that dependencies are analyzed before dependents. This makes it possible to execute each crate analysis in a different process, which maps well to the compilation model used by Cargo and the Rust Compiler.

4.5 Report Findings

The last two steps of the High-Level Architecture in depicted in Figure 4-1 are to collect the summaries and to report findings.
After all crates are summarized by the *Data-Flow Analyzer*, the *Cargo Subcommand* collects the summaries and opens the summary for the top-level crate (on which the analysis was originally started).
The vulnerability condition for the entry points is used to decide if a finding is reported. For the values $\bot$ and `false`, no vulnerability is reported, and for the values $\top$ and `true`, a vulnerability is reported.
In the context of a Rust application program, the function `main` is the only entry point. It has no function parameters, and therefore the vulnerability condition does not refer to parameter values and can be evaluated to either of the four mentioned values.
In the context of Rust software libraries, the entry-point functions are exported functions of the library as they can be called by dependents of the library. As regular functions can have parameters, the vulnerability condition can depend on the parameter values and cannot be evaluated but remains symbolic. If the vulnerability condition of an exported library function remains symbolic, a vulnerability is reported. This is significant because the data-flow analysis assesses the possibility of a vulnerability occurring with any parameter value of the function. Consequently, it must operate under the assumption that the condition could evaluate to `true` for specific parameter values.

The resulting rules for reporting findings at entry points are:

- The value $\bot$ (bottom) and `false` report no vulnerability.
- The value $\top$ (top) and `true` report a vulnerability.
- Symbolic values that reference parameters report a vulnerability.

A limitation of the described design is that the format of the summaries does not enable one to report the data-flow or call stack that leads to a vulnerable function call, as this information are not included in the function summary. It is estimated that the format of such data must be carefully designed to avoid a huge increase in the complexity of the summaries.

5 Implementation

This chapter illustrates the implementation of the static code analysis tool for automatic vulnerability audits based on the design presented in Chapter 4. Section 5.1 outlines the implementation of the *Data-Flow Analyzer* as shown in the high-level architecture depicted in Figure 4-1. Section 5.2 describes the implementation of the *Cargo Subcommand* also shown in the high-level architecture. Section 5.3 outlines the implementation of the persistable summary store, which is required to pass analysis results from one invocation of the analyzer to another invocation of the analyzer and back to the *Cargo Subcommand*. Section 5.4 outlines the implementation of the Analysis IR data structures and discusses performance considerations as well as functionality to print the IR textually. Section 5.5 concludes this chapter and highlights insights gained during implementation phase of this thesis.

5.1 Implementation of the Data-Flow Analyzer "flowcheck"

The Data-Flow Analyzer uses the Rust compiler as a library and adds a data-flow analysis after the internal program representations are generated. This section describes the implementation of this compiler extension. Listing 5-1 shows code that uses the Rust compiler as a library.

```
1  #![feature(rustc_private)]
2  extern crate rustc_driver;
3  extern crate rustc_hir;
4  extern crate rustc_middle;
5  extern crate rustc_data_structures;
6
7  struct FlowcheckCallbacks;
8
9  fn main() {
10   let mut callbacks = FlowcheckCallbacks;
11
12   // 2) Run Compiler
13   let exit_code = rustc_driver::catch_with_exit_code(|| {
14     let args: Vec<String> = std::env::args().into_iter().collect();
15     rustc_driver::RunCompiler::new(&args, &mut callbacks).run()
16   });
17
18   // 3) Exit process with exit code from compiler
19   std::process::exit(exit_code);
20 }
```

Listing 5-1: Running the Compiler with Callbacks

The Rust Compiler consists of crates that contribute a well-defined part of the compiler's functionality: The crate `rustc_driver` contributes the Compilation Driver, which manages the compilation process and defines the behavior of the `rustc` Command Line

Interface (CLI) programm. The crate `rustc_hir` contributes the High-Level Intermediate Representation (HIR) and the crate `rustc_middle` contributes the Mid-Level Intermediate Representation (MIR).
The code in Listing 5-1 runs the compiler driver using the crate `rustc_driver`. Rust provides a dedicated way to use parts of the Rust compiler as dependencies. This method involves two changes to the top-level module of the client crate. First, the annotation `#![feature(rustc_private)]` on line 1 acquires access to the Rust compiler internals. Second, the individual crates that are required as dependencies are explicitly defined using the keyword `extern crate`, as used in lines 2–5. This keyword does not allow to specify the version of the dependency and therefore it makes the implementation dependent on the specific version of the Rust toolchain that is installed. To ensure that the correct Rust toolchain is used to compile the extension, the required toolchain is declared in the file `rust-toolchain.toml` as shown in Listing 5-2.

```
1 [toolchain]
2 channel = "nightly-2023-04-16"
3 components = [ "clippy", "rustfmt", "rustc-dev", "rust-src", "rust-std", "llvm-
      tools-preview" ]
```

Listing 5-2: Toolchain File Defines Required Rust Tools

The `rustc_driver` crate provides the trait[1] `Callbacks` that allows a compiler extension to contribute functionality, which is then invoked by the compiler driver at specific points in time during the lifecycle of the compilation process. Listing 5-3 shows the definition of the trait `Callbacks`,

```
1  pub trait Callbacks {
2    fn config(&mut self, _config: &mut interface::Config) {
3    }
4
5    fn after_parsing<'tcx>(
6      &mut self, _compiler: &interface::Compiler, _queries: &'tcx Queries<'tcx>
7    ) -> Compilation {
8      Compilation::Continue
9    }
10
11   fn after_expansion<'tcx>(
12     &mut self, _compiler: &interface::Compiler, _queries: &'tcx Queries<'tcx>
13   ) -> Compilation {
14     Compilation::Continue
15   }
16
17   fn after_analysis<'tcx>(
18     &mut self, _compiler: &interface::Compiler, _queries: &'tcx Queries<'tcx>
19   ) -> Compilation {
20     Compilation::Continue
21   }
22 }
```

[1]In Rust, Traits define abstract methods to be implemented by other types; similar to Java Interfaces.

Listing 5-3: Trait Callbacks of the Rust Compiler

which also provides default implementations for each trait methods. The callback method `config` is invoked before creating the compiler instance and allows to modify the internal configuration of the compiler. The `after_parsing` callback is called after the source code is parsed, gives access to the parsed data-structures and allows to return a value that decides whether to continue or stop the compilation process. Similarly, the `after_expansion` and `after_analysis` callback methods are called after macro expansion and after program analysis, respectively and also allow to stop the compilation process early.

The callbacks implemented by the type `FlowcheckCallbacks` are shown in Listing 5-4.

```
22 impl rustc_driver::Callbacks for FlowcheckCallbacks
23 {
24   fn config(&mut self, config: &mut rustc_interface::interface::Config) {
25     // Set config options
26     config.opts.unstable_opts.always_encode_mir = true;
27     config.opts.maybe_sysroot = find_sysroot(); // runs `rustc --print=sysroot`
28     config.opts.debug_assertions = false; // disables overflow-checks (!)
29     config.opts.unstable_opts.mir_opt_level = Some(0);
30     config.opts.cg.panic = Some(rustc_target::spec::PanicStrategy::Abort);
31   }
32
33   fn after_analysis<'tcx>(
34     &mut self,
35     compiler: &rustc_interface::interface::Compiler,
36     queries: &'tcx rustc_interface::Queries<'tcx>
37   ) -> rustc_driver::Compilation {
38
39     // Abort if there are errors in the session
40     compiler.session().abort_if_errors();
41
42     // Get access to the global context (tcx)
43     queries.global_ctxt().unwrap().enter(|tcx| {
44
45       let hir = tcx.hir();
46       let mir = tcx.instance_mir(/*...*/);
47
48       // [analysis on HIR and MIR]
49
50     }
51
52     // Always continue to generate code so that artifacts of dependencies
53     // are built before being accessed by dependents
54     rustc_driver::Compilation::Continue
55   }
56 }
```

Listing 5-4: Implementation of Callbacks of Flowcheck

The method `config` initializes the configuration of the Rust compiler. It specifies to always generate MIR, defines the `sysroot` directory of the Rust Toolchain, disables debug assertions, selects the MIR optimization level zero and sets the panic strategy `PanicStrategy::Abort`. The debug assertions are disabled to disable overflow checks when performing numeric operations on primitive numeric values, which makes the generated MIR code simpler to analyze. The panic strategy is changed because the default panic strategy `PanicStrategy::Unwind` throws an exception whenever a panic is triggered and generates instructions specific to exception-handling in the MIR. This is an implicit control flow, which is currently not supported by the analysis. In contrast, the selected panic strategy `PanicStrategy::Abort` aborts the program whenever a panic is triggered (e.g. when an internal invariant of the Rust language is violated) by generating the MIR statement `Unreachable`, which is supported by the analysis.
The method `after_analysis` is called after MIR code is generated. It is implemented by the compiler extension to execute the data-flow analysis by first acquiring access to the global context containing the generated MIR code and then executing the analysis on that generated MIR code. The method always returns `Compilation::Continue` to ensure that the complete compilation process is executed and compiled artifacts are created. This is a requirement for compiling a dependency tree because the Rust compiler relies on compiled artifacts being present for all dependencies of the currently compiled project.
In summary, it was possible to extend the Rust compiler from the outside and to use it as a library by interfacing with the compiler directly using the keyword `extern crate`. This dedicated method for dependency management diverges from the regular way to manage dependencies in Rust: Crate dependencies are usually managed by the official Package Manager Cargo. Here, the project configuration file `Cargo.toml` specifies the dependencies of the project, and Cargo takes care of downloading, compiling, and linking the dependencies. This method cannot be used to use crates from the Rust compiler as dependencies because the Rust compiler depends on other parts of the Rust toolchain, which are not managed using crates.
Still, it was possible to implement the Data-Flow Analyzer as a separate crate, which has several benefits: The compiler extension does not need to be directly included in the Rust compiler source code tree and it simplifies the installation of the tool as third-party software. Still, a specific Rust toolchain must be installed to execute the tool.
In the future, the Stable MIR (SMIR) format, which was briefly introduced in Section 2.5.5, might provide a stable serialization format that is common across different versions of the Rust toolchain and mitigate this drawback.

5.2 Implementation of the Cargo Subcommand "cargo-flowcheck"

The design describes an extension to the Package Manager Cargo that enables the inter-crate analysis. Cargo has the capability to compile a Rust crate including its dependencies. The Cargo Subcommand "cargo-flowcheck" uses and refines this capability by delegating invocations of the Rust Compiler to the extended Rust Compiler "flowcheck", which also performs the data-flow analysis on that crate.
This is possible by implementing the trait `cargo::core::ops::Executor`, which is used to invoke the Rust compiler. The main function of the Cargo Subcommand "cargo-flowcheck" is shown in Listing 5-5.

```
1  struct FlowcheckExecutor {
2    lockfile: PathBuf
3  }
4
5  fn main() {
6    // Parse command line arguments
7    let args: &Args = parse_command_line_arguments();
8
9    // Configure cargo using arguments
10   let mut config = cargo::Config::default().unwrap();
11   config.configure(args);
12
13   // Activate "cargo check" style compilation
14   let workspace = args.workspace(&config).unwrap();
15   let mut compile_opts = args.compile_options(
16     &config,
17     cargo::core::compiler::CompileMode::Check { test: false },
18     Some(&workspace),
19     cargo::util::command_prelude::ProfileChecking::Custom
20   ).unwrap();
21
22   // Open existing lockfile or generate new lockfile
23   let lockfile = workspace.root().join("Cargo.lock");
24   if !lockfile.exists() { cargo::ops::generate_lockfile(&workspace).unwrap(); }
25
26   // Compile with custom executor
27   let exec1: Arc<dyn Executor> = Arc::new(FlowcheckExecutor { lockfile });
28   cargo::ops::compile_with_exec(&workspace, &compile_opts, &exec1).unwrap();
29
30   // Load persisted summaries and display results
31   display_results();
32 }
```

Listing 5-5: Main function of the Cargo Subcommand "cargo-flowcheck"

The main function begins by parsing the command line arguments using the `clap`[2] parser.

[2] https://crates.io/crates/clap

The command-line argument `--json` is defined to export the generated analysis results into a JSON file for manual inspection or automatic evaluation. The command-line argument `--target` allows to define the target platform to use for the compilation. Rust supports conditional compilation for a specific platform and platform specific code might only be vulnerable on that specific platform. In such cases, the vulnerability can only be detected when compiling for the vulnerable platform. This functionality is required by Requirement *R5 (Support analyzing platform-specific Rust code)*.
In line 14, Cargo is configured to use the check mode (`CompileMode::Check`). This mode is usually used to only perform analyses on the source code. In line 23, the path to the lockfile required for the dependency analysis is acquired and the lockfile is auto-generated if it does not exist. In line 27, a FlowcheckExecutor is created and in line 28, the compilation process is started using the custom executor.
Listing 5-6 shows the custom executor for the Cargo Subcommand "cargo-flowcheck". It implements the methods `exec` and `force_rebuild`.

```
34 impl cargo::core::compiler::Executor for FlowcheckExecutor {
35   fn exec(
36     &self,
37     cmd: &ProcessBuilder,
38     id: PackageId,
39     target: &Target,
40     mode: CompileMode,
41     on_stdout_line: &mut dyn FnMut(&str) -> CargoResult<()>,
42     on_stderr_line: &mut dyn FnMut(&str) -> CargoResult<()>,
43   ) -> CargoResult<()> {
44
45     // Construct new process builder
46     let mut cmd = cmd.clone();
47     cmd.program("flowcheck");
48     cmd.args(&["--cap-lints", "allow"]);
49     cmd.env("FLOWCHECK_LOCKFILE", &self.lockfile);
50
51     // Execute Command by delegating to default executor
52     DefaultExecutor.exec(&cmd, id, target, mode, on_stdout_line, on_stderr_line)
53   }
54
55   fn force_rebuild(&self, _unit: &Unit) -> bool {
56     true // No fingerprint => always reconstruct all summaries
57   }
58 }
```

Listing 5-6: Custom Executor for the Cargo Subcommand "cargo-flowcheck"

The method `exec` executes the Data-Flow Analyzer "flowcheck" and passes it additional arguments and environment variables. First, it clones the immutable process builder in line 46 to get writable access and make changes. The program name is set to `flowcheck` and the additional argument `--cap-lints allow` is set. This setting prevents the com-

piler from handling linting errors as fatal errors, which allows a data-flow analysis to be performed even if there are linting warnings.
The environment variable `FLOWCHECK_LOCKFILE` passes the path to the lock file of the analyzed project, which is necessary to perform the dependency analysis in "flowcheck". Passing values via environment variable has the benefit that the command-line interface of the Rust compiler does not need to be changed.
The method `force_rebuild` always returns `true` to invalidate the file system-based cache of compilation artifacts created by Cargo. The reason for this is a safety measure and has to do with the way the cache operates. Cargo creates a fingerprint for the compiled code based on properties such as version numbers and modification dates, to invalidate the cache once any code contributing to the compilation was modified. The fingerprint does not include information about the analysis results generated by flowcheck and thus Cargo decides if a compilation is necessary only based on the compilation artifacts. In case no analysis results are present, but compilation artifacts are present, then flowcheck would not be called, and the respective results would be missing.
The tool does not use the regular `target` folder to persist compilation artifacts, to prevent interference with regular Cargo builds. This is mainly because the tool sets compiler flags that are specifically intended for static analysis. Persisting compilation artifacts that were generated with these flags can result in the problem that the regular Cargo picks them up in a regular compilation of the code that is performed afterwards. In order to circumvent this problem, the tool uses a temporary directory as target folder by default, and the command line switch `--target` can be used to set a specific target folder to enable debugging uses cases.

5.3 Implementation of the Persistable Summary Store

The Cargo Subcommand `cargo-flowcheck` supports analyzing crates with dependencies, but the compiler extension flowcheck only analyzes a single crate. Therefore, if a crate has crate dependencies, flowcheck is invoked multiple times, once for each crate in the dependency tree. The analysis results of each flowcheck execution are then persisted on the disk and are picked up by later invocations of flowcheck when dependents are analyzed. To persist these analysis results on the disk, the serialization framework *serde*[3] is used, which enables serializing and deserializing Rust data structures by annotating them with the annotations `[derive(Serialize, Deserialize)]`. Listing 5-7 shows the data structures of the persistable summary store. The type `SummaryFile` is a persistable datastructure and contains all data that are passed between analysis invocations.
The `SummaryFile` contains the version of the file format to support the detection of

[3] https://serde.rs/

summary files that are inconsistent with an evolved version of the schema. The name of the crate that is summarized is also included to prevent applying summaries to inappropriate crates. The `PersistedSummaryStore` is included in the file and stores the summaries of all exported functions. For an application program, the summary of the `main` function is persisted, and for a library, summaries for all public functions are persisted.
The re-export map contains the names of the re-exported items and allows the data-flow analysis to analyze into functions that are re-exported using the keyword `pub use`[4] and are therefore present in several modules via different fully qualified names as described in Section 4.4.

```
1  #[derive(Serialize, Deserialize)]
2  pub struct SummaryFile {
3      pub file_format_version: String,
4      pub crate_info: CrateInfo,
5      pub store: PersistableSummaryStore,
6  }
7
8  #[derive(Serialize, Deserialize)]
9  pub struct PersistableSummaryStore {
10     pub exported_functions: BTreeMap<String, PersistableFunctionSummary>,
11     pub reexport_map: BTreeMap<String, Vec<String>>,
12 }
```

Listing 5-7: Data structures for the Persistable Summary Store

5.4 Analysis IR

The Analysis IR is implemented using Rust structs and enumerations, and an excerpt is shown in Listing 5-8. Rust Enumerations can have different attributes for each variant, which makes them a candidate for modeling class hierarchies. In classical object-oriented programming languages, inheritance is the predominant feature for implementing class hierarchies. Classical inheritance enables one to extend classes that are not defined in the same project. This means that it is not possible to know all subclasses statically at compile time, and method calls on the class need to use dynamic dispatch to support implementations in external subclasses. In contrast, Rust enumerations list every variant explicitly in the definition of the enumeration, similar to sealed classes in Kotlin. Using enumerations, a static dispatch can be performed, which is more efficient at run-time. One potential drawback of this technique is that it is not possible to extend the IR from the outside, but because the IR serves as an internal contract for several parts of the analysis pipeline and is not visible outside of the tool, this is not considered a problem.

[4] https://doc.rust-lang.org/nightly/rustdoc/write-documentation/re-exports.html

```
pub struct Program {
    pub functions: Vec<Function>,
    pub entry_points: BitSet,
}
pub struct Function {
    pub basic_blocks: Vec<BasicBlock>,
    pub num_parameters: usize,
    pub num_locals: usize,
    pub fqn: StringSymbol,
    pub initial_summary: Option<FunctionSummary>,
}
pub struct BasicBlock {
    pub statements: Vec<Statement>,
    pub terminator: Terminator,
}
pub enum Statement {
    Assign { lhs: LocalIndex, rhs: Expression },
}
pub enum Terminator {
    Goto { target: BasicBlockIndex },
    Switch { conditional_jumps: Vec<ConditionalJump> },
    Call {
        function: FunctionReference,
        arguments: Vec<LocalIndex>,
        return_value_local: Option<LocalIndex>,
        goto_block: Option<BasicBlockIndex>
    },
    Return { value: Option<LocalIndex> },
    Unreachable,
}
pub struct ConditionalJump {
    pub condition: LocalIndex,
    pub target: BasicBlockIndex
}
```

Listing 5-8: Excerpt of the Analysis IR data structures

The name of a function is stored in the attribute `fqn`, as a fully qualified name using the type `StringSymbol`. This symbol is an index into a deduplicated symbol table storing interned strings. This design has been shown to reduce the memory footprint of an analysis (Kawachiya, Ogata, & Onodera, 2008) and can also lead to performance gains, since functions are contained in a vector that is iterated by the hot part of the algorithm. In this scenario, packing the data in the vector more densly allows for better spatial cache locality, which can lead to less cache misses when reading from the vector often.
The Analysis IR supports printing to a textual syntax to enable debugging use cases. Figure 5-1 shows the exemplary program that was first introduced in Section 4.3.1 as well as the generated textual representation of the code in the Analysis IR.

```
1  fn main() {
2
3    let a = 3;
4
5    let b = 4;
6
7    f(a + b, a - b);
8
9  }
10
11
12 fn f(p1: i32, p2: i32) -> {
13
14   if p1 > 0 {
15
16     vulnerable_if_negative(p2);
17
18   } else {
19
20     vulnerable();
21
22   }
23
24 }
```

(a) Rust Source Code

```
1  fn f0() { // entry point
2    bb0 {
3      $1 = 3;
4      $2 = 4;
5      $4 = $1 + $2;
6      $5 = $1 - $2;
7      $3 = f1($4, $5);
8      goto bb1;
9    }
10   bb1 { return; }
11 }
12 fn f1($1, $2) {
13   bb0 {
14     $3 = $1 > 0;
15     $8 = 0 == $3;
16     $9 = 0 != $3;
17     switch bb2 if $8, bb1 if $9;
18   }
19   bb1 { f2($2); goto bb4; }
20   bb2 { f3(); goto bb4; }
21   bb4 { return; }
22 }
23 extern fn f2($1);
24 extern fn f3();
```

(b) Analysis IR

Figure 5-1: Example Program and Corresponding Analysis IR

5.5 Conclusion

The implementation of the developed tool consists of the Data-Flow Analyzer `flowcheck`, which extends the Rust compiler as well as the Cargo Subcommand `cargo-flowcheck`, which extends the build tool Cargo. By extending these existing components, it was possible to reuse functionality present in the Rust ecosystem, to be compatible with existing technology as well as reduce the overall implementation work. The compilation cache of Cargo has been disabled to prevent relying on invalid earler analysis results. This results in the complete recompilation and re-analysis of the source code each time the tool is executed but is was estimated to be crucial to prevent unintended interference with the regular Cargo builds. To support caching of these results, the fingerprinting system of Cargo could be further adapted to include data about the persistable summary store. The implementation of the Analysis IR employs best practices of software engineering and uses internal representations that are best suitable for the access patterns that are common in a static analysis tool.

6 Evaluation

In this chapter, the developed tool cargo-flowcheck for automatic vulnerability audits is evaluated. The experimental setup used throughout the evaluation is described in Section 6.1. In Section 6.2, a microbenchmark is introduced that was developed in the context of this thesis to test cargo-flowcheck's support for individual features of the Rust language, and the results are discussed. In Section 6.3, the support of the tool to identify vulnerabilities with affected parameters is evaluated. In Section 6.4, the tool is executed on a larger set of published Rust programs to identify findings and measure run-time performance. In Section 6.5 possible threats to the validity of these results are discussed. In Section 6.6, reproducibility of the results presented in this evaluation is addressed. In Section 6.7, the key findings and limitations discovered in the evaluation are concluded.

6.1 Experimental Setup

To increase the reproducibility of the evaluation, the setup comprising its hardware and software components is described in this section. The evaluation was carried out on a MacBook M1 Pro Model 18.1 machine with the following system specifications and analysis parametrization:

- CPU: 10 cores, 3.2 GHz
- RAM: 16 GB
- Architecture: ARM (64-Bit)
- Operating System: Mac OS Ventura 13.6
- Rust Nightly Toolchain, version `nightly-2023-04-16`
- Parameter $K = 30$ (Complexity Limitation)

The Rust Nightly Toolchain, version `nightly-2023-04-16`, was used to carry out the evaluation. The nightly version was selected because it is required by the developed tool. The developed tool depends exactly on this version of the Rust toolchain because it accesses the internal data structures of the Rust Compiler in version `nightly-2023-04-16` and thus requires this version of the toolchain to be used. The value for the parameter K that limits the complexity of expressions as introduced in Section 4.3.3 is provided here to increase the reproducibility of the evaluation. The value was chosen as a compromise between performance and precision. Identifying the optimal value for K was beyond the scope of this thesis and can be studied in future work. However, by randomly selecting some sample programs, we achieved reasonably precise results with $K = 30$.

6.2 Evaluation of Support for Rust Language Features

It was deliberately decided that the tool focuses on a subset of Rust's Mid-Level Intermediate Representation (MIR), and therefore it is expected that not all language features of the Rust language are supported. It is not straightforward to predict in advance which language features will be supported by the tool, as the MIR is created by transforming the Rust source code into an internal representation, and multiple language features may be mapped to the same patterns in the MIR. Therefore, a microbenchmark consisting of test cases for individual Rust language features was developed in the context of this thesis to test cargo-flowcheck's support for these features on the Rust source code level.
The supported Rust language features of the tool are evaluated using a microbenchmark[1] that was developed in the context of this thesis and consists of a set of positive and negative test cases. The Rust Book[2] was used to infer a set of 44 language features that were used to build the microbenchmark testing these features, and for each language feature represented in the microbenchmark there are multiple tests for assessing different aspects of that feature. In total, the microbenchmark contains 178 test cases.
The microbenchmark is designed so that for each feature tested there are both positive and negative tests. The positive tests expect to find a vulnerability, and the negative tests expect to find no vulnerability. This is important because unsupported language features might result in either a false positive or a false negative test. Thus, both the positive test and the negative test must pass in order for a language feature to be marked as supported. Figure 6-1 shows one pair of positive and negative tests that assess the support to assign a value to a mutable variable. The functions `assert_vulnerable` and `assert_not_vulnerable` are provided by the microbenchmark and are used to define a

```
1  #[test]
2  fn test_positive_variable_mutable() {
3    assert_vulnerable(quote! {
4      fn main() {
5        let mut a = false;
6        a = true;
7        vulnerable_if(a);
8      }
9    });
10 }
```

(a) Positive Test Asserts Vulnerability

```
1  #[test]
2  fn test_negative_variable_mutable() {
3    assert_not_vulnerable(quote! {
4      fn main() {
5        let mut a = true;
6        a = false;
7        vulnerable_if(a);
8      }
9    });
10 }
```

(b) Negative Test Asserts No Vulnerability

Figure 6-1: Testing the Assignment to a Mutable Variable

[1] Folder `masterthesis-implementation/microbenchmark` in submitted zip file
[2] https://doc.rust-lang.org/book

positive or negative test, respectively. The test code uses special functions `vulnerable` and `vulnerable_if`. The microbenchmark provides summaries that define the function behavior: The function `vulnerable` triggers a vulnerability unconditionally and the function `vulnerable_if` receives a Boolean argument and triggers a vulnerability if the argument is `true`.
This setup allows to calculate the number of true positives, true negatives, false positives, and false negatives. These numbers are then used to calculate the precision and recall of the results.

6.2.1 Results of Microbenchmark

The results of the microbenchmark are shown in Table 6-1. There are only 16 of all 44 language features, which passed all tests. This is quite a low number. For each language feature, the column `True Positives` shows the number of positive tests that passed, as well as the total number of positive tests, and likewise the column `True Negatives` shows the number of negative tests that passed and the total number of negative tests. Passing positive tests are counted as true positives, and passing negative tests are counted as true negatives. Failing negative tests are counted as a false positive result, and failing positive tests are counted as a false negative result. In total, there are 78 true positives, 54 true negatives, 36 false positives, and 10 false negatives. The calculated precision is 68.4% and the calculated recall is 88.6%.
The microbenchmark shows that 28 of 44 language features are not fully supported. There are seven reasons that have been identified as the cause of this, which will be discussed next.
The first reason is that the analysis lacks field sensitivity, array sensitivity, and alias sensitivity. This causes the test cases for the Slice Type, Tuple Type, Arrays, Structs and References to fail.
The second reason is that the analysis only supports a subset of Rust Types. The Character Type, Enum Types, Result Type and Function Pointers are not supported in the abstract domain, and therefore the corresponding tests fail. To support Function Pointers, both the abstract domain and call handling need to be extended.
The third reason is that Boolean Operators are transformed into multiple basic blocks by the Rust Compiler, which are harder to analyze. According to the Rust Book[3], the Boolean Operators && and || have short-circuiting semantics and the right-hand side of the operator is conditionally evaluated depending on the value of the left-hand side. The Rust Compiler generates a code pattern with multiple basic blocks. The operator value is then calculated using the switch instruction, which joins the results of those basic blocks.

[3] https://doc.rust-lang.org/book/appendix-02-operators.html

If the values differ, then the analysis over-approximates, which leads to a false positive finding in the microbenchmark.
The fourth reason is that a model of the Rust Standard Library is missing. The Rust Standard Library contains additional data structures, like Collection Types, Iterators and Smart Pointers. The corresponding tests fail because the behavior of these data structures

Tested Language Feature	True Positives	True Negatives	Passed all tests
Variables	3/3	3/3	✓
Constants	2/2	0/2	✗
Integer Overflow	0/1	1/1	✗
Numeric Type Casts	2/2	2/2	✓
Arithmetic Operators	6/6	6/6	✓
Boolean Operators	6/6	4/6	✗
Bitwise Operators	3/4	0/4	✗
Comparison Operators	7/7	7/7	✓
Boolean Type	2/2	2/2	✓
Character Type	1/1	0/1	✗
String Literals	2/2	0/2	✗
Tuples	1/1	0/1	✗
Arrays	1/1	0/1	✗
Block Expression	1/1	1/1	✓
Parameter Passing	1/1	1/1	✓
Comments	1/1	1/1	✓
If Statement	1/1	1/1	✓
If Expression	4/4	2/4	✗
Loops	4/4	4/6	✗
Slice Type	0/1	1/1	✗
References	1/2	1/2	✗
Structs	1/1	0/1	✗
Methods	2/2	2/2	✓
Enum Types	0/1	1/1	✗
Match Expression	2/2	1/2	✗
Match Statement	1/1	1/1	✓
Option Type	2/2	0/2	✗
Modules	1/1	1/1	✓
Collection Types	3/3	0/3	✗
Panic	1/1	1/1	✓
Result Type	0/3	3/3	✗
Traits	1/2	2/2	✗
Iterators	1/1	0/1	✗
Smart Pointers	4/4	0/4	✗
Deref Trait	1/1	0/1	✗
Trait Objects, Dynamic Calls	2/2	1/2	✗
Pattern Matching	1/1	0/1	✗
Unsafe Block	1/1	1/1	✓
Pointers	1/1	0/1	✗
Type Aliases	1/1	1/1	✓
Function Pointers	0/1	1/1	✗
Return Closure from Function	1/1	0/1	✗
Macros	1/1	1/1	✓

Table 6-1: Supported Rust Language Features

are not modeled. of the Rust Standard Library is missing.
The fifth reason is that Rust Constants are not explicitly supported by the analysis. In MIR, constants are handled as global entities, even if they are defined within a function. To support constants in the analysis, the subset of supported MIR instructions needs to be extended.
The sixth reason is that the static analysis does not check for overflows of numeric values, which causes the integer overflow to be undetected and the corresponding tests to fail.
The seventh reason is that dynamic calls are not handled by the analysis, and therefore language features that depend on them are not supported. Trait methods allow dynamic dispatch in Rust and use dynamic calls. These dynamic calls are assumed to refer to the default implementation of the Trait if it exists. Test cases that define traits without a default implementation show that no function is called. Instead of this behavior, the analysis should correctly identify that the method is called on types that implement the trait based on a type hierarchy or callgraph.

6.3 Evaluation of Support for Vulnerabilities with Affected Parameters

The support for vulnerabilities with affected parameters is evaluated using a second microbenchmark[4] developed in the context of this thesis. This microbenchmark uses a synthetic RustSec Database that includes the definition of 10 vulnerabilities, each testing individual aspects of a specification of affected parameters. The test code in the microbenchmark is defined in a library crate, and all test functions are defined as public functions. The tool analyzes all public functions and creates vulnerability reports for each of them. The microbenchmark defines one positive and one negative test for each feature tested. Table 6-2 shows the test code and the definition of a vulnerability for testing the

```rust
fn less_than_one(value: i32) { }

pub fn test_positive_less_than() {
    less_than_one(0);
}

pub fn test_negative_less_than() {
    less_than_one(2);
}
```

(a) Test Code: Positive And Negative Test

```
[affected.parameters]
"test::less_than_one" = [
    {
        parameter = 1,
        operator = "<",
        rust_type = "i32",
        rust_value = "1"
    }
]
```

(b) Vulnerability Definition

Table 6-2: Testing Operator Less Than

[4]Folder `masterthesis-implementation/extension-benchmark` in submitted zip file

less than operator. The results of all tests are shown in Table 6-3. The operators are

Tested Feature	True Positives	True Negatives	Passed all tests
Operator `<`	1/1	1/1	✓
Operator `<=`	1/1	1/1	✓
Operator `>`	1/1	1/1	✓
Operator `>=`	1/1	1/1	✓
Operator `==`	1/1	1/1	✓
Operator `!=`	1/1	1/1	✓
Option::None	1/1	1/1	✓
Empty String	1/1	1/1	✓
Boolean `true`	1/1	1/1	✓
Boolean `false`	1/1	1/1	✓

Table 6-3: Supported Specifications of Affected Parameters

tested by passing an integer `i32` to the vulnerable function as the first parameter. All test cases for the microbenchmark pass.

6.4 Macrobenchmark

The objective of this part of the evaluation is to run the tool on real-world applications to measure the runtime performance of the data-flow analysis and to obtain first insights into the quality of the findings.

The evaluation is performed by running the developed tool on a set of crates that are publicly available in the Rust Package Registry. The data set consists of 430 crates[5] selected at random but containing at least one vulnerable dependency in their `Cargo.lock` file and are compilable on the evaluation system. The tool is then executed iteratively on each crate, and the timings and findings are recorded and persisted. If the total execution time of the tool exceeds 60 minutes, the analysis will be stopped, and the corresponding crate will be marked to exceed the time limit during the analysis.

The findings that are reported by the tool during the evaluation are recorded and analyzed afterward to get insights into the quality of the reported findings. Additionally, the *total analysis time* is measured, which is the duration in which the tool analyzes the complete dependency tree. The individual analysis duration for each crate in the dependency tree is measured, and then the sum is calculated afterwards to acquire the total analysis time.

[5]File `masterthesis-implementation/macrobenchmark/compilable.txt` in submitted zip file

6.4.1 Results of Macrobenchmark

During the evaluation of 430 crates, 150 crates exceeded the maximum execution time of 60 minutes and were removed from the evaluation in retrospect. Figure 6-2 shows the distribution of the analysis time. The figure only includes data for the 280 crates[6] for which the total execution time of the tool was below 3600 seconds and does not show the 150 cases that exceeded this limit. For these remaining 280 crates, the data[7] shows that in 270 cases (96.4%), the analysis finishes in less than 600 seconds (10 minutes). These 270 cases correspond to 62.7% of the total 430 crates. 135[8] of the 150 cases (90%) exceeding the execution time limit take a long time analyzing the library `unicode-normalization`, which they use as an external dependency. This library contains a function consisting of a match statement with more than 700 comparisons between a character and a range. This check causes the analysis to run into a complexity problem. Section 6.4.2 analyses this problem in detail and identifies possible mitigations for future work. This problem was not identified in the evaluation of the microbenchmark because the problem relies on repetitive code, but the microbenchmark is designed to test only one language feature at a time. In summary, it has been identified that the analysis finishes in under 10 minutes in 62.7% of all cases and finishes in less than one hour in 65.1% of all cases. In 34.9% of the cases, the total execution of the tool exceeds the limit of one hour, and for 90% of those cases exceeding the limit, the reason is known and a possible mitigation was found

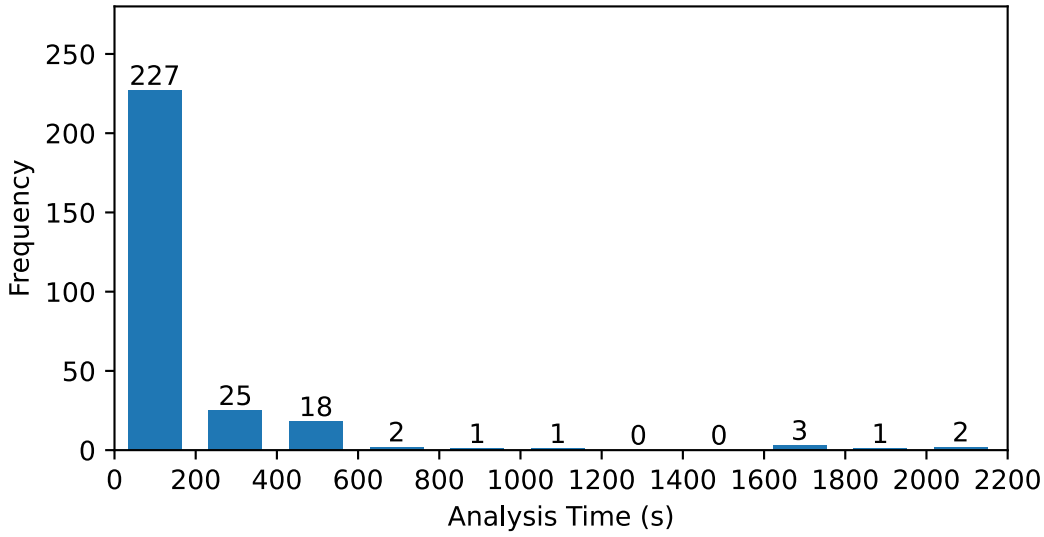


Figure 6-2: Distribution of Analysis Time

[6]File `masterthesis-implementation/macrobenchmark/finished-below-one-hour.txt` in submitted zip file
[7]File `masterthesis-implementation/macrobenchmark/analysis-times.txt` in submitted zip file
[8]File `masterthesis-implementation/macrobenchmark/dependents-of-unicode-normalization.txt` in submitted zip file

that could also positively influence the analysis time of the other cases.
The findings for the crates that cargo-flowcheck finished analyzing within the time limitation (286 crates), are discussed next. There are 138 crates that use an external dependency with a potential vulnerability that specifies the affected functions in the vulnerability definition. The tool can only check the data-flow for these 138 cases, where the affected functions are known. The tool found two vulnerable crates, `rzip-0.9.6` and `weblab-0.2.27`. In both cases the vulnerability `RUSTSEC-2020-0071`[9] *(Potential segfault in the time crate)*, was found, which was already discussed in Section 3.3.1.
Therefore, there are two positive cases where crates are identified as vulnerable and 136 negative cases where crates are identified as not vulnerable. To verify these results, the source code of the analyzed crates is inspected next. In one positive case, the tool found the vulnerability in the function `main` of the application program `rzip-0.9.6`[10]. This is a simple command-line interface (CLI) tool for managing compressed files. This finding is a true positive, since the application invokes the vulnerable function `time::now()`.
The function calls that are involved in the vulnerability are shown below:

1. `rzip::main()`[11] (entry point)
2. `rzip::Zipper::archive()`[12]
3. `rzip::Zipper::append_entry()`[13]
4. `zip::write::FileOptions::default()`[14]
5. `time::now()` (vulnerable)

The other positive case is found in the crate `weblab-0.2.27`, which contains two vulnerable functions: `main` and `error_main`. This crate is a software library that can be used to generate assignments to practice single concepts of Rust. Both functions `main` and `error_main` are public and, therefore, exported by the library. The tool correctly handles both of these functions as entry points and scans for possible vulnerabilities in each. Both of these functions are actually vulnerable, and therefore the crate is correctly identified as vulnerable.

[9] https://rustsec.org/advisories/RUSTSEC-2020-0071
[10] https://crates.io/crates/rzip/0.9.6
[11] https://github.com/mass10/rzip/blob/b8ea0caf79e7/src/main.rs#L59
[12] https://github.com/mass10/rzip/blob/b8ea0caf79e7/src/application/core.rs#L90
[13] https://github.com/mass10/rzip/blob/b8ea0caf79e7/src/application/core.rs#L78
[14] https://github.com/zip-rs/zip/blob/7edf2489d5cf/src/write.rs#L101

The function calls that are involved in the vulnerability are shown below:

1. `weblab::main()`[15] (entry point)
2. `weblab::error_main()`
3. `weblab::generate_zip()`
4. `zip::write::FileOptions::default()`
5. `time::now()` (vulnerable)

In one negative case, the tool found no vulnerability in the code of the `rustcat-1.1.0` application program, which is a network application that enables listening to network ports. The dependency analysis identified the potential vulnerability `RUSTSEC-2021-0119`[16] *(Out-of-bounds write in nix::unistd::getgrouplist)* in the dependency `nix-0.20.0` with the affected function `nix::unistd::getgrouplist`. The dependency tree visualized by the tool `cargo tree`[17] showed that this dependency is only used by the intermediate dependency `rustyline-8.2.0`[18]. This dependency was further analyzed and it was identified that it does not invoke the affected function `nix::unistd::getgrouplist`. Therefore, this negative finding is a true negative and shows that the precision of the results increases when the analyzed program does not call any affected function of a vulnerable dependency.
In another negative case, the tool found no vulnerability in the code of the `heatmap-0.6.4` software library that generates heatmaps and histograms. The dependency analysis identified the potential vulnerability `RUSTSEC-2020-0071`[19] *(Potential segfault in the time crate)* in the dependency `time-0.1.45`, which was discussed in Section 3.3.1. The analyzed library does not call any affected functions of the crate `time`. Therefore, this negative finding is a true negative and shows that the precision of the results increases when the analyzed program does not call any affected function of a vulnerable dependency.
The total number of analyzed crates including all dependencies is 11.195 (40 dependencies on average per top-level crate), underlining the relevance of an inter-crate approach.
212 (74%) crates have a nonempty re-export map. This means that they use the keyword `pub use` to re-export functions into different modules, and this re-export is visible to the dependents of the crate, underlining the relevance of this feature.

[15] https://gitlab.ewi.tudelft.nl/cese/software-fundamentals/weblab-rs/-/blob/b2eddaa4913f/weblab/src/cli.rs#L738
[16] https://rustsec.org/advisories/RUSTSEC-2021-0119
[17] https://doc.rust-lang.org/cargo/commands/cargo-tree.html
[18] https://crates.io/crates/rustyline/8.2.0
[19] https://rustsec.org/advisories/RUSTSEC-2020-0071

6.4.2 Complexity Problem Analysis

During analysis of 135 crates (including the crate `taos-optin-0.7.0`[20]), the external dependency `unicode-normalization`[21] was analyzed. This library provides functionality for unicode character composition and decomposition. During analysis, the tool spends most of the time analyzing the function `is_public_assigned`[22] and does not finish after a time span of 60 minutes, hinting at a combinatorial explosion in the algorithm `AnalyzeBasicBlock`.

The function `is_public_assigned` consists of 710 lines of Rust code, and an excerpt is shown in Figure 6-3a. The function checks consecutively if a character is within a specific range using Pattern Matching. To test if a character is within a range, both the lower and upper bounds are tested, each with one comparison. Depending on the outcome of the comparison, different basic blocks are branched to.

This branch in the control flow graph causes the algorithm to add two successor basic blocks to the worklist. While two worklist items are added, the algorithm pops one element at a time from the worklist causing the worklist to grow. The basic blocks that remain on the worklist are not processed until the analysis reaches the bottom of the function for the first time. At this point, the remaining basic blocks that are still on the worklist start to be visited for the first time. Pairs of consecutive basic blocks have a common successor basic block (in Figure 6-3b both basic blocks `bb0` and `bb1` have a common successor `bb2`, same holds for `bb2`, `bb3` and `bb4` and so on). This leads to the scenario that the remaining basic blocks on the worklist are predecessors to already visited basic blocks, which are then visited a second time. Each basic block performs an assignment to a local variable, which is used by the same basic block for the jump condition of the switch statement. Because of this assignment, the state of the lattice changes. This happens for each basic block.

There are an exponential number of combinations of local variables that are either initialized or not yet initialized, and therefore the lattice state differs at the beginning of each basic block for an exponential number of cases. This is an exponential explosion in the complexity for the analysis depending on the number of variables. However, the algorithm is bound to terminate eventually. Although the variables holding the jump conditions are only used locally in one basic block by the corresponding switch statement, they prevent reaching a fixed point on the function level. As a solution, a live-variable analysis could be integrated into the approach to detect variables that are only used locally within one basic block and keeps these values out of the basic block summary. It has been identified

[20] https://crates.io/crates/taos-optin/0.7.0
[21] https://crates.io/crates/unicode-normalization/0.1.22
[22] https://github.com/unicode-rs/unicode-normalization/blob/master/src/tables.rs#L33573

```rust
#[inline]
fn is_public_assigned(c: char) -> bool {
  match c {
      '\u{0000}'..='\u{0377}'
    | '\u{037A}'..='\u{037F}'
    | '\u{0384}'..='\u{038A}'
    | '\u{038C}'
    | '\u{038E}'..='\u{03A1}'
    | '\u{03A3}'..='\u{052F}'
    | '\u{0531}'..='\u{0556}'
    | '\u{0559}'..='\u{058A}'
    | '\u{058D}'..='\u{058F}'
    | '\u{0591}'..='\u{05C7}'
    | '\u{05D0}'..='\u{05EA}'
    | '\u{05EF}'..='\u{05F4}'
    | '\u{0600}'..='\u{070D}'
    | '\u{070F}'..='\u{074A}'
    | '\u{074D}'..='\u{07B1}'
    | '\u{07C0}'..='\u{07FA}'
    | '\u{07FD}'..='\u{082D}'
    | '\u{0830}'..='\u{083E}'
    | '\u{0840}'..='\u{085B}'
    // (690 cases omitted)
    => true,
  _ => false,
}
```

(a) Excerpt of Rust Code

```
bb0 {
  $1274 = unsupported <= $1;
  $1276 = 0 == $1274;
  $1277 = 0 != $1274;
  switch bb2 if $1276, bb1 if $1277;
}
bb1 {
  $1275 = $1 <= unsupported;
  $1278 = 0 == $1275;
  $1279 = 0 != $1275;
  switch bb2 if $1278, bb1276 if $1279;
}
bb2 {
  $1272 = unsupported <= $1;
  $1280 = 0 == $1272;
  $1281 = 0 != $1272;
  switch bb4 if $1280, bb3 if $1281;
}
bb3 {
  $1273 = $1 <= unsupported;
  $1282 = 0 == $1273;
  $1283 = 0 != $1273;
  switch bb4 if $1282, bb1276 if $1283;
}

// (1274 basic blocks omitted)
```

(b) Excerpt of Analysis IR

Figure 6-3: Function unicode_normalization::tables::is_public_assigned

that the Rust compiler already performs this live-variable analysis, and the MIR includes special instructions `StorageLive` and `StorageDead` marking the lifetimes of individual local variables. These instructions are currently not part of the supported subset of MIR. Modifying the original source code to include a specific amount of range checks resulted in the execution times shown in Table 6-4. The *factor* represents for each row the ratio

#range checks	*execution time*	*factor*
10	89 ms	—
20	152 ms	×2.0
30	355 ms	×2.4
40	757 ms	×2.3
50	1538 ms	×2.0
60	2779 ms	×1.8
50	4768 ms	×1.7
60	7533 ms	×1.6
70	11483 ms	×1.5
80	16898 ms	×1.5
90	23810 ms	×1.4
100	32732 ms	×1.3

Table 6-4: Number of Range Checks and Analysis Time

between the execution time and the previous execution time rounded to one decimal place. The data suggests an exponential relationship between the number of range checks and the resulting execution time.
Replacing all range checks in the original code by matches against a single value reduces the analysis time of the function to 83ms, which is in an expected range.

6.5 Threats to Validity

Wohlin et al. (2012) provide a checklist to identify threats to the validity of experiments in software engineering. This checklist incorporates the conclusion validity, internal validity, construct validity, and external validity.
According to Wohlin, the *conclusion validity* is threatened if there are "issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment", including issues of low statistical power, violation of assumptions of statistical tests, and threats of fishing and incorrect significance levels in statistical tests.
The *conclusion validity* of the evaluation performed in this thesis is threatened because it verifies the correctness of the results only by checking a small sample size. No false positive cases were detected; still a larger sample size could have revealed false positive cases. Two true positives were detected, showing that the approach can detect vulnerabilities, but it cannot be concluded how reliably these vulnerabilities are found on the basis of the small sample size. Two negative cases were investigated and were both found to be true negatives. Still, this does not show the absence of false negative cases, and investigating more negative cases could have revealed the existence of false negative results.
According to Wohlin, the *internal validity* is threatened if there are "influences that can affect the independent variable with respect to causality, without the researcher's knowledge", including the influence of selecting individual cases from a larger group of cases and the influence of testing only a single time.
The *internal validity* of the evaluation performed in this thesis is threatened, because the microbenchmark is designed to test a limited set of language features that are mentioned in the Rust Book. Still, it is expected that this source contains the most relevant language features of the Rust language. Moreover, the measured run-times are only tested once and could therefore be dependent on external factors on the evaluation machine. Still, by using a larger dataset of 430 crates, the effect of an individual measurement on the whole result is expected to be reduced.
According to Wohlin, the *construct validity* is threatened if there are "concerns generalizing the result of the experiment to the concept or theory behind the experiment", including design threats related to levels of constructs. Here, the level of support for a

construct has an influence on the result of the evaluation.
The *construct validity* of the evaluation performed in this thesis is threatened because of the architectural decision not to report the data-flow that leads to the vulnerability to the user. This is a limitation of the design reducing the possibility to verify individual data-flows in the evaluation. If a crate is vulnerable in the code, but the tool found a false data-flow as evidence for the correct vulnerability, then the evaluation might incorrectly treat this as a true positive, while in reality, there are both a false positive (for the invalidly wrong data-flow) and a false negative (for this missing actual data-flow). To mitigate this problem in the context of this evaluation, debugging code was temporarily added to the tool allowing one to get insights into the nature of the identified data-flows and confirming their correctness.
According to Wohlin, the *external validity* is threatened if there are "conditions limiting the ability to generalize the results of the experiment to industrial practice" including the interaction of selection and treatment, the interaction of setting and treatment, and the interaction of history and treatment.
The *external validity* of the evaluation performed in this thesis is threatened, because the macrobenchmark only analyzes a selected number of crates and rejects crates that do not compile in the evaluation system or do not contain a vulnerable dependency. To mitigate this threat, the crates were randomly selected, and it was tested for a small number of cases that a crate with no vulnerable dependencies does not report any vulnerability as required by the High-Level Architecture.
Another threat to the *external validity* is that crates were discarded in the macrobenchmark whenever the total execution time of the tool exceeded one hour, which limits the ability to generalize the measured run-times. To mitigate the threat, a qualitative analysis was performed in Section 6.4.2 that affects 90% of all cases exceeding the time limit.
The hardware configuration can influence the results when platform-specific vulnerabilities are analyzed and can also influence performance measurements. To mitigate this threat, the hardware configuration used in the evaluation is shown in Section 6.1.
The complexity limitation parameter K limits the ability to generalize the results for cases with a different value for K. However, stating the actual value of $K = 30$ as used in the evaluation increases the reproducibility.
The evaluation does not compare the tool to other tools. This could reduce the ability to generalize the results and, therefore, threaten the external validity. Still, the test cases of the microbenchmark are provided to increase reproducibility.

6.6 Reproducibility

To increase the reproducibility of the evaluation, the following steps have been taken. The evaluated tool "cargo-flowcheck"[23] is provided in a zip file and the installation process is described in Appendix A1. The experimental setup of the evaluation is described in Section 6.1, including information about the system configuration on which the evaluation was performed.
The microbenchmarks can be executed in the directory `microbenchmark` and `evaluation-benchmark` using the command `cargo test` or running the tests in an Integrated Development Environment (IDE). The test cases are named with a naming convention so that filtering only the positive or negative tests is possible. The command `cargo test positive` shows the results of positive tests and `cargo test negative` shows the results of negative tests. The macrobenchmark results are included in the directory `macrobenchmark`. The tool `cargo-flowcheck` was invoked on each compilable crate using the following command, which generates json output including the results of the analysis and the analysis times:

```
cargo flowcheck --db masterthesis-advisory-db-fork --no-fetch --json
```

6.7 Conclusion

The tool supports 16 of 44 tested features in the Rust language completely (36%), and the reasons for not supporting the remaining 28 features were discussed in detail. The main reason is the focus on a subset of the Mid-Level Representation (MIR) as well as the missing support for array sensitivity, field sensitivity, alias sensitivity and dynamic calls. The tool supports the extension designed for the RustSec Database to specify the values of the affected parameters.
The macrobenchmark shows that the developed tool works as designed and can be used to find real vulnerabilities in real-world applications in a reasonable time frame (62.7% finish in less than 10 minutes). The tool does not finish analyzing in one hour for 150 crates and in 135 of these cases, the reason is a dependency on the library `unicode-normalization`, which uses a specific programming pattern that causes the analysis to waste a lot of time. Possible mitigations are identified for future work in Section 6.4.2. All two positive findings were manually verified as true positives, and two negative findings were manually verified to be true negatives. It is still possible that other negative cases are false negatives because of an unsupported language feature that is involved in the data-flow that prevents the analysis from tracing the data-flow completely.

[23] Folder `masterthesis-implementation` in submitted zip file

7 Related Work

This chapter examines various related approaches to analyzing Rust code for bug detection. In Section 7.1, the MirChecker approach using Abstract Interpretation is presented. In Section 7.2, approaches involving taint analysis are presented. In Section 7.3 approaches involving external verifiers and model checkers are shown. In Section 7.4 these approaches are compared to the approach proposed in this thesis.

7.1 Approach Involving Abstract Interpretation

Li et al. (2021) introduces MirChecker, an automated bug detection framework for Rust programs that performs Abstract Interpretation on Rust's Mid-level Intermediate Representation (MIR), tracking both numerical and symbolic information. MirChecker utilizes constraint solving techniques to identify potential runtime crashes and memory-safety errors and provides informative diagnostics to users. It is integrated as an additional analysis step in the official Rust compiler and is seamlessly integrated with the official Package Manager Cargo. The fundamental design of MirChecker adheres to the Monotone Framework (Kam & Ullman, 1977). For every statement, transfer functions are defined to describe how these properties are manipulated and passed on. It utilizes two abstract domains, one for numerical values (used for the bound analysis) and one for symbolic values (used as a memory model). A fixed-point algorithm is executed to propagate properties through the control flow graph. MirChecker supports interprocedural analysis but skips recursive functions and only supports function calls for which the call target can be statically determined and is defined in the same crate. This makes MirChecker highly related. Still, MirChecker does not utilize the RustSec Database, does not scan deeply into external dependencies, and unlike the implementation developed in this thesis, it does not support recursion. On the other hand, MirChecker supports alias sensitivity for local variables, which is a feature that this thesis does not support.

7.2 Approaches Involving Taint Analysis

Bae et al. (2021) introduce Rudra, an approach to detect bugs in unsafe Rust code using taint analysis. The Rudra approach focuses on three safety guarantees that are enforced by the Rust compiler for safe code, but need to be implemented manually, when writing unsafe Rust code. Firstly, it addresses "Panic Safety" by ensuring that the program aborts in a controlled way in the presence of panics within unsafe code to prevent memory safety bugs. Second, it enforces the "Higher Order Invariant" ensuring safe functions receive safe inputs without errors. Lastly, Rudra addresses the "Propagation of Send/Sync in Generic Types," tackling challenges in manually implementing thread safety traits for

generic types, such as raw pointers. Rudra employs a hybrid analysis approach operating on the High-Level Representation (HIR) to collect function declarations and trait implementations along with their declared safety and on the Mid-Level Representation (MIR) to analyze the code semantics through a data-flow analysis on the control-flow graph.
Cui, Chen, Xu, and Zhou (2023) introduce SafeDrop, an approach that identifies memory corruption bugs originating from heap memory deallocation in combination with unsafe code. The approach focuses on issues with automatic deallocation associated with the Rust ownership model and the resource acquisition is initialization (RAII) pattern. The Rust ownership model aims to prevent dangling pointers and memory leaks but is observed to cause critical bugs when combined with manually written unsafe code, where it can lead to dropping buffers still in use, leading to use-after-free bugs. The approach uses taint analysis and alias analysis on the Rust Mid-Level Representation (MIR), where it traverses the control-flow graph of a function and identifies paths to check, and then performs the analysis for invalid drops on these paths. Compared to the approach described in this thesis, SafeDrop does not utilize the RustSec Database and does not scan deeply into external dependencies, in the way that the inter-crate analysis described in this thesis does.

7.3 Approaches Involving External Verifiers

Lindner, Aparicius, and Lindgren (2018) propose a contract-based verification process to statically ensure memory safety and panic-free execution of Rust code transformed into LLVM bitcode using the symbolic execution engine KLEE (Cadar, Dunbar, & Engler, 2008). The proposed solution addresses challenges with unchecked raw array indexing and panic handling for safety-critical applications. The Rust compiler includes runtime checks that lead to runtime panics on violation and downgraded performance of the program as the checks. The approach detects panics at compile time, improving both correctness and performance by eliminating the need for run-time verification code. The approach uses contracts as formal specifications that define the expected behavior of a program or its components. After successful verification of these contracts, the program is guaranteed to be safe and free of run-time panics even without run-time checks, leading to improved correctness, safety, and performance.
(Baranowski, He, & Rakamarić, 2018) propose a verification approach for Rust programs using the SMACK verifier (Rakamarić & Emmi, 2014). The Rust Compiler is used to transform the Rust program into LLVM IR, which is supported as input for the SMACK verifier. Some Rust-specific functionality was added to support the specific patterns that emerge in the LLVM IR generated for Rust programs Additionally models of Rust libraries, particularly the Rust datastructure, `Vec` (a dynamically sized array) were added using

SMACK's existing modeling capabilities. SMACK then translates the LLVM IR code into the Boogie Language (Leino, 2008), which is then verified using back-end Boogie verifiers like Corral (Lal, Qadeer, & Lahiri, 2012).
CRUST (Toman et al., 2015) addresses memory safety concerns associated with unsafe library code in Rust by idenfying essential functions containing unsafe code (so-called *drivers*) and generating test cases in C language for them by automatically adding memory safety assertions and translating them into equivalent C code. This C code is then subsequently verified using the CBMC (C Bounded Model Checker), a widely used model checker introduced by Kroening and Tautschnig (2014) that further transforms the problem into SMT (Satisfiable Modulo Theories).
Prusti (Astrauskas et al., 2022) is a formal verification approach to checking invariants in the Rust program code, which are derived from the program's semantics (loop bounds) and can also be annotated by the user. Prusti is based on the published master thesis Rust2Viper (Hahn, 2016) and interfaces with the Rust compiler to obtain the High-Level Intermediate Representation (HIR) and Mid-Level Representation (MIR) of the Rust code to derive invariants for the program. Prusti also allows the user to define additional invariants in the Rust source code using a set of Rust macros that are provided by Prusti and allow one to describe conditions that must hold at a specific point in the program and contribute to the specification for the verifier. The complete specification is then converted to Viper Language (Müller, Schwerhoff, & Summers, 2016), a verification language in which verification is performed. Prusti supports external crates, in general, but invariants specified in external crates are not seen.

7.4 Conclusion

This chapter outlined related approaches for analyzing Rust code to identify bugs. The commonality between all these approaches is that they utilize the Rust compiler to provide preprocessed representations of the Rust program, before performing any analysis on it. While MirChecker, Rudra and SafeDrop operate on the Rust IRs directly, other approaches like No Panic, SMACK and Prusti transform the code further to external verification languages (KLEE, Boogie, Viper) in combination with off-the-shelf verifiers as back-ends mapping the generated errors back to the Rust source code.
All of the identified approaches operate on one single crate in isolation, unlike the inter-crate data-flow analysis implemented in this thesis, and according to the research performed in the context of this thesis, there are no other hybrid approaches that combine the RustSec Database with a data-flow analysis, as has been done in this thesis.

8 Conclusion and Future Work

This chapter concludes the thesis and summarizes its results. Section 8.1 summarizes the thesis and answers the three research questions, and Section 8.2 describes possible future work.

8.1 Conclusion

This thesis shows the feasibility of a hybrid approach to identify security vulnerabilities in Rust code. This hybrid approach uses the RustSec Database to identify vulnerabilities in external dependencies on the project level and then performs a subsequent data-flow analysis to confirm in the code if vulnerable libraries are used in vulnerable ways, checking if vulnerable functions are actually called and, where applicable, if parameters are actually in the vulnerable range of values. The thesis also contributes the developed tool "cargo-flowcheck"[1] and a microbenchmark[2] for testing the support for language features of Rust code in a static code analysis.

In the context of Research Question *RQ1 (How can the RustSec Database be utilized to support data-flow analysis)*, this thesis identified that extending the RustSec Database to include vulnerable parameter values allows to increase the precision of detecting vulnerabilities that only trigger for specific parameter values.

Only 101 of all 429 code-related database entries (23.5%) in the RustSec Database define a list of affected functions. These lists were analyzed revealing instances where they were incomplete. Incomplete vulnerability entries in the database are a threat to the approach described in this thesis, as the approach relies on these definitions.

To deal with an incomplete list of affected functions, the data-flow analysis scans into the vulnerable crate itself to identify transitively affected functions. If the affected functions are not specified at all, no subsequent data-flow analysis can be performed, and the tool operates only on the project level, similar to the related tool cargo-audit. Furthermore, the thesis identified two affected functions, which are not part of the RustSec Database and were added to a forked version of the database.

Some vulnerabilities were identified that are only triggered by specific parameter values. This information is currently not specified for the vulnerabilities in the RustSec database. The database schema was extended to enable specifying these parameter values via a comparison with a literal value. An inequality is helpful to represent ranges of values: For vulnerability `RUSTSEC-2022-0051`[3], all negative values triggered the vulnerability if passed as argument for the parameter `size`, which expects only positive values. The RustSec

[1] Folder `masterthesis-implementation` in submitted zip file
[2] Folder `masterthesis-implementation/microbenchmark` in submitted zip file
[3] https://rustsec.org/advisories/RUSTSEC-2023-0051

database was forked[4] and vulnerability entries were modified to include value ranges for affected parameters and to add additional affected functions that were identified.
In the context of Research Question *RQ2 (How can data-flows across external dependencies be analyzed)*, this thesis identified that it is possible perform a data-flow analysis by summarizing the data-flow behavior of all external dependencies individually in the topological order of their inter-dependencies. This ensures that the analysis results of dependencies are available before the analysis of dependents needs them. Performing a data-flow analysis across the boundaries of individual software projects then allows to identify vulnerabilities involving multiple software dependencies.
In the context of Research Question *RQ3 (What program representations of Rust code are suitable for data-flow analysis)*, the representations Abstract Syntax Tree (AST), High-Level IR (HIR), Typed High-Level IR (HIR), Mid-Level IR (MIR) and LLVM IR were compared. It was identified that, unlike the AST, HIR, and THIR, both the MIR and LLVM IR model the control flow and can therefore be used for a data-flow analysis with the LLVM IR standing out for its language-agnostic nature, supporting various source code languages. Still, for the database-based approach, the MIR was preferred because it provides the most Rust-related information, which is used to query the RustSec Database. Furthermore, the HIR was used to identify re-exported functions in libraries, which was shown to be relevant in 74% of programs that were analyzed in the evaluation.
For evaluation, a microbenchmark was created to assess support for Rust language features. The microbenchmark revealed that 28 of 44 language features are not fully supported. The main reason for this is that only a well-defined subset of MIR instructions is supported, and the analysis has no field sensitivity, array sensitivity, or alias sensitivity capabilities. The tool fully supports the designed extension to the RustSec Database to specify the values of the affected parameters.
To evaluate the findings in real-world programs, the tool was executed on randomly selected crates from the Rust Package Registry that use at least one vulnerable dependency and are compilable on the evaluation system. For the 138 crates in the evaluation, the affected functions were available in the database, which is a prerequisite to performing the data-flow analysis. The tool confirmed the vulnerabilities of two crates and both were identified as true positives by manual inspection. For the other 136 crates, the tool did not confirm the vulnerability in the code. Two negative cases were manually inspected and identified to be true negatives.
The run-time performance of all analyzed crates was evaluated. 270 crates were completed in less than 10 minutes (62.7%) with an average dependency tree size of 40, highlighting the importance of including external dependencies in the analysis. For 90% of the crates

[4]Folder `masterthesis-advisory-db-fork` in submitted zip file

that took more than one hour, a specific programming pattern was identified that triggers the algorithm's worst-case performance and a possible extension to the approach was identified that could reduce this effect.

8.2 Future Work

The scope of this thesis was chosen to be inter-crate analysis, and some functionality was not considered in the design, like alias sensitivity, field sensitivity, and array sensitivity. In addition, as input for the analysis, only a subset of the MIR instructions is considered. The evaluation showed that, because of this, not all language features of Rust are supported by the approach. Therefore, one logical future work is to remove these limitations, adding the sensitivities, and supporting a larger subset of MIR or the complete MIR to increase the support for Rust language features. Furthermore, the influence of the parameter K (complexity limitation) on the results can be investigated.
One limitation of the designed tool is that it does not report the data-flow that was identified. Instead, only the function that serves as an entry point into the code is reported (e.g. the main function). This makes it difficult to comprehend the reported findings. Future work could address this problem and report the data-flow for each finding.
In the evaluation, a real-world pattern was determined that triggers the algorithm's worst-case performance. Detecting the scope of variables using a live-variable analysis or by utilizing the lifetime data present in the MIR can reduce the impact of locally used variables and can mitigate such code from causing exponential analysis complexity. The RustSec Database could be extended further to support Boolean formulas that describe more complex conditions with respect to the parameter values.
Furthermore, Cargo's compilation cache can be extended to include the function summaries produced by the analysis (the fingerprinting system might need adaptation). This can lead to performance gains when analyzing the same project multiple times and could make it feasible to integrate the tool into an Integrated Development Environment (IDE), as the function summary approach enables one to invalidate and re-analyze only (transitively) changed parts of the code. In general, more real-time feedback could be added to the tool, including a live output of vulnerabilities found.
To further reduce analysis times, a public summary registry could be created that contains analysis results for all published crates in the Rust Package Registry[5]. The tool could then download precomputed summaries for external libraries. An interesting question is how generic functions can be summarized, since MIR creates a copy for every combination of generic arguments used (monomorphization), but when analyzing an external library, it is not known which generic types will be used by clients of the library.

[5] https://crates.io

Bibliography

Allen, F. E. (1970, July). Control flow analysis. *ACM SIGPLAN Notices*, *5*(7), 1–19. doi: 10.1145/390013.808479

Astrauskas, V., Bílỳ, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., ... Summers, A. J. (2022). The prusti project: Formal verification for rust. In *Lecture notes in computer science* (p. 88–108). Springer International Publishing. doi: 10.1007/978-3-031-06773-0_5

Bae, Y., Kim, Y., Askar, A., Lim, J., & Kim, T. (2021, October). Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the 28th symposium on operating systems principles* (p. 84–99). New York, NY, USA: ACM. doi: 10.1145/3477132.3483570

Baranowski, M., He, S., & Rakamarić, Z. (2018). Verifying rust programs with smack. In *Automated technology for verification and analysis* (p. 528–535). Springer International Publishing. doi: 10.1007/978-3-030-01090-4_32

Bodden, E. (2018a, July). The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Companion proceedings for the issta/ecoop 2018 workshops* (p. 85–93). Amsterdam Netherlands: ACM. doi: 10.1145/3236454.3236500

Bodden, E. (2018b, May). State of the systems security. In *Proceedings of the 40th international conference on software engineering: Companion proceeedings* (p. 550–551). ACM. doi: 10.1145/3183440.3183462

Cadar, C., Dunbar, D., & Engler, D. (2008, December). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th usenix symposium on operating systems design and implementation.* San Diego, CA: USENIX Association.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th acm sigact-sigplan symposium on principles of programming languages* (p. 238–252). Los Angeles, California: ACM Press. doi: 10.1145/512950.512973

Cui, M., Chen, C., Xu, H., & Zhou, Y. (2023, May). Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *Transactions on Software Engineering and Methodology*, *32*(4), 1–21. doi: 10.1145/3542948

Decan, A., Mens, T., & Grosjean, P. (2018, February). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, *24*(1), 381–416. doi: 10.1007/s10664-017-9589-y

Hahn, F. (2016). *Rust2viper: Building a static verifier for rust* (Master's thesis, ETH Zürich). doi: 10.3929/ETHZ-A-010669150

Kam, J. B., & Ullman, J. D. (1977, September). Monotone data flow analysis frameworks. *Acta Informatica*, *7*(3), 305–317. doi: 10.1007/bf00290339

Kawachiya, K., Ogata, K., & Onodera, T. (2008, October). Analysis and reduction of memory inefficiencies in java strings. *ACM SIGPLAN Notices*, *43*(10), 385–402. doi: 10.1145/1449955.1449795

Kikas, R., Gousios, G., Dumas, M., & Pfahl, D. (2017, May). Structure and evolution of package dependency networks. In *Ieee/acm 14th international conference on mining software repositories* (p. 102–112). IEEE. doi: 10.1109/MSR.2017.55

Kroening, D., & Tautschnig, M. (2014). Cbmc – c bounded model checker. In *Tools and algorithms for the construction and analysis of systems* (p. 389–391). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-54862-8_26

Lal, A., Qadeer, S., & Lahiri, S. K. (2012). A solver for reachability modulo theories. In *Lecture notes in computer science* (p. 427–443). Springer Berlin Heidelberg. doi: 10.1007/978-3-642-31424-7_32

Lattner, C., & Adve, V. (2004, March). Llvm: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on code generation and optimization* (p. 75–86). IEEE. doi: 10.1109/cgo.2004.1281665

Leino, K. R. M. (2008, June). *This is boogie 2* (techreport). Redmond, WA, USA. (Issue: KRML 178)

Li, Z., Wang, J., Sun, M., & Lui, J. C. (2021, November). Mirchecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the conference on computer and communications security* (p. 2183–2196). New York, NY, USA: ACM. doi: 10.1145/3460120.3484541

Lindner, M., Aparicius, J., & Lindgren, P. (2018, July). No panic! verification of rust programs by symbolic execution. In *Proceedings of the 16th international conference on industrial informatics* (p. 108–114). IEEE. doi: 10.1109/indin.2018.8471992

Matsakis, N. (2015, July). *Rust rfc 1211.*

McAfee, & CSIS. (2014, June). *Net losses: estimating the global cost of cybercrime.* Retrieved 2023-11-02, from https://www.csis.org/analysis/net-losses-estimating-global-cost-cybercrime

Mell, P., Scarfone, K., & Romanosky, S. (2006). Common vulnerability scoring system. *IEEE Security & Privacy*, *4*(6), 85-89. doi: 10.1109/MSP.2006.145

Møller, A., & Schwartzbach, M. I. (2018, October). *Static program analysis.*

Müller, P., Schwerhoff, M., & Summers, A. J. (2016, December). Viper: A verification infrastructure for permission-based reasoning. In *Verification, model checking, and abstract interpretation* (p. 41–62). Springer Berlin Heidelberg. doi: 10.1007/978-3

-662-49122-5_2

Preston-Werner, T. (2013, June). *Semantic versioning 2.0.0.* Retrieved 2023-11-01, from https://semver.org/spec/v2.0.0.html

Preston-Werner, T., Gedam, P., et al. (2019). *Toml: Tom's obvious minimal language.* Retrieved 2023-11-01, from https://github.com/toml-lang/toml

Rakamarić, Z., & Emmi, M. (2014). Smack: Decoupling source language details from verifier implementations. In *Lecture notes in computer science* (p. 106–113). Springer International Publishing. doi: 10.1007/978-3-319-08867-9_7

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, *74*(2), 358–366. doi: 10.1090/s0002-9947-1953-0053041-6

Ryder, B. (1979, May). Constructing the call graph of a program. *Transactions on Software Engineering*, *SE-5*(3), 216–226. doi: 10.1109/tse.1979.234183

Sharir, M., & Pnueli, A. (1978, September). *Two approaches to interprocedural data flow analysis* (techreport). New York, NY, USA. (Issue: 002)

Stable MIR Librarification Project Group. (2023, June). *Stable mir.* Retrieved 2023-06-27, from https://github.com/rust-lang/project-stable-mir

The OWASP Foundation. (2021, November). *Owasp top ten — owasp foundation.* Retrieved 2023-11-19, from https://owasp.org/www-project-top-ten/

The Rust Project Developers. (2018, January). *Rust compiler development guide (rustc-dev-guide).* Retrieved 2023-11-01, from https://github.com/rust-lang/rustc-dev-guide

Toman, J., Pernsteiner, S., & Torlak, E. (2015, November). Crust: A bounded verifier for rust (n). In *Proceedings of the 30th international conference on automated software engineering* (p. 75–80). IEEE. doi: 10.1109/ase.2015.77

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering* (C. Wohlin, Ed.). Berlin: Springer Berlin Heidelberg.

Zhang, X., Mangal, R., Naik, M., & Yang, H. (2014). Hybrid top-down and bottom-up interprocedural analysis. In *Proceedings of the 35th acm sigplan conference on programming language design and implementation* (pp. 249–258). ACM. doi: 10.1145/2594291.2594328

A Appendix

This appendix provides additional data to increase the reproducibility of the results presented in this thesis. Appendix A1 outlines the steps to install and use the static code analysis tool developed in this thesis and Appendix A2 describes the process and the resulting data of a pre-evaluation performed in the context of this thesis that systematically analyzes all published crates in the Rust Package Registry using a dependency analysis to identify how many published crates are susceptible to each vulnerability.

A1 Installation and Usage of Cargo Flowcheck

The developed tool "cargo-flowcheck" can be installed using the following process:

1. Install Rust using Rustup.[1]
2. Install Rust Toolchain: `rustup default nightly-2023-04-16`
3. On Linux (X86_64) set shell variable
   ```
   export LD_LIBRARY_PATH=$HOME/.rustup/toolchains/nightly-2023-04-16-x86_64-
       ↪ unknown-linux-gnu/lib
   ```
4. On MacOSX (Aarch64) set shell variable
   ```
   export DYLD_LIBRARY_PATH=$HOME/.rustup/toolchains/nightly-2023-04-16-
       ↪ aarch64-apple-darwin/lib
   ```
5. Extract the zip file and change to the top-level directory `cargo-flowcheck`.
6. Execute the script `./install.sh`
7. *(Optional)* Execute the script `./check.sh` to perform a self test
8. Switch to a Rust project and run the tool: `cargo flowcheck`

To download and scan an external crate, `cargo-download`[2] can be used. The following example analyzes the `utc-0.1.0` crate:

1. Install cargo-download by executing `cargo install cargo-download`
2. Download the external crate: `cargo download -x utc==0.1.0`
3. The crate is downloaded and extracted into the subdirectory `utc-0.1.0`
4. Switch directory: `cd utc-0.1.0`
5. Run Cargo Flowcheck: `cargo flowcheck`

[1] https://www.rust-lang.org/tools/install
[2] https://crates.io/crates/cargo-download

A2 Pre-Evaluation of Vulnerable Crates using Dependency Analysis

This section outlines the process and results of a pre-evaluation as performed in the context of this thesis. This pre-evaluation systematically analyzes all published crates in the Rust Package Registry using a dependency analysis to identify which crates are susceptible to which vulnerabilities.
This pre-evaluation was performed at the beginning of the thesis and **does not** perform the data-flow analysis, which is developed in this thesis. Instead, it only uses the dependency analysis, which is provided by the previously available tool cargo-audit. The result of this pre-evaluation makes it possible to group all findings by the vulnerability and thus enables one to derive a ranking of vulnerabilities that have a high reach in the Rust ecosystem.
The objective of this pre-evaluation is to identify which vulnerabilities in the RustSec Database are found in the dependency analysis and how often they are found. For this, the raw data collected from dependency analysis is used to calculate the backward mapping from the vulnerabilities to the affected crates. Using this information, it is possible to create a ranking of vulnerabilities ordered by their number of occurrences in the Rust ecosystem, as identified by a dependency analysis.
Related work (Decan, Mens, & Grosjean, 2018) has been identified that studies the evolution of dependency networks for the Rust ecosystem but does not map dependency networks to Rust vulnerabilities.

A2.1 Metrics

For the first research objective, the *total number of vulnerable crates found by dependency analysis* in the Rust ecosystem must be obtained. This allows one to get insights on the relevance of security audits in general. The raw data acquired is then used to calculate the following metrics of the second research objective.
For the second research objective, which is to identify how often vulnerabilities are found in the Rust ecosystem, four different metrics should be used. These metrics differ in the way, transitive dependencies, and crate versions are counted related to the respective vulnerability and the crate that is marked as vulnerable.
The metric *direct dependents* "**D**" counts all vulnerable dependents of the crate that contains the vulnerability. This metric does not count each version of the published crates separately.
The metric *direct dependents, including crate versions* "**D**×**V**" counts all crate versions of all vulnerable dependents of the crate that contains the vulnerability.

The metric *transitive dependents* "**T**" counts all vulnerable transitive dependents of the crate that contains the vulnerability, but does not count each version of the crate separately.
The metric *transitive dependents, including crate versions* "**T**×**V**" counts all crate versions of all transitively vulnerable dependents of the crate that contains the vulnerability.

A2.2 Automated Download and Preprocessing of Rust Crates

This section describes the process of systematically downloading all crates the Rust Package Registry and analyzing them using a dependency analysis. The Rust Package Registry provides a git repository[3] containing an index of all published crates along with additional metadata for each published version of each crate. This index contained 817.417 crate versions at the access date of June 20, 2023. The package registry also stores the archived source code for each crate version and makes them publicly available via a REST API. The source code archives of all crates in the package registry were downloaded using the open-source tool Panamax[4].
Some archives contained invalid path names that involved backslashes as the path separator instead of forward slashes, as required for tarballs. Experiments showed that Cargo tolerates these crates and allows to use them as dependencies. Therefore, they should not be rejected and need to be included in the evaluation and a special extraction tool was created that detects and corrects these pathnames automatically.
In the case that a crate does not contain a lockfile `Cargo.lock` file, it is automatically generated by Cargo. When generating lockfiles, Cargo does not include *yanked dependencies* (crates that are marked as removed). For these cases, the hard-coded check was removed on a local copy of the Cargo source code, and then the missing lockfiles were generated. All crates have been audited using the tool `cargo-audit`.

A2.3 Results and Conclusion of Pre-Evaluation

The results of the pre-evaluation based on a dependency analysis and without a subsequent data-flow analysis are shown in Table A2-2. The affected version range of vulnerabilities was considered in the evaluation when collecting the data, but is not shown explicitly in the table. The column headers have been abbreviated and the abbreviations are described in Table A2-1. The results show detailed information on the effect that individual vulnerabilities have on the Rust ecosystem. The most prevalent vulnerabilities were discussed in Chapter 3.

[3] https://github.com/rust-lang/crates.io-index (revision `eb6c9ada5a`, accessed on June 20, 2023)
[4] https://github.com/panamax-rs/panamax

Column Name	Description
Vulnerability ID	The identifier in the RustSec Database of the vulnerability in question.
Crate	The crate that contains the vulnerability.
I	The vulnerability informational status. For normal vulnerabilities, this field is empty. For informational vulnerabilities, U means Unmaintained and N means Notice.
F	Number of affected functions specified in the vulnerability definition, or 0 if none are defined. If the kind of vulnerability is not code-related (type U or I), then the affected functions are not available (n/a).
D	Number of direct dependents of the crate that contains the vulnerability.
D×V	Number of versions of direct dependents of the crate that contains the vulnerability.
T	Number of transitive dependents of the crate that contains the vulnerability.
T×V	Number of versions of transitive dependents of the crate that contains the vulnerability.

Table A2-1: Explanation of Column Names

Table A2-2: Pre-Evaluation Results

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2021-0145	atty		0	967	9391	32641	**237160**
RUSTSEC-2020-0071	time		9	1477	10392	24525	**179415**
RUSTSEC-2021-0139	ansi_term	U	n/a	812	7210	18666	**137303**
RUSTSEC-2022-0013	regex		0	3581	24077	13787	**88587**
RUSTSEC-2023-0018	remove_dir_all		3	56	448	12321	**85713**
RUSTSEC-2022-0041	crossbeam-utils		0	243	1736	12534	**85579**
RUSTSEC-2023-0005	tokio		0	6639	40239	12690	**77040**
RUSTSEC-2023-0034	h2		0	48	503	10987	**70984**
RUSTSEC-2020-0016	net2	U	n/a	205	1658	10818	**68904**
RUSTSEC-2021-0124	tokio		1	4459	25080	11208	**68598**
RUSTSEC-2022-0078	bumpalo		1	28	230	10091	**64676**
RUSTSEC-2022-0006	thread_local		0	44	310	9962	**59864**
RUSTSEC-2020-0159	chrono		0	3785	25083	8623	**55831**
RUSTSEC-2019-0036	failure		1	3417	17698	9420	**55757**
RUSTSEC-2020-0036	failure	U	n/a	3417	17698	9420	**55757**
RUSTSEC-2020-0070	lock_api		5	41	214	8319	**55691**
RUSTSEC-2022-0022	hyper		0	2355	15446	8369	**54758**
RUSTSEC-2023-0044	openssl		1	747	6553	8328	**53657**
RUSTSEC-2021-0078	hyper		0	2326	15290	8189	**53539**
RUSTSEC-2021-0079	hyper		0	2326	15290	8189	**53539**
RUSTSEC-2020-0053	dirs	U	n/a	2275	16244	8235	**52743**
RUSTSEC-2023-0045	memoffset		1	43	257	7561	**49342**
RUSTSEC-2023-0022	openssl		1	693	6233	7646	**49244**
RUSTSEC-2023-0023	openssl		2	693	6233	7646	**49244**
RUSTSEC-2023-0024	openssl		2	693	6233	7646	**49244**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2023-0001	tokio		0	4264	22688	6779	**37356**
RUSTSEC-2022-0048	xml-rs	U	n/a	383	2663	5366	**33615**
RUSTSEC-2020-0095	difference	U	n/a	86	1176	2418	**32977**
RUSTSEC-2022-0004	rustc-serialize		1	1042	7461	5664	**32954**
RUSTSEC-2021-0127	serde_cbor	U	n/a	406	2938	4245	**31973**
RUSTSEC-2021-0093	crossbeam-deque		0	24	192	4211	**26848**
RUSTSEC-2023-0033	borsh		0	377	2462	2155	**26710**
RUSTSEC-2020-0056	stdweb	U	n/a	51	272	4143	**26358**
RUSTSEC-2018-0017	tempdir	U	n/a	441	3660	3416	**23077**
RUSTSEC-2021-0003	smallvec		1	208	1120	4513	**22812**
RUSTSEC-2020-0168	mach	U	n/a	9	33	3212	**20755**
RUSTSEC-2022-0040	owning_ref		0	99	601	2692	**20265**
RUSTSEC-2018-0015	term	U	n/a	188	2077	3559	**19299**
RUSTSEC-2020-0080	miow		0	6	9	3630	**18286**
RUSTSEC-2020-0027	traitobject		2	9	123	2607	**18103**
RUSTSEC-2021-0144	traitobject	U	n/a	9	123	2607	**18103**
RUSTSEC-2020-0077	memmap	U	n/a	412	3263	2048	**16900**
RUSTSEC-2020-0078	net2		0	38	248	3426	**16834**
RUSTSEC-2021-0119	nix		1	410	3223	2388	**15208**
RUSTSEC-2021-0020	hyper		0	444	1814	2289	**12803**
RUSTSEC-2021-0064	cpuid-bool	U	n/a	0	0	2285	**12736**
RUSTSEC-2020-0146	generic-array		0	29	118	2235	**12046**
RUSTSEC-2023-0002	git2		0	753	7162	1514	**11643**
RUSTSEC-2023-0042	ouroboros		0	72	623	868	**11313**
RUSTSEC-2023-0004	bzip2		0	114	1592	1227	**11308**
RUSTSEC-2022-0019	crossbeam-channel		0	516	2539	2444	**11295**
RUSTSEC-2021-0060	aes-soft	U	n/a	25	114	1805	**11071**
RUSTSEC-2021-0059	aesni	U	n/a	13	67	1777	**10963**
RUSTSEC-2020-0060	futures-task		1	8	24	2265	**10872**
RUSTSEC-2023-0003	libgit2-sys		0	7	194	1378	**10791**
RUSTSEC-2020-0054	directories	U	n/a	774	4986	1694	**10749**
RUSTSEC-2022-0021	crossbeam-queue		0	34	139	2553	**10724**
RUSTSEC-2020-0079	socket2		0	38	280	2153	**10155**
RUSTSEC-2020-0059	futures-util		1	190	814	2111	**9965**
RUSTSEC-2021-0072	tokio		1	1518	5880	2325	**9690**
RUSTSEC-2022-0090	libsqlite3-sys		0	60	635	1178	**9371**
RUSTSEC-2021-0141	dotenv	U	n/a	710	4043	1477	**9108**
RUSTSEC-2022-0061	parity-wasm	U	n/a	111	845	1187	**8549**
RUSTSEC-2022-0092	rmp-serde		0	376	3119	1031	**8424**
RUSTSEC-2018-0018	smallvec		0	103	566	2086	**8176**
RUSTSEC-2021-0140	rusttype	U	n/a	195	1345	1251	**7578**
RUSTSEC-2021-0153	encoding	U	n/a	211	1925	1210	**7482**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2021-0146	twoway	U	n/a	57	495	1172	**7319**
RUSTSEC-2020-0073	image		6	607	3727	1140	**7245**
RUSTSEC-2020-0163	term_size	U	n/a	178	1665	973	**7024**
RUSTSEC-2020-0144	lzw	U	n/a	13	64	1109	**7006**
RUSTSEC-2023-0006	openssl-src		0	2	11	658	**6843**
RUSTSEC-2023-0007	openssl-src		0	2	11	658	**6843**
RUSTSEC-2023-0009	openssl-src		0	2	11	658	**6843**
RUSTSEC-2023-0010	openssl-src		0	2	11	658	**6843**
RUSTSEC-2021-0076	libsecp256k1		0	85	516	1054	**6719**
RUSTSEC-2020-0091	arc-swap		1	20	75	1359	**6396**
RUSTSEC-2023-0040	users	U	n/a	163	1169	404	**5918**
RUSTSEC-2020-0026	linked-hash-map		0	54	292	1217	**5876**
RUSTSEC-2021-0130	lru		2	193	1629	769	**5792**
RUSTSEC-2023-0031	spin		0	33	192	1204	**5792**
RUSTSEC-2022-0081	json	U	n/a	486	3486	794	**5702**
RUSTSEC-2022-0070	secp256k1		1	198	1272	923	**5649**
RUSTSEC-2022-0071	rusoto_credential	U	n/a	82	479	819	**5581**
RUSTSEC-2021-0115	zeroize_derive		0	3	11	862	**5366**
RUSTSEC-2021-0073	prost-types		1	130	868	851	**5209**
RUSTSEC-2020-0061	futures-task		1	5	12	1265	**5200**
RUSTSEC-2019-0011	memoffset		0	7	37	656	**5130**
RUSTSEC-2021-0080	tar		1	184	1299	712	**5052**
RUSTSEC-2023-0028	buf_redux	U	n/a	27	186	739	**4967**
RUSTSEC-2016-0005	rust-crypto	U	n/a	554	2953	1227	**4935**
RUSTSEC-2022-0011	rust-crypto		0	554	2953	1227	**4935**
RUSTSEC-2020-0020	stb_truetype	U	n/a	4	41	766	**4742**
RUSTSEC-2021-0070	nalgebra		0	352	2393	785	**4728**
RUSTSEC-2021-0131	brotli-sys		0	3	10	674	**4707**
RUSTSEC-2022-0032	openssl-src		0	2	8	494	**4661**
RUSTSEC-2022-0020	crossbeam		0	154	992	800	**4631**
RUSTSEC-2020-0008	hyper		0	505	2082	1185	**4607**
RUSTSEC-2022-0046	rocksdb		1	120	1130	353	**4536**
RUSTSEC-2019-0039	typemap	U	n/a	69	499	760	**4244**
RUSTSEC-2022-0074	prettytable-rs		0	378	2623	623	**4130**
RUSTSEC-2020-0082	ordered-float		0	84	740	727	**4075**
RUSTSEC-2021-0065	anymap	U	n/a	74	563	490	**4004**
RUSTSEC-2021-0081	actix-http		0	99	452	533	**3679**
RUSTSEC-2022-0014	openssl-src		0	1	1	401	**3677**
RUSTSEC-2016-0001	openssl		0	71	564	598	**3294**
RUSTSEC-2018-0006	yaml-rust		0	77	630	525	**3260**
RUSTSEC-2020-0049	actix-codec		0	51	251	429	**3226**
RUSTSEC-2020-0045	actix-utils		0	39	216	422	**3166**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2020-0097	xcb		0	46	235	519	**3090**
RUSTSEC-2021-0019	xcb		0	46	235	519	**3090**
RUSTSEC-2022-0082	warp		0	285	1796	478	**3018**
RUSTSEC-2020-0048	actix-http		0	75	351	394	**3008**
RUSTSEC-2020-0162	tokio-proto	U	n/a	70	417	643	**2955**
RUSTSEC-2019-0014	image		1	284	1603	515	**2954**
RUSTSEC-2020-0018	block-cipher-trait	U	n/a	48	144	523	**2952**
RUSTSEC-2022-0029	crossbeam		0	91	521	500	**2776**
RUSTSEC-2022-0080	parity-util-mem	U	n/a	28	221	510	**2654**
RUSTSEC-2021-0095	mopa		0	37	263	519	**2613**
RUSTSEC-2021-0023	rand_core		2	12	44	935	**2599**
RUSTSEC-2022-0008	windows		0	163	496	473	**2538**
RUSTSEC-2023-0015	ascii		0	16	93	326	**2512**
RUSTSEC-2020-0006	bumpalo		0	2	51	571	**2408**
RUSTSEC-2022-0054	wee_alloc	U	n/a	180	1105	394	**2364**
RUSTSEC-2018-0001	untrusted		0	58	350	458	**2352**
RUSTSEC-2020-0158	slice-deque	U	n/a	15	54	419	**2270**
RUSTSEC-2021-0047	slice-deque		0	15	54	419	**2270**
RUSTSEC-2022-0035	websocket		0	67	410	166	**2191**
RUSTSEC-2021-0067	cranelift-codegen		0	47	794	256	**2189**
RUSTSEC-2021-0090	ash		0	33	227	444	**2040**
RUSTSEC-2019-0033	http		1	112	332	676	**2018**
RUSTSEC-2019-0034	http		1	112	332	676	**2018**
RUSTSEC-2019-0035	rand_core		2	25	60	760	**1979**
RUSTSEC-2021-0089	raw-cpuid		0	17	117	321	**1968**
RUSTSEC-2021-0013	raw-cpuid		0	15	123	309	**1944**
RUSTSEC-2021-0097	openssl-src		0	0	0	294	**1917**
RUSTSEC-2021-0098	openssl-src		0	0	0	294	**1917**
RUSTSEC-2020-0043	ws		0	104	554	241	**1800**
RUSTSEC-2022-0055	axum-core		0	15	52	299	**1794**
RUSTSEC-2021-0137	sodiumoxide	U	n/a	161	1214	240	**1759**
RUSTSEC-2020-0151	generator		0	5	66	624	**1750**
RUSTSEC-2020-0058	stream-cipher	U	n/a	29	156	366	**1749**
RUSTSEC-2020-0062	futures-util		1	67	177	579	**1651**
RUSTSEC-2020-0145	heapless		1	118	481	339	**1649**
RUSTSEC-2020-0046	actix-service		0	67	381	312	**1610**
RUSTSEC-2020-0014	rusqlite		6	180	1291	228	**1553**
RUSTSEC-2023-0037	xsalsa20poly1305	U	n/a	19	121	145	**1549**
RUSTSEC-2021-0122	flatbuffers		0	38	423	195	**1513**
RUSTSEC-2017-0002	hyper		0	282	1077	357	**1507**
RUSTSEC-2022-0056	clipboard	U	n/a	157	964	244	**1486**
RUSTSEC-2016-0002	hyper		0	279	1067	350	**1480**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2021-0091	gfx-auxil		0	0	0	339	**1469**
RUSTSEC-2021-0096	spirv_headers	U	n/a	10	71	355	**1460**
RUSTSEC-2022-0037	async-graphql		0	55	1168	78	**1401**
RUSTSEC-2023-0035	enumflags2		0	26	136	257	**1392**
RUSTSEC-2020-0100	sys-info		1	55	484	136	**1374**
RUSTSEC-2021-0021	nb-connect		0	2	17	381	**1351**
RUSTSEC-2021-0055	openssl-src		0	0	0	233	**1328**
RUSTSEC-2023-0019	kuchiki	U	n/a	84	478	235	**1312**
RUSTSEC-2022-0051	lz4-sys		0	4	27	177	**1261**
RUSTSEC-2020-0057	block-cipher	U	n/a	40	80	332	**1231**
RUSTSEC-2023-0020	const-cstr		0	40	146	253	**1208**
RUSTSEC-2021-0057	openssl-src		0	0	0	211	**1198**
RUSTSEC-2021-0058	openssl-src		0	0	0	211	**1198**
RUSTSEC-2017-0004	base64		0	58	317	212	**1186**
RUSTSEC-2022-0084	libp2p		0	48	321	186	**1103**
RUSTSEC-2020-0041	sized-chunks		0	2	18	307	**1058**
RUSTSEC-2020-0019	tokio-rustls		0	25	142	172	**1053**
RUSTSEC-2020-0096	im		0	112	458	301	**1053**
RUSTSEC-2020-0068	multihash		2	67	248	376	**1029**
RUSTSEC-2021-0134	rental	U	n/a	41	249	205	**1028**
RUSTSEC-2019-0006	ncurses		5	35	172	196	**1017**
RUSTSEC-2018-0005	serde_yaml		0	121	550	209	**975**
RUSTSEC-2020-0081	mio		0	22	122	230	**971**
RUSTSEC-2021-0044	rocket		0	139	922	147	**969**
RUSTSEC-2022-0053	mapr	U	n/a	6	56	121	**947**
RUSTSEC-2021-0061	aes-ctr	U	n/a	50	254	219	**939**
RUSTSEC-2019-0025	serde_cbor		0	70	447	159	**900**
RUSTSEC-2019-0013	spin		1	42	208	308	**891**
RUSTSEC-2023-0025	git-hash	U	n/a	20	466	67	**854**
RUSTSEC-2020-0029	rgb		0	21	88	225	**827**
RUSTSEC-2022-0075	wasmtime		0	80	518	151	**826**
RUSTSEC-2022-0076	wasmtime		2	80	518	151	**826**
RUSTSEC-2020-0052	crossbeam-channel		0	43	119	300	**820**
RUSTSEC-2021-0126	rust-embed		0	90	547	119	**786**
RUSTSEC-2020-0147	rulinalg	U	n/a	18	107	111	**758**
RUSTSEC-2022-0068	capnp		0	40	322	186	**733**
RUSTSEC-2019-0026	sodiumoxide		2	65	632	84	**715**
RUSTSEC-2020-0031	tiny_http		0	79	294	164	**705**
RUSTSEC-2021-0147	daemonize	U	n/a	71	358	100	**682**
RUSTSEC-2021-0037	diesel		1	93	517	124	**673**
RUSTSEC-2020-0122	beef		0	14	108	124	**667**
RUSTSEC-2019-0040	boxfnonce	U	n/a	11	88	104	**653**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2021-0056	openssl-src		0	0	0	148	**637**
RUSTSEC-2020-0023	rulinalg		2	16	80	99	**594**
RUSTSEC-2021-0041	parse_duration		1	37	276	66	**584**
RUSTSEC-2020-0009	flatbuffers		2	22	181	88	**513**
RUSTSEC-2020-0002	prost		0	83	372	129	**494**
RUSTSEC-2023-0036	tree_magic	U	n/a	34	318	69	**491**
RUSTSEC-2020-0025	bigint	U	n/a	40	150	103	**480**
RUSTSEC-2021-0103	molecule		0	14	72	142	**479**
RUSTSEC-2019-0012	smallvec		1	8	16	206	**458**
RUSTSEC-2021-0116	arrow		0	44	248	80	**453**
RUSTSEC-2021-0117	arrow		0	44	248	80	**453**
RUSTSEC-2021-0118	arrow		0	44	248	80	**453**
RUSTSEC-2021-0128	rusqlite		7	64	225	94	**453**
RUSTSEC-2019-0009	smallvec		1	8	16	203	**451**
RUSTSEC-2022-0038	juniper		0	49	356	66	**449**
RUSTSEC-2021-0135	tower-http		0	13	47	50	**444**
RUSTSEC-2022-0043	tower-http		0	13	47	50	**444**
RUSTSEC-2020-0003	rust_sodium	U	n/a	24	214	105	**442**
RUSTSEC-2020-0044	atom		0	1	1	106	**434**
RUSTSEC-2023-0026	git-path	U	n/a	12	184	51	**432**
RUSTSEC-2019-0019	blake2		0	42	290	74	**419**
RUSTSEC-2023-0039	buffered-reader		0	8	128	38	**409**
RUSTSEC-2023-0029	nats		0	20	156	44	**403**
RUSTSEC-2021-0151	ncollide2d	U	n/a	15	200	32	**401**
RUSTSEC-2020-0093	async-h1		2	14	48	76	**399**
RUSTSEC-2019-0005	pancurses		2	44	253	68	**396**
RUSTSEC-2021-0150	ncollide3d	U	n/a	18	168	51	**382**
RUSTSEC-2023-0043	ftp	U	n/a	12	98	44	**376**
RUSTSEC-2022-0003	ammonia		1	15	84	72	**373**
RUSTSEC-2021-0110	wasmtime		3	36	207	80	**366**
RUSTSEC-2021-0062	miscreant	U	n/a	14	249	22	**359**
RUSTSEC-2021-0054	rkyv		1	11	286	14	**357**
RUSTSEC-2021-0063	comrak		0	48	265	59	**331**
RUSTSEC-2022-0044	markdown	U	n/a	32	291	35	**316**
RUSTSEC-2022-0077	claim	U	n/a	3	18	41	**316**
RUSTSEC-2023-0014	cortex-m-rt		0	83	214	117	**315**
RUSTSEC-2023-0041	trust-dns-server		0	24	152	41	**309**
RUSTSEC-2020-0015	openssl-src		0	0	0	63	**301**
RUSTSEC-2021-0026	comrak		0	44	242	53	**301**
RUSTSEC-2021-0114	nanorand		1	10	49	70	**298**
RUSTSEC-2021-0011	fil-ocl		0	0	0	22	**297**
RUSTSEC-2023-0008	openssl-src		0	2	11	80	**287**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2023-0011	openssl-src		0	2	11	80	**287**
RUSTSEC-2023-0012	openssl-src		0	2	11	80	**287**
RUSTSEC-2023-0013	openssl-src		0	2	11	80	**287**
RUSTSEC-2018-0014	chan	U	n/a	48	229	67	**285**
RUSTSEC-2022-0059	openssl-src		0	2	11	80	**283**
RUSTSEC-2022-0064	openssl-src		0	2	11	80	**283**
RUSTSEC-2022-0065	openssl-src		0	2	11	80	**283**
RUSTSEC-2021-0113	metrics-util		0	17	96	59	**282**
RUSTSEC-2022-0001	lmdb	U	n/a	16	185	29	**268**
RUSTSEC-2019-0027	libsecp256k1		1	19	68	94	**264**
RUSTSEC-2020-0028	rocket		1	66	261	66	**261**
RUSTSEC-2022-0025	openssl-src		0	1	1	79	**255**
RUSTSEC-2022-0026	openssl-src		0	1	1	79	**255**
RUSTSEC-2022-0027	openssl-src		0	1	1	79	**255**
RUSTSEC-2019-0037	pnet		1	43	170	62	**254**
RUSTSEC-2023-0027	async-nats		0	17	102	50	**252**
RUSTSEC-2019-0032	crust	U	n/a	2	119	14	**251**
RUSTSEC-2020-0167	pnet_packet		0	10	31	67	**251**
RUSTSEC-2021-0149	nphysics2d	U	n/a	6	116	9	**242**
RUSTSEC-2021-0129	openssl-src		0	1	1	76	**239**
RUSTSEC-2020-0098	rusb		0	24	89	52	**234**
RUSTSEC-2021-0005	glsl-layout		0	12	59	45	**233**
RUSTSEC-2021-0120	abomonation		0	17	184	24	**227**
RUSTSEC-2021-0106	bat		0	26	122	44	**223**
RUSTSEC-2020-0161	array-macro		0	4	22	47	**222**
RUSTSEC-2020-0021	rio		0	8	60	22	**218**
RUSTSEC-2018-0007	trust-dns-proto		0	8	15	50	**215**
RUSTSEC-2022-0049	iana-time-zone		1	1	3	125	**212**
RUSTSEC-2020-0007	bitvec		1	33	104	67	**208**
RUSTSEC-2023-0038	sequoia-openpgp		0	28	187	36	**208**
RUSTSEC-2021-0035	quinn		0	8	59	15	**200**
RUSTSEC-2016-0004	libusb	U	n/a	38	133	52	**197**
RUSTSEC-2019-0003	protobuf		1	35	166	44	**195**
RUSTSEC-2021-0001	mdbook		0	41	176	43	**193**
RUSTSEC-2022-0028	neon		2	35	184	40	**192**
RUSTSEC-2016-0003	portaudio		0	20	88	33	**182**
RUSTSEC-2020-0149	appendix		0	6	41	37	**175**
RUSTSEC-2022-0002	dashmap		14	28	121	50	**172**
RUSTSEC-2022-0063	linked_list_allocator		0	20	112	37	**167**
RUSTSEC-2017-0001	sodiumoxide		0	17	123	21	**166**
RUSTSEC-2021-0015	calamine		0	15	116	18	**155**
RUSTSEC-2022-0083	evm		0	39	105	56	**155**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2023-0046	cyfs-base		0	15	146	15	**146**
RUSTSEC-2020-0113	atomic-option		0	7	61	23	**145**
RUSTSEC-2017-0007	lz4-compress	U	n/a	14	74	27	**144**
RUSTSEC-2020-0076	routing	U	n/a	12	131	13	**134**
RUSTSEC-2020-0123	libp2p-deflate		0	1	3	29	**129**
RUSTSEC-2021-0069	lettre		3	22	91	28	**127**
RUSTSEC-2020-0072	futures-intrusive		0	18	51	39	**121**
RUSTSEC-2021-0048	stackvector		0	0	0	41	**121**
RUSTSEC-2022-0072	hyper-staticfile		0	21	104	26	**121**
RUSTSEC-2019-0028	flatbuffers		1	12	73	27	**118**
RUSTSEC-2022-0050	interledger-packet	U	n/a	28	102	34	**118**
RUSTSEC-2022-0069	hyper-staticfile		0	20	103	25	**117**
RUSTSEC-2020-0064	ffi_utils	U	n/a	7	101	7	**113**
RUSTSEC-2020-0067	quic-p2p	U	n/a	4	39	9	**113**
RUSTSEC-2018-0019	actix-web		0	25	87	32	**111**
RUSTSEC-2018-0016	quickersort	U	n/a	9	69	19	**106**
RUSTSEC-2021-0074	ammonia		0	7	34	31	**103**
RUSTSEC-2022-0015	pty	U	n/a	7	51	12	**101**
RUSTSEC-2020-0111	may_queue		0	1	45	11	**99**
RUSTSEC-2021-0136	sass-rs	U	n/a	11	85	14	**98**
RUSTSEC-2020-0086	safe_core	U	n/a	8	92	8	**92**
RUSTSEC-2019-0017	once_cell		4	20	56	33	**88**
RUSTSEC-2021-0017	postscript		0	4	62	11	**85**
RUSTSEC-2021-0142	dotenv_codegen	U	n/a	20	56	28	**82**
RUSTSEC-2019-0004	libp2p-core		0	16	73	18	**79**
RUSTSEC-2020-0063	safe-nd	U	n/a	8	78	8	**78**
RUSTSEC-2020-0140	model		0	0	0	8	**78**
RUSTSEC-2021-0071	grep-cli		1	8	43	15	**78**
RUSTSEC-2021-0148	nphysics3d	U	n/a	6	78	6	**78**
RUSTSEC-2020-0065	fake_clock	U	n/a	3	21	12	**75**
RUSTSEC-2022-0085	matrix-sdk-crypto		0	4	10	25	**75**
RUSTSEC-2022-0034	pkcs11		0	7	59	10	**72**
RUSTSEC-2019-0010	libflate		1	6	6	43	**69**
RUSTSEC-2021-0004	lazy-init		0	6	21	15	**69**
RUSTSEC-2021-0066	evm-core		0	3	49	5	**62**
RUSTSEC-2020-0128	cache		0	1	21	6	**61**
RUSTSEC-2021-0006	cache		0	1	21	6	**61**
RUSTSEC-2022-0073	alloc-cortex-m	U	n/a	12	46	17	**61**
RUSTSEC-2020-0069	lettre		3	12	37	17	**58**
RUSTSEC-2019-0001	ammonia		3	6	15	18	**56**
RUSTSEC-2020-0109	stderr	U	n/a	5	56	5	**56**
RUSTSEC-2020-0143	multiqueue		0	9	44	15	**56**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2022-0010	enum-map		0	0	0	21	**53**
RUSTSEC-2022-0088	tauri		0	7	22	9	**53**
RUSTSEC-2018-0002	tar		0	5	17	17	**52**
RUSTSEC-2021-0121	crypto2		3	2	11	5	**52**
RUSTSEC-2020-0084	safe_authenticator	U	n/a	3	45	4	**50**
RUSTSEC-2019-0002	slice-deque		0	7	16	18	**49**
RUSTSEC-2022-0012	arrow2		0	2	2	24	**49**
RUSTSEC-2021-0038	fltk		3	5	37	8	**48**
RUSTSEC-2022-0024	double-checked-cell	U	n/a	4	36	8	**48**
RUSTSEC-2022-0087	slack-morphism		0	2	48	2	**48**
RUSTSEC-2022-0086	slack-morphism		0	2	47	2	**47**
RUSTSEC-2020-0115	ruspiro-singleton		0	7	34	9	**44**
RUSTSEC-2020-0136	toolshed		0	8	38	11	**44**
RUSTSEC-2021-0029	truetype		0	2	40	4	**44**
RUSTSEC-2022-0052	os_socketaddr		0	5	26	10	**44**
RUSTSEC-2018-0013	safe-transmute		0	1	6	2	**43**
RUSTSEC-2020-0035	chunky		0	1	43	1	**43**
RUSTSEC-2020-0101	conquer-once		0	5	41	5	**41**
RUSTSEC-2020-0066	safe_bindgen	U	n/a	0	0	5	**40**
RUSTSEC-2020-0089	nanorand		0	3	6	17	**39**
RUSTSEC-2020-0137	lever		0	6	13	13	**39**
RUSTSEC-2022-0039	odbc	U	n/a	4	26	8	**37**
RUSTSEC-2020-0131	rcu_cell		0	1	12	7	**36**
RUSTSEC-2020-0051	rustsec		0	5	26	8	**35**
RUSTSEC-2021-0132	compu-brotli-sys		0	1	12	5	**34**
RUSTSEC-2018-0022	temporary		0	0	0	8	**33**
RUSTSEC-2022-0009	libp2p-core		1	10	23	15	**32**
RUSTSEC-2021-0010	containers		0	5	20	9	**31**
RUSTSEC-2020-0004	lucet-runtime-internals		0	5	23	5	**29**
RUSTSEC-2020-0005	cbox		0	2	28	2	**28**
RUSTSEC-2020-0139	dces		0	11	18	12	**26**
RUSTSEC-2022-0060	orbtk	U	n/a	3	20	6	**26**
RUSTSEC-2021-0075	ark-r1cs-std		1	20	22	22	**24**
RUSTSEC-2020-0074	pyo3		0	8	20	8	**20**
RUSTSEC-2020-0112	buttplug		0	3	18	4	**20**
RUSTSEC-2020-0153	bite		0	1	20	1	**20**
RUSTSEC-2021-0009	basic_dsp_matrix		0	1	20	1	**20**
RUSTSEC-2021-0112	tectonic_xdv		0	1	13	4	**19**
RUSTSEC-2021-0143	kamadak-exif		1	3	18	4	**19**
RUSTSEC-2022-0066	conduit-hyper		0	0	0	3	**18**
RUSTSEC-2020-0038	ordnung		0	1	17	1	**17**
RUSTSEC-2020-0083	safe_app	U	n/a	2	17	2	**17**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2021-0083	derive-com-impl		1	0	0	9	**16**
RUSTSEC-2021-0034	office	U	n/a	1	9	2	**15**
RUSTSEC-2021-0111	tremor-script		0	2	14	3	**15**
RUSTSEC-2022-0091	tauri		0	6	15	6	**15**
RUSTSEC-2018-0012	orion		0	0	0	1	**14**
RUSTSEC-2021-0049	through		0	3	7	4	**14**
RUSTSEC-2021-0088	csv-sniffer		0	3	13	4	**14**
RUSTSEC-2020-0138	lexer		0	2	12	2	**12**
RUSTSEC-2021-0007	av-data		0	5	12	5	**12**
RUSTSEC-2021-0025	jsonrpc-quic	U	n/a	1	12	1	**12**
RUSTSEC-2022-0016	wasmtime		1	5	8	7	**12**
RUSTSEC-2022-0036	r2d2_odbc	U	n/a	2	11	3	**12**
RUSTSEC-2018-0020	libpulse-binding		0	2	11	2	**11**
RUSTSEC-2019-0038	libpulse-binding		0	2	11	2	**11**
RUSTSEC-2020-0039	simple-slab		0	2	11	2	**11**
RUSTSEC-2021-0002	interfaces2	U	n/a	2	11	2	**11**
RUSTSEC-2021-0028	toodee		1	1	11	1	**11**
RUSTSEC-2023-0021	stb_image		0	2	8	5	**11**
RUSTSEC-2017-0006	rmpv		0	3	10	3	**10**
RUSTSEC-2020-0075	branca		2	2	10	2	**10**
RUSTSEC-2022-0018	totp-rs		1	1	10	1	**10**
RUSTSEC-2020-0119	ticketed_lock		0	2	5	3	**9**
RUSTSEC-2019-0007	asn1_der		0	1	3	3	**8**
RUSTSEC-2019-0023	string-interner		0	1	2	5	**8**
RUSTSEC-2020-0118	tiny_future		0	1	8	1	**8**
RUSTSEC-2021-0018	qwutils		1	4	8	4	**8**
RUSTSEC-2019-0022	portaudio-rs		0	0	0	3	**7**
RUSTSEC-2020-0001	trust-dns-server		0	2	4	3	**7**
RUSTSEC-2021-0100	sha2		0	2	5	3	**7**
RUSTSEC-2022-0030	rulex		0	2	7	2	**7**
RUSTSEC-2022-0031	rulex		0	2	7	2	**7**
RUSTSEC-2022-0067	lzf		2	2	4	3	**7**
RUSTSEC-2020-0012	os_str_bytes		0	3	6	3	**6**
RUSTSEC-2020-0024	tough		0	2	6	2	**6**
RUSTSEC-2020-0050	dync		0	1	4	2	**6**
RUSTSEC-2021-0068	iced-x86		1	3	6	3	**6**
RUSTSEC-2021-0084	bronzedb-protocol		0	3	4	5	**6**
RUSTSEC-2023-0032	ntru		2	0	0	2	**6**
RUSTSEC-2019-0030	streebog		0	1	5	1	**5**
RUSTSEC-2020-0011	plutonium	N	n/a	0	0	3	**5**
RUSTSEC-2020-0092	concread		0	2	3	3	**5**
RUSTSEC-2021-0033	stack_dst		1	0	0	1	**5**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2022-0005	ftd2xx-embedded-hal	U	n/a	0	0	2	**5**
RUSTSEC-2018-0003	smallvec		0	1	2	3	**4**
RUSTSEC-2020-0032	alpm-rs		0	1	4	1	**4**
RUSTSEC-2020-0087	try-mutex		0	2	4	2	**4**
RUSTSEC-2021-0039	endian_trait		0	1	2	1	**4**
RUSTSEC-2021-0087	columnar		0	2	4	2	**4**
RUSTSEC-2022-0057	badge	U	n/a	2	4	2	**4**
RUSTSEC-2020-0033	alg_ds		0	2	3	2	**3**
RUSTSEC-2020-0037	crayon		0	1	3	1	**3**
RUSTSEC-2021-0027	bam		1	1	3	1	**3**
RUSTSEC-2021-0043	uu_od		0	0	0	1	**3**
RUSTSEC-2021-0094	rdiff		0	1	3	1	**3**
RUSTSEC-2021-0125	simple_asn1		0	1	1	3	**3**
RUSTSEC-2022-0017	array-macro		0	2	3	2	**3**
RUSTSEC-2022-0079	elf_rs		0	2	3	2	**3**
RUSTSEC-2020-0010	tiberius	U	n/a	2	2	2	**2**
RUSTSEC-2020-0017	internment		1	1	2	1	**2**
RUSTSEC-2020-0130	bunch		0	1	2	1	**2**
RUSTSEC-2020-0155	acc_reader		0	2	2	2	**2**
RUSTSEC-2021-0030	scratchpad		1	0	0	1	**2**
RUSTSEC-2021-0031	nano_arena		2	1	2	1	**2**
RUSTSEC-2021-0036	internment		0	1	2	1	**2**
RUSTSEC-2022-0007	qcell		0	1	1	2	**2**
RUSTSEC-2022-0062	matrix-sdk		0	1	2	1	**2**
RUSTSEC-2018-0004	claxon		0	1	1	1	**1**
RUSTSEC-2019-0020	generator		0	1	1	1	**1**
RUSTSEC-2020-0099	aovec		0	0	0	1	**1**
RUSTSEC-2020-0102	late-static		0	1	1	1	**1**
RUSTSEC-2020-0106	multiqueue2		0	1	1	1	**1**
RUSTSEC-2020-0107	hashconsing		0	1	1	1	**1**
RUSTSEC-2020-0116	unicycle		0	0	0	1	**1**
RUSTSEC-2020-0124	async-coap		0	1	1	1	**1**
RUSTSEC-2020-0126	signal-simple		0	1	1	1	**1**
RUSTSEC-2020-0134	parc		0	1	1	1	**1**
RUSTSEC-2020-0165	mozjpeg		1	0	0	1	**1**
RUSTSEC-2021-0046	telemetry		0	0	0	1	**1**
RUSTSEC-2021-0085	binjs_io		0	1	1	1	**1**
RUSTSEC-2023-0017	maligned		4	1	1	1	**1**
RUSTSEC-2016-0006	cassandra	U	n/a	0	0	0	**0**
RUSTSEC-2017-0003	security-framework		0	0	0	0	**0**
RUSTSEC-2017-0005	cookie		0	0	0	0	**0**
RUSTSEC-2018-0008	slice-deque		0	0	0	0	**0**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2018-0009	crossbeam		0	0	0	0	**0**
RUSTSEC-2018-0010	openssl		0	0	0	0	**0**
RUSTSEC-2018-0011	arrayfire		0	0	0	0	**0**
RUSTSEC-2018-0021	libpulse-binding		2	0	0	0	**0**
RUSTSEC-2019-0008	simd-json		0	0	0	0	**0**
RUSTSEC-2019-0015	compact_arena		1	0	0	0	**0**
RUSTSEC-2019-0016	chttp		0	0	0	0	**0**
RUSTSEC-2019-0018	renderdoc		2	0	0	0	**0**
RUSTSEC-2019-0021	linea		0	0	0	0	**0**
RUSTSEC-2019-0024	rustsec-example-crate		0	0	0	0	**0**
RUSTSEC-2019-0029	chacha20		0	0	0	0	**0**
RUSTSEC-2019-0031	spin	U	n/a	0	0	0	**0**
RUSTSEC-2020-0013	fake-static		0	0	0	0	**0**
RUSTSEC-2020-0022	ozone		0	0	0	0	**0**
RUSTSEC-2020-0030	mozwire		0	0	0	0	**0**
RUSTSEC-2020-0034	arr		0	0	0	0	**0**
RUSTSEC-2020-0040	obstack		0	0	0	0	**0**
RUSTSEC-2020-0042	stack		0	0	0	0	**0**
RUSTSEC-2020-0047	array-queue		0	0	0	0	**0**
RUSTSEC-2020-0055	libpulse-binding		0	0	0	0	**0**
RUSTSEC-2020-0085	safe_vault	U	n/a	0	0	0	**0**
RUSTSEC-2020-0088	magnetic		0	0	0	0	**0**
RUSTSEC-2020-0090	thex		0	0	0	0	**0**
RUSTSEC-2020-0094	reffers		0	0	0	0	**0**
RUSTSEC-2020-0103	autorand		0	0	0	0	**0**
RUSTSEC-2020-0104	gfwx		0	0	0	0	**0**
RUSTSEC-2020-0105	abi_stable		0	0	0	0	**0**
RUSTSEC-2020-0108	eventio		0	0	0	0	**0**
RUSTSEC-2020-0114	va-ts		0	0	0	0	**0**
RUSTSEC-2020-0117	conqueue		0	0	0	0	**0**
RUSTSEC-2020-0120	libsbc		0	0	0	0	**0**
RUSTSEC-2020-0121	abox		0	0	0	0	**0**
RUSTSEC-2020-0125	convec		0	0	0	0	**0**
RUSTSEC-2020-0127	v9		0	0	0	0	**0**
RUSTSEC-2020-0129	kekbit		0	0	0	0	**0**
RUSTSEC-2020-0132	array-tools		0	0	0	0	**0**
RUSTSEC-2020-0133	scottqueue		0	0	0	0	**0**
RUSTSEC-2020-0135	slock		0	0	0	0	**0**
RUSTSEC-2020-0141	noise_search		0	0	0	0	**0**
RUSTSEC-2020-0142	syncpool		0	0	0	0	**0**
RUSTSEC-2020-0148	cgc		0	0	0	0	**0**
RUSTSEC-2020-0150	disrustor		0	0	0	0	**0**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2020-0152	max7301		0	0	0	0	**0**
RUSTSEC-2020-0154	buffoon		0	0	0	0	**0**
RUSTSEC-2020-0156	libsecp256k1-rs		0	0	0	0	**0**
RUSTSEC-2020-0157	vm-memory		0	0	0	0	**0**
RUSTSEC-2020-0160	shamir		0	0	0	0	**0**
RUSTSEC-2020-0164	cell-project		2	0	0	0	**0**
RUSTSEC-2020-0166	personnummer	N	n/a	0	0	0	**0**
RUSTSEC-2021-0008	bra		0	0	0	0	**0**
RUSTSEC-2021-0012	cdr		0	0	0	0	**0**
RUSTSEC-2021-0014	marc		0	0	0	0	**0**
RUSTSEC-2021-0016	ms3d		0	0	0	0	**0**
RUSTSEC-2021-0022	yottadb		4	0	0	0	**0**
RUSTSEC-2021-0024	safe-api	U	n/a	0	0	0	**0**
RUSTSEC-2021-0032	byte_struct		0	0	0	0	**0**
RUSTSEC-2021-0040	arenavec		0	0	0	0	**0**
RUSTSEC-2021-0042	insert_many		0	0	0	0	**0**
RUSTSEC-2021-0045	adtensor		0	0	0	0	**0**
RUSTSEC-2021-0050	reorder		0	0	0	0	**0**
RUSTSEC-2021-0051	outer_cgi		0	0	0	0	**0**
RUSTSEC-2021-0052	id-map		0	0	0	0	**0**
RUSTSEC-2021-0053	algorithmica		0	0	0	0	**0**
RUSTSEC-2021-0077	better-macro		1	0	0	0	**0**
RUSTSEC-2021-0082	vec-const		0	0	0	0	**0**
RUSTSEC-2021-0086	flumedb		0	0	0	0	**0**
RUSTSEC-2021-0092	messagepack-rs		0	0	0	0	**0**
RUSTSEC-2021-0099	cosmos_sdk	U	n/a	0	0	0	**0**
RUSTSEC-2021-0101	pleaser		0	0	0	0	**0**
RUSTSEC-2021-0102	pleaser		0	0	0	0	**0**
RUSTSEC-2021-0104	pleaser		0	0	0	0	**0**
RUSTSEC-2021-0105	git-delta		0	0	0	0	**0**
RUSTSEC-2021-0107	ckb		0	0	0	0	**0**
RUSTSEC-2021-0108	ckb		0	0	0	0	**0**
RUSTSEC-2021-0109	ckb		0	0	0	0	**0**
RUSTSEC-2021-0123	fruity		4	0	0	0	**0**
RUSTSEC-2021-0133	cargo-download	U	n/a	0	0	0	**0**
RUSTSEC-2021-0138	mz-avro		0	0	0	0	**0**
RUSTSEC-2021-0152	out-reference		1	0	0	0	**0**
RUSTSEC-2022-0023	static_type_map	U	n/a	0	0	0	**0**
RUSTSEC-2022-0033	openssl-src		0	0	0	0	**0**
RUSTSEC-2022-0042	rustdecimal		0	0	0	0	**0**
RUSTSEC-2022-0045	oqs		0	0	0	0	**0**
RUSTSEC-2022-0047	oqs		0	0	0	0	**0**

Continued on next page

Table A2-2 – continued from previous page

Vulnerability ID	Crate	I	F	D	D×V	T	T×V
RUSTSEC-2022-0058	inconceivable	N	n/a	0	0	0	**0**
RUSTSEC-2022-0089	aliyun-oss-client		0	0	0	0	**0**
RUSTSEC-2023-0016	partial_sort		0	0	0	0	**0**
RUSTSEC-2023-0030	versionize		0	0	0	0	**0**
CVE-2015-20001	std		0	n/a	n/a	n/a	**n/a**
CVE-2017-20004	std		0	n/a	n/a	n/a	**n/a**
CVE-2018-1000622	rustdoc		0	n/a	n/a	n/a	**n/a**
CVE-2018-1000657	std		1	n/a	n/a	n/a	**n/a**
CVE-2018-1000810	std		1	n/a	n/a	n/a	**n/a**
CVE-2018-25008	std		0	n/a	n/a	n/a	**n/a**
CVE-2019-1010299	std		0	n/a	n/a	n/a	**n/a**
CVE-2019-12083	std		0	n/a	n/a	n/a	**n/a**
CVE-2019-16760	cargo		0	n/a	n/a	n/a	**n/a**
CVE-2020-36317	std		0	n/a	n/a	n/a	**n/a**
CVE-2020-36318	std		0	n/a	n/a	n/a	**n/a**
CVE-2020-36323	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-28875	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-28876	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-28877	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-28878	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-28879	std		0	n/a	n/a	n/a	**n/a**
CVE-2021-29922	std		3	n/a	n/a	n/a	**n/a**
CVE-2021-31162	std		0	n/a	n/a	n/a	**n/a**
CVE-2022-21658	std		1	n/a	n/a	n/a	**n/a**