RustGuard: Detecting Rust Data Leak Issues with Context-Sensitive Static Taint Analysis

Shanlin Deng, Mingliang Liu, Si Wu, and Baojian Hua(✉)

School of Software Engineering, Suzhou Institute for Advanced Research
University of Science and Technology of China, Suzhou 215123, China
{dengshanlin, liumingliang, wusi98}@mail.ustc.edu.cn, bjhua@ustc.edu.cn

Abstract. Rust is a promising language by providing strong safety guarantees through its advanced features including borrowing semantics and lifetime checking, and has been adopted in security-critical domains. However, Rust programs may still be vulnerable to sensitive data leak issues due to its lack of information flow checking capabilities. As a result, these data leaks undermine Rust's strong security guarantees.
In this paper, to fill the current gap, we propose a novel information flow checking approach for Rust language by leveraging static taint analysis, to detect potential data leak issues. We first propose an approach to annotate sensitive data within Rust programs by utilizing Rust's macro features. We then design an information flow checking algorithm based on static taint analysis, in which we use tainted abstract domains to model data sensitivity and use transfer functions to model the data flow. Furthermore, we design a context-sensitive algorithm to track the propagation of tainted values across procedure boundaries by leveraging a functional approach. We implement our approach in a software prototype RustGuard by extending Rust's official `rustc` compiler and conduct extensive evaluations with it. Our evaluation results demonstrate that our approach achieves precision and recall both of 91.67%, while introducing only an additional 14.07% runtime overhead and negligible memory consumption to detect data leak issues. Moreover, compared with the state-of-the-art approach *Cocoon*, our approach achieves stronger usability by requiring few program modifications.

Keywords: Rust · Data Leak · Taint Analysis.

1 Introduction

Software failures or vulnerabilities may lead to devastating consequences, particularly in security-critical scenarios [23]. Safe programming languages are essential to prevent vulnerabilities by ruling out many security issues at an early development stage [38]. Rust is a promising safe programming language providing both strong security guarantees by synthesizing decades of research results and practical experience from programming language design. Specifically, Rust guarantees strong memory and concurrency safety by incorporating novel and advanced

abstractions including ownership models and lifetime tracking, adhering to zero-cost abstraction principles [11]. These security guarantees have led to increasing popularity of Rust in the past several years [12], especially in security-critical domains including operating system kernels [8,10,32], browser engines [22], and blockchain protocols [4,7].

Unfortunately, the security guarantees provided by Rust are not a silver bullet, and Rust programs still suffer from sensitive data leak issues [3]. Here a sensitive data leak refers to confidential data are accidentally or intentionally distributed to unauthorized entities, posing a significant threat to data integrity [41]. For instance, the recently reported the Wormhole vulnerability [2, 13] in the rising Solana smart contract developed with Rust resulted in financial losses exceeding $320 million. Compounding this issue, Rust developers struggle to detect data leak issues like Wormhole, due to the lack of both security guarantees of data integrity in Rust and effective detection approaches for Rust [21]. These factors underscore the critical need for detecting sensitive data leak in Rust.

Recognizing this criticality and urgency, researchers have conducted significant studies to enhance Rust security. First, extensive research has focused on fundamental aspects of Rust security [16, 17, 27, 33, 40, 48]. However, these efforts overlooked sensitive data leak issues in Rust, because they have focused on Rust memory and concurrency security vulnerabilities. Second, while some approaches have been proposed to track information flows to detect data leaks for other languages like Java [15, 44, 47], techniques for Rust are still lacking. Third, it remains unclear how to adapt existing approaches for other languages to Rust, due to the dramatic syntactic and semantic discrepancies between Rust and other languages. Finally, some recent studies [31] propose to incorporate information flow control (IFC) mechanisms [39] into Rust. Unfortunately, such incorporations incur not only compatibility issues but also significant migration costs of legacy code, as they made invasive modifications to Rust's official syntax.

In this paper, to fill the present gap, we propose a novel approach to detect data leak issues in Rust based on static taint analysis. Our key research goal is to propose an automated, lightweight, and cost-effective approach that is of practical end-user usability to detect data leak issues within Rust programs. Guided by this goal, we first propose a syntactic approach to annotate data sensitivity in Rust programs, by utilizing Rust macros [6]. We then establish a formal language model termed RITA (Rust Intermediate for Taint Analysis) to formalize Rust's core syntax. We next design an abstract taint domain with tainted data states and transfer functions to formalize tainted data flows on RITA. Finally, we design a context-sensitive inter-procedural data flow analysis algorithm to precisely traces tainted data across procedural boundaries.

During the whole process, three technical challenges must be tackled. **C1:** the strict type enforcement in Rust makes taint annotations while maintaining code compatibility challenging. To address this challenge, we leverage a distinct Rust feature, the Rust macros, to annotate taint directly at source level to ensure type validity without introducing any potential compatibility issues. **C2:** the unique ownership mechanism of Rust makes taint analysis challenging. Our

solution establishes a new Rust intermediate representation RITA as the foundational layer for the analysis and utilizes the official `rustc` compiler to translate the Rust source code to RITA. Specifically, our analysis algorithm processes Rust programs after the standard borrow checking of ownership, enabling more effective data flow analysis by avoiding the complex interleaving of ownership checking. **C3:** Rust's advanced features (e.g., trait [5]) complicate the global control flows, making it challenging to track taint information propagation globally. To overcome this, we propose a context-sensitive inter-procedural flow analysis with a functional approach to discriminate tainted data at each procedure call site, thereby enhancing the overall precision of the analysis.

We implement a prototype RustGuard for our approach and evaluate it on micro-benchmarks as well as on large Rust projects and real-world CVEs. Experimental results demonstrate that our approach reaches 91.67% precision and recall in detecting data leak issues, with a compile time overhead of 14.07% and negligible memory consumption, but without any runtime overhead. Furthermore, our approach can detect existing CVEs and outperform existing state-of-the-art approach Cocoon [31].

Contribution. To summarize, we take a new step towards enhancing Rust security by detecting data leak issues with context-sensitive static taint analysis. And our work makes the following contributions:

- We present a novel approach to detect Rust data leak issues with a context-sensitive static taint analysis.
- We implement a software prototype RustGuard for our approach by extending the official `rustc` compiler.
- We conduct extensive evaluations to demonstrate the effectiveness, efficiency, and practical usability of our approach, surpassing state-of-the-art.

The rest of this paper is organized as follows. Section 2 presents our motivation and challenges. Section 3 describes the approach to perform the analysis. Section 4 introduces the evaluations results. Section 5 discusses the limitations and our future work. Section 6 presents the related work, and Section 7 concludes.

2 Motivation and Challenges

In this section, we present the motivation (§ 2.1) and the technical challenges (§ 2.2) for this study.

2.1 Motivation

Although Rust is designed with security mechanisms to ensure memory and thread safety, it remains vulnerable to data leak risks. To better illustrate such issues, we present in Fig. 1 a sample program comprising a data leak issue that we adapted from CVE-2022-31162 [3] in Slack Morphism for Rust [1], a modern and widely used async client library for Rust. This CVE originates from

Fig. 1. Slack Morphism for Rust contains a data leak issue CVE-2022-31162 [3].

insecure debugging practices, and is classified as a high severity issue with a score of 7.5. Specifically, the variable `uri_str` at line 9 of the code represents a URI returned from the function `input`, which contains potentially sensitive information that should not be leaked. Unfortunately, the reference `&uri_str` at line 11 is passed to the function `output`, which is then used by the `debug!` macro at line 27 that triggers a sensitive data leak at line 30, because this sensitive URI is inadvertently output to a debug log.

Meanwhile, the Slack Morphism CVE is not unique. As another example, Solana [46] is a rising smart contract platform developed with Rust to resolve ETH's long-standing speed limit, and is considered to be the world's fastest blockchain. Recently, Solana is reported to contain the Wormhole vulnerability [2, 13] due to the lack of effective data integrity checking, culminating in $320 million in asset losses. In the coming decade, with the ever increasing adoption of Rust in security critical domains like blockchains, sensitive data leak issues in Rust continue to proliferate.

Unfortunately, effective approaches for detecting data leak issues in Rust remain underdeveloped. First, the builtin safety mechanisms provided by Rust cannot detect data leak issues because these mechanisms focus on memory and thread safety instead of data integrity. As a result, Rust's builtin safety mechanisms cannot detect the CVE in Fig. 1, as it does not violate any of Rust's safety rules. Second, existing data flow control approaches [31, 39] cannot directly detect such CVEs like the one in Fig. 1, because they require extensive rewriting of the source code and program logic to incorporate their specific APIs. As we will discuss in § 4.5, even though the rewriting is possible without considering the extensive labor required, it still remains technical daunting.

2.2 Challenges

Nevertheless, developing an effective approach to detect Rust data leak still faces three core technical challenges.

C1: challenge of sensitivity annotation. Data leak detection often requires sensitivity annotation of metadata directly within source code [26, 31, 44, 47] as a first step, because sensitivity is essentially a semantic property rather than a syntactic one. However, Rust's strict type system imposes rigorous constraints on such source-level annotations and requires them to comply with type safety rules. Furthermore, while introducing new domain specific languages (DSLs) may alleviate the burden of annotations, it may introduce compatibility issues to legacy code.

Solution: To address this challenge, we leverage a distinct Rust feature, Rust macros [6], to annotate data sensitivity within programs. This approach not only complies with Rust's type system but also eliminates the need for additional data structures or modifications to program logic. Furthermore, as macros are expanded into the abstract syntax trees during compilation, we can leverage the standard Rust compiler to translate these annotation metadata to RITA.

C2: Rust's ownership and lifetime. Rust incorporates unique features of ownership [14] and lifetime [5] to establish a memory safety paradigm without the need of garbage collectors, and utilizes complex rules for lifetime checking *after* normal type checking. Consequently, program analysis failing to properly address the complex interleaving of lifetime and taint checking may produce imprecise results or even result in analytical failure.

Solution: To address this challenge, we propose a new intermediate representation RITA (Rust Intermediate for Taint Analysis), and leverage the `rustc` compiler to transform Rust sources with complex language features into RITA to conduct subsequent analysis. Furthermore, we carefully arrange the ordering of lifetime and taint checking so that the latter is triggered only after the former finished, thus avoiding the complex interleaving.

C3: difficulty in sensitivity tracking. As a language advocating functional programming paradigms, Rust programs exhibit extensive function calls through *trait* [5], which allow indirect and virtual calls. These features bring challenges to static analysis as they create intricate control flow patterns, leading to precision degradation even for programs with modest sizes.

Solution: To address this challenge, we design a context-sensitive inter-procedural analysis with a functional approach [36], to track sensitivity flows within the program. We annotate function parameters to distinguish different call sites of the same function, thereby precisely tracking sensitivity.

3 Approach

In this section, we present our approach to conduct the study. We first introduce the overall workflow (§ 3.1), then the design details (§ 3.2 and § 3.3), followed by the implementation (§ 3.4).

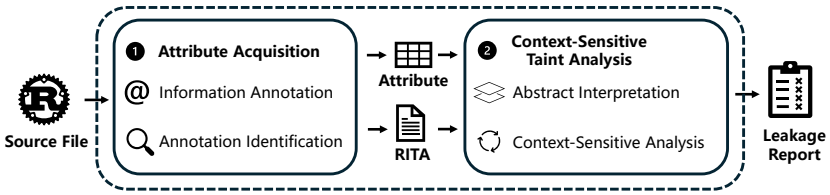

Fig. 2. The workflow of our approach.

3.1 Workflow

Fig. 2 illustrates the systematic workflow of our approach, comprising two key phases: attribute acquisition and context-sensitive taint analysis. First, the attribute acquisition (❶) phase annotates data sensitivity in the Rust source code, which indicates the desired taint status. Then the annotated Rust programs are compiled into the RITA representation along with attributes for subsequent processing. Second, the context-sensitive taint analysis phase (❷) utilizes a context-sensitive inter-procedural static taint analysis on the RITA to track the propagation of sensitive attributed data with an abstract lattice we designed, to detect potential data leaks. The following sections elaborate the details of each phase in sequence.

3.2 Attribute Acquisition

The attribute acquisition phase operates on Rust source code and annotates sensitive information alongside identification to obtain sensitive attributes.

Sensitivity annotation. To detect data leak issues, the analysis often requires, as a first step, obtaining data sensitivity information contained within the program. However, data sensitivity information is essentially a deep semantics property instead of a syntactic one. As a result, data sensitivity cannot be acquired through syntactic analysis, which inevitably necessitates extra semantic annotations. Additionally, as discussed in challenge **C1**, these annotations must comply with Rust's strict type system, making existing annotation approaches proposed for other languages [26,47] inapplicable to Rust. Furthermore, approaches [31,39] requiring invasive code modifications would significantly increase developer burden and compromise detection usability. Consequently, a key requirement for sensitivity annotation for Rust is to keep compatibility with Rust's strict type enforcement while minimizing code modifications to enhance usability.

To fulfill this requirement, we utilize a distinct Rust feature, Rust macro [6], to perform sensitivity annotations. We select Rust macros for two reasons. First, they are essentially meta-programming features that allow us to extend Rust syntax in a compatible and type safe manner. Second, unlike C/C++ preprocessor macros which are processed during preprocessing, Rust macros manipulate the abstract syntax tree (AST), allowing us to directly propagate sensitivity

Constant	c	$\in$	$\mathbb{Z}$	Label	l	$\in$	$\mathbb{Z}$
BinaryOp	$\oplus$	$::=$	$+ \mid - \mid \ldots$	UnaryOp	$\bullet$	$::=$	$! \mid -$
Operand	op	$::=$	$c \mid p$	Place	p	$::=$	$v \mid *p \mid p.n \mid p[v]$

Rvalue	r	$::=$	$op \mid \&p \mid \bullet op \mid op_1 \oplus op_2$
Statement	s	$::=$	$s_1;\ s_2 \mid p = r$
Terminator	t	$::=$	$Call(f, [op_1, op_2, \ldots], (p, b)) \mid Goto(l) \mid$ $Switchint(op, [b_1, b_2, ...])$
Block	b	$::=$	$l : s_1; \ldots; s_n; t$
Function	f	$::=$	$x(y_1, \ldots, y_n)\{b_1 \ldots b_n\}$

Fig. 3. Core syntax of the language model RITA.

information to RITA via AST. Specifically, we design attribute macros including `#[taint::source]` and `#[taint::sink]` to conveniently represent the two-point lattice (see § 3.3) we leverage for taint checking. Moreover, thanks to the inherently extensible nature of Rust macros, practitioners can introduce other attribute macros to represent more complex lattices, reasoning other properties of programs.

Annotation identification. As sensitivity annotations are applied to the source code for usability, while analytical processing occurs at the RITA representation, we instrument the standard Rust compiler `rustc` to expand the attribute macros for subsequent analysis. During the compilation, we design a compilation pass that distills the sensitivity attributes and record them on the corresponding RITA representations.

3.3 Context-sensitive Taint Analysis

We design a context-sensitive taint analysis for detecting data leak issues in Rust, comprising four key parts: a language representation RITA, a lattice as an abstract domain, transfer functions and data flow equations, and a context-sensitive inter-procedural analysis based on a functional approach.

Language representation. We first design a language representation to reduce the grammatical complexity of the Rust language and guide the design of abstract states required for analysis as well as transfer functions. Since the existing MIR remains overly complex, and modeling all Rust features is impractical [33], we introduce a simplified language representation RITA aligned with the core syntax of Rust's MIR, and strategically retains only the components essential for our taint analysis, while categorizing non-essential elements as extensible modules for potential future integration.

Fig. 3 presents the formal representation of RITA, using a simplified context-free grammar. A Rust program comprises functions f, which can be uniformly

represented using Control Flow Graph (CFG). A function f consists of a list of formal arguments y_i, $1 \leq i \leq n$, and a list of basic blocks b. A basic block b contains a unique label l, a list of statements s_i, $0 \leq i \leq n$, and a terminator t. A terminator t can be jumps, function calls or switches, while a statement s comprising sequences, or assignment of rvalues r (i.e., use, borrow and computation) to the given place p.

Abstract domain. We then utilize an abstract domain [18] to characterize the data taint states of data within the program, thereby reducing the complexity of concrete execution states. We construct multiple lattice structures as our analysis domains. Specifically, we employ a binary lattice $\langle\{\bot, \top\}, \sqsubseteq\rangle$ to represent the taint states, where the bottom element ($\bot$) denotes non-tainted state whereas the top element ($\top$) denotes tainted state. To maintain the taint status of each element, we design a mapping lattice that associates each variable within a function to its corresponding state lattice. Finally, we maintain an alias set to perform points-to analysis.

Transfer function. We formalize a set of transfer functions [18] to model the propagation of taint across statements in a function. We use $\sigma(v)$ to denote the taint status of variable v, and constants are inherently assigned a non-tainted status, i.e., $\sigma(c) = \bot$.

We focus on the impact of assignment statements on variable states, since other types of statements do not contribute to taint propagation. For assignments containing only a single right operand $dst ::= src \mid dst ::= \bullet src \mid dst ::= \&src$, we establish that the taint status of the right operand src directly propagates to the left operand dst, expressed as $\sigma(dst) = \sigma(src)$. Specifically, for borrow operations, we additionally add dst to the alias set associated with src. For assignment statements containing two operands $dst ::= op_1 \oplus op_2$, we establish that the taint status of any right operand op_1 or op_2 propagates to the corresponding left operand dst, denoted as $\sigma(dst) = \sigma(op_1) \sqcup \sigma(op_2)$. Finally, any modification to dst necessitates systematic updates to all its aliases.

Data flow equations. We employ data flow equations [29] to characterize the propagation of taint states between basic blocks within a control flow graph. Specifically, we utilize the following data flow equations

$$In[n] = \bigsqcup_{p \in pred[n]} Out[p], \quad Out[n] = In[n] \bigsqcup f(n), \tag{1}$$

to describe the flow propagation, where the input taint state $In[n]$ for a node n is calculated from the states of all its predecessor p, while the output taint state $Out[n]$ is determined by the input state $In[n]$ and its local transfer function $f(n)$.

Context-sensitity and Algorithms. We introduce context-sensitive analysis [36] to improve precision by distinguishing different call sites. Specifically, to speed up the analysis, we adopt the functional approach with summary instead of the call-string approach, and distinguish context information through parameter states at call sites.

Finally, implement these theoretical foundations to design a context-sensitive taint analysis algorithm, as presented in Algorithm 1. This algorithm uses a fix-

Algorithm 1: Context-sensitive analysis algorithm.

Input: Control Flow Graph: G
Output: Abstract State: $State$
Init: $State[v] \leftarrow \bot$, $Record \leftarrow Empty$

```
1  Function FixedPoint(G):
2      WorkList ← all blocks in G in postorder;
3      while WorkList is not empty do
4          Basic block b ← remove(WorkList);
5          State[b] ← new_state ← ⨆_{p∈pred[b]} State[p];
6          foreach statement s ∈ b do
7              State[b] ← State[b] ⊔ Transfer(s);
8          if b.terminator is Call(f, [op1, op2, ...], (p, b)) then
9              arg_state ← {State(op1), State(op2), ...};
10             if f annotated as Source then
11                 State[p] ← ⊤;
12             else if f annotated as Sink and any arg_state is ⊤ then
13                 report a data leak;
14             else
15                 if Record[f(arg_state)] does not exist then
16                     Record[f(arg_state)] ← ⊤;
17                     Record[f(arg_state)] ← FixedPoint(f.CFG);
18                 State[b] ← State[b] ⊔ Transfer(Record[f(arg_state)]);
19         if new_state ≠ State[b] then
20             foreach b' ∈ succ[b] do
21                 WorkList ← WorkList ∪ {b'};
```

point strategy by iterating over all basic blocks within a function via a worklist. During the iteration, the algorithm evaluates the effects of each statement against the abstract state employing transfer functions (line 7). For inter-procedural analysis, we record the abstract states of function parameters during callee function invocations, into a function summary. When encountering previously recorded states, we directly reuse the recorded result in the summary; otherwise, we analyze the function being called function (line 17). To prevent the infinite analysis for recursive invocations, we insert a temporary state before entering the callee function (line 16) .

3.4 Implementation

To validate our approach, we design and implement a prototype system RustGuard. We implement our detection analysis using the Rust language, and utilize the most recent `rustc` compiler (version 1.89.0-nightly) and toolchain. Our design builds upon components from the `rustc` compiler. To access the internal

data structures in `rustc`, we leverage the `rustc-dev` package. We implement our solution as a compiler-integrated callback mechanism through `Analysis` trait in `rustc`, enabling iterative program analysis within the compiler infrastructure.

4 Evaluation

To understand the effectiveness of our approach, we evaluate RustGuard on both micro-benchmarks and real-world Rust programs. Specifically, our evaluation aims to answer the following research questions:
RQ1: Effectiveness. Since our approach aims to detect data leak issues in Rust, is RustGuard effective in achieving this goal?
RQ2: Performance. How much time and memory does RustGuard require to detect issues in Rust programs?
RQ3: Ablation study. As we employ a context-sensitive approach to improve our analysis, how does this approach contribute to the detection of data leak issues?
RQ4: Compare with state-of-the-art. Does RustGuard outperform existing approaches?

All the experiments and measurements are performed on a server with one 12 physical Intel i7 core (20 hyperthread) CPU and 128 GB of RAM. The machine runs 64-bit Ubuntu 24.04 Linux with kernel version 6.8.0. The Rust programs are compiled with `rustc` version 1.89.0-nightly build.

4.1 Datasets

We conduct the evaluation using a set of micro-benchmarks consisting of 18 test cases we created and two real-world open-source Rust projects.
Micro-benchmarks. Evaluating the effectiveness of Rust data leak detection requires a test suite with ground truth, but, to the best of our knowledge, there currently exists no readily available test set for Rust. Since establishing ground truth for large and complex real-world programs is impractical, we take the first step to manually construct a micro-benchmark containing 18 test cases, including 12 positive samples with data leak issues and 6 secure negative samples without such issue, as presented by the first 12 rows and last 6 rows in Table 1, respectively. Moreover, as the second column of Table 1 shows, our test suite covers Rust's unique syntax such as borrowing, traits, generics, and closures, as well as various non-linear control flows, because these unique syntax elements and complex data flows can potentially impact the efficiency and accuracy of analysis. Currently, we are maintaining and augmenting it by including more benchmarks while covering more Rust features.
Real-world projects. We select two open-source Rust projects, Spotify TUI [9] and Slack Morphism for Rust [1], as real-world benchmarks. We select Spotify TUI because it is a popular open source Rust project on GitHub with over 18.1K stars and has been used by prior work [31] as a case study, which allows us to compare our approach with state-of-the-art. We select the Slack Morphism for Rust because it contains Rust data leak CVEs [3] that are relevant to this study.

Table 1. Experimental results on micro-benchmarks.

Test Case	Kind	RustGuard			RustGuard$^{-context}$			rustc	
		Loc	Time (ms)	Memory (MB)	Loc	Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
1*	simple	10:5	125	1.697	10:5	123	1.696	106	1.695
2*	arithmetic	9:5	132	1.697	9:5	143	1.697	126	1.697
3*	cond-branch	17:5	150	1.697	17:5	116	1.696	109	1.697
4*	inter-procedural	8:5	162	1.696	8:5	155	1.696	155	1.696
5*	recursion	8:5	150	1.696	8:5	147	1.697	149	1.697
6*	reference	10:5	124	1.697	9:5,10:5	121	1.696	123	1.697
7*	argument	9:5	120	1.695	9:5	128	1.695	138	1.697
8*	struct field	18:5	139	1.696	17:5,18:5	131	1.696	106	1.695
9*	impl	18:5	150	1.696	18:5	125	1.696	86	1.697
10*	generic	19:5	149	1.698	18:5,19:5	152	1.697	120	1.698
11*	trait	30:14	151	1.697	30:14	147	1.697	152	1.698
12*	closure	–	153	1.697	–	139	1.696	137	1.696
13	const	–	136	1.697	–	137	1.697	126	1.698
14	const uop	–	140	1.696	–	132	1.696	138	1.696
15	const rvalue	–	129	1.696	–	134	1.696	101	1.697
16	recursion	–	125	1.697	–	141	1.676	121	1.696
17	switchint	–	129	1.696	–	125	1.696	113	1.696
18	struct field	22:5	133	1.695	22:5	131	1.695	138	1.695

*: Cases with data leak issue
Loc $(r : c)$: issue locations reported, where r and c denotes the corresponding row and column, respectively

4.2 RQ1: Effectiveness

To answer RQ1 by investigating the effectiveness of our approach, we first evaluate RustGuard on the micro-benchmarks. We use *precision* and *recall* as the metrics to measure the effectiveness, and the definition of these two metrics is provided by equations $precision = tp/(tp + fp)$ and $recall = tp/(tp + fn)$, where tp, fp, and fn denote true positives, false positives, and false negatives, respectively.

Experimental results are summarized in the column **RustGuard** in Table 1. Among the 12 positive test cases, RustGuard successfully detected 11 cases but missed one case, achieving a precision of 91.67%. Moreover, for the 6 issue-free negative test cases, RustGuard reported a false data leak issue, yielding a recall of 91.67%. To summarize, these results demonstrate that RustGuard can effectively detect data leak issues in Rust programs with diverse syntactic features.

To further investigate the root causes of why RustGuard incurs both false negatives and false positives, we conduct a comprehensive manual audit of the corresponding source code. This inspection revealed that the false negatives are caused by the Rust closures, a feature that our implementation only partially

supported. Additionally, the false positives are caused by composite structures containing both sensitive and non-sensitive members, for which RustGuard conservatively treats the entire struct as sensitive.

4.3 RQ2: Performance

To answer RQ2 by investigating the performance of RustGuard, we measure the time and memory consumption during analysis. To this end, we execute test cases using both the unmodified `rustc` and RustGuard that augmented with our analysis, and measure compilation time and peak memory usage with the widely used `time` and `valgrind` utilities, respectively. To eliminate potential bias, we repeat the above process on each test case 5 rounds to calculate the average analysis time and memory usage, following prior work on Rust data flow analysis [20].

The columns **RustGuard** and `rustc` in Table 1 present the experimental results, respectively. Compared to the original `rustc` compiler, RustGuard incurs a runtime overhead up to 14.07% and negligible memory consumption, which is in line with prior work [20]. Furthermore, as RustGuard performs static checking during the compilation phase, it incurs no runtime overhead.

4.4 RQ3: Ablation Study

To justify the contribution of context-sensitive analysis in improving the precision of analysis, we perform an ablation study. Specifically, we redesign a prototype system RustGuard$^{-context}$ that removes the context-sensitive component from the analysis while keeping all other components identical. To this end, the RustGuard$^{-context}$ prototype performs a context-free analysis. We then applied the RustGuard$^{-context}$ to the micro-benchmarks and compare the results generated by RustGuard.

The column **RustGuard**$^{-context}$ in Table 1 presents the experimental results of RustGuard$^{-context}$. RustGuard$^{-context}$ detected 11 out of 12 positives but missing the same case as RustGuard. However, RustGuard$^{-context}$ falsely reported data leaks for test cases 6, 8, and 10. For example, RustGuard$^{-context}$ reports, for the 6th test case, that there are two data leaks at both 9:5 and 10:5 which the former one is a false positive. Moreover, RustGuard$^{-context}$ reported one issue that coincided with RustGuard for the 6 negative samples. Overall, RustGuard$^{-context}$ achieved a recall of 91.67% but a precision only of 73.33%. These comparative results between RustGuard and RustGuard$^{-context}$ demonstrate that context-sensitivity component in RustGuard improves the precision of data leak detection.

4.5 RQ4: Compare with Existing Studies

We compare RustGuard with the existing research Cocoon [31], a recent work on Rust data integrity that is most relevant to our work, on two real-world Rust

Fig. 4. Comparison between RustGuard (left) and Cocoon (right).

projects. However, since Cocoon emphasizes information flow control over program data rather than directly detecting data leak issues, our comparison focus on the program's intrusiveness and practical usability to avoid any potential bias. Furthermore, for fairness, we utilize the test cases from Cocoon [31] and public CVEs for comparison, but do not use the test cases in this work.

First, we adapted the test case Spotify TUI from Cocoon [31], as shown in Fig. 4(a). Spotify client is a terminal written in Rust, and contains a data leak issue at line 196 where the function `output` may leak a user password as a 32-digit hexadecimal string, which is used for authenticating with the Spotify API server. To detect this data leak issue with RustGuard, we require only two lines of code modifications, that is, explicitly marking both the source point of the password string and the leak point by attribute macros (i.e., `#[taint::sink]` at line 201). With these annotations, RustGuard fully automated detects this issue and log the precise location with root causes in the terminal (Fig. 4(b)) for subsequent diagnosis. For comparison, we run this test case with Cocoon. However, as a first step, we have to extensively rewrite this test case to incorporate Cocoon's special syntax and APIs (e.g., `secret_block!` at line 185). The resulting program after rewriting is given in Fig. 4(c), with contains considerable more lines of distinct code. Nevertheless, the metric of code size does not necessarily reflect the real efforts for such code rewriting, and we speculate that for large Rust projects,

such invasive modifications of legacy code might incur more labor and thus is less cost-effective.

Next, we evaluate both RustGuard and Cocoon on real-world CVE-2022-31162 [3] in Slack Morphism for Rust (details in Fig. 1), and reach the same conclusion. These comparative results demonstrate that RustGuard is more cost-effective by exhibiting lower intrusiveness and better end-user usability, compared to existing approaches.

5 Limitations and Future Work

In this section, we discuss the limitations of this work and our plans for future work. First, our current prototype implementation of RustGuard does not fully support Rust's closure syntax, which might produce false positives due to conservatism. Supporting these advanced Rust syntax features remains a challenge in the Rust security community [33], and we will continue to extend our prototype to support these features following recent studies [27, 48].

Second, our current approach only uses two-point lattices to model data sensitivity. While this approach is sufficient to model and reason taint data, adopting more precise yet computationally intensive models including octagon [35] and polyhedra [19] will make our approach apply to other security analysis beyond taint analysis. However, deploying such sophisticated models would concurrently increase analysis time and memory overhead, and a critical trade-off requires further investigation in our future work.

Finally, as one of our intended design goals, RustGuard does not address implicit data flows. Meanwhile, existing studies [30] demonstrate that tracking implicit data flows remains complex and inefficient. As studies on Rust's implicit data flows, to the best of our knowledge, are still lacking, we believe the first step we may take towards investigating manifestation patterns of implicit data flows in Rust, based on which corresponding mitigations can be proposed.

6 Related Work

In recent years, there have been a significant amount of studies on Rust security and information flow security most relevant to this work.

Rust Security. Existing research on Rust safety has been conducted from multiple perspectives. In empirical studies, Evans et al. [25] investigated the usage of unsafe mechanisms in real-world Rust applications, while Xu et al. [45] surveyed 186 memory-safety vulnerabilities related to Rust to explore its memory safety issues. In the field of vulnerability detection, existing technologies include MirChecker [33], a static bug detection framework designed for Rust; RustSan [17], a Rust memory purification technique; Rudra [16] to detect Rust memory-safety vulnerabilities; XRust [34] for preventing cross-region memory corruption by unsafe memory isolation, and RULF [27] and FRIES [48] for fuzzing Rust libraries. Finally, research on formal verification of Rust programs includes KRust [43], RustBelt [28], among others.

However, the previous studies overlooked Rust's data leak issues that are addressed by this study.

Information Flow Security. Information flow security is a critical field of program security. To detect data leaks in Android applications, Yang et al. [47] designed LeakMiner based on static taint analysis, while Newsome et al. [37] employed dynamic taint analysis to automate the detection, analysis, and signature generation of vulnerabilities in commercial software. Additionally, techniques such as TaintDroid [24], FlowDroid [15] and DroidTrack [42] have been widely adopted in this domain. In addition to being an important data leak detection technique, data flow control is also a critical approach for ensuring data flow security. Pullicino et al. [39] designed Jif, a Java extension language, by adding security labels to Java's type system to enable language-based security features. Lamba et al. [31], on the other hand, developed a security library that implements type-based static data flow control techniques for Rust.

However, these techniques either fail to account for Rust's unique features like ownership and lifetime, making them inapplicable to Rust programs, or require extensive rewriting of existing program logic, thereby incurring considerable labor or even incompatibility issues.

7 Conclusion

In this work, we present a novel information flow checking approach for Rust by leveraging static taint analysis to detect data leak issues. We first propose to utilize Rust macros to annotate data sensitivity in Rust programs. We then establish a language model termed RITA to formalize Rust's core syntax. To characterize and trace the tainted data states, we design an abstract domain, transfer functions and finally a context-sensitive inter-procedural data flow analysis algorithm. We implement a software prototype called RustGuard and conduct experiments to evaluate it on micro-benchmarks as well as real-world CVEs. The experimental results demonstrate that RustGuard reaches 91.67% precision and recall with a compile time overhead of 14.07%, negligible memory consumption and zero runtime overhead. Furthermore, our approach can detect existing CVEs and outperforms existing state-of-the-art approach *Cocoon*. Overall, our work represents a new step towards security enhancement of Rust, making Rust's promise of being a safe language a reality.

References

1. Abdolence/slack-morphism-rust: A modern async client library for rust, supports slack web / events api/socket mode and block kit., https://github.com/abdolence/slack-morphism-rust
2. Check instructions sysvar · wormhole-foundation/wormhole@e8b9181, https://github.com/wormhole-foundation/wormhole/commit/e8b91810a9bb35c3c139f86b4d0795432d647305
3. Cve-2022-31162 - osv, https://osv.dev/vulnerability/CVE-2022-31162

4. Diem/diem: Diem's mission is to build a trusted and innovative financial network that empowers people and businesses around the world., https://github.com/diem/diem
5. Generic types, traits, and lifetimes - the rust programming language, https://doc.rust-lang.org/book/ch10-00-generics.html
6. Macros - the rust programming language, https://doc.rust-lang.org/book/ch20-05-macros.html
7. Openethereum/parity-ethereum: The fast, light, and robust client for ethereum-like networks., https://github.com/openethereum/parity-ethereum
8. Redox - your next(gen) os - redox - your next(gen) os, https://www.redox-os.org/
9. Rigellute/spotify-tui: Spotify for the terminal written in rust, https://github.com/Rigellute/spotify-tui
10. Rust for linux, https://github.com/Rust-for-Linux
11. The rust programming language - the rust programming language, https://doc.rust-lang.org/stable/book/
12. Tiobe index, https://www.tiobe.com/tiobe-index/
13. Update solana to 1.9.4 · wormhole-foundation/wormhole@7edbbd3, https://github.com/wormhole-foundation/wormhole/commit/7edbbd3677ee6ca681be8722a607bc576a3912c8
14. What is ownership? - the rust programming language, https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
15. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Notices **49**(6), 259–269 (Jun 2014). https://doi.org/10.1145/2666356.2594299
16. Bae, Y., Kim, Y., Askar, A., Lim, J., Kim, T.: Rudra: Finding memory safety bugs in rust at the ecosystem scale. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 84–99. ACM, Virtual Event Germany (Oct 2021). https://doi.org/10.1145/3477132.3483570
17. Cho, K., Kim, J., Duy, K.D., Lim, H., Lee, H.: Rustsan: Retrofitting addresssanitizer for efficient sanitization of rust
18. Cousot, P., Cousot, R.: 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 - POPL '77. pp. 238–252. ACM Press, Los Angeles, California (1977). https://doi.org/10.1145/512950.512973
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '78. pp. 84–96. ACM Press, Tucson, Arizona (1978). https://doi.org/10.1145/512760.512770
20. Cui, M., Chen, C., Xu, H., Zhou, Y.: Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. ACM Transactions on Software Engineering and Methodology **32**(4), 1–21 (Oct 2023). https://doi.org/10.1145/3542948
21. Cui, S., Zhao, G., Gao, Y., Tavu, T., Huang, J.: Vrust: Automated vulnerability detection for solana smart contracts. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 639–652. ACM, Los Angeles CA USA (Nov 2022). https://doi.org/10.1145/3548606.3560552

22. Developers, T.S.P.: Servo aims to empower developers with a lightweight, high-performance alternative for embedding web technologies in applications., https://servo.org/
23. Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., Halderman, J.A.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference. pp. 475–488. ACM, Vancouver BC Canada (Nov 2014). https://doi.org/10.1145/2663716.2663755
24. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems **32**(2), 1–29 (Jun 2014). https://doi.org/10.1145/2619091
25. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 246–257. ACM, Seoul South Korea (Jun 2020). https://doi.org/10.1145/3377811.3380413
26. Grech, N., Smaragdakis, Y.: P/taint: Unified points-to and taint analysis. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 1–28 (Oct 2017). https://doi.org/10.1145/3133926
27. Jiang, J., Xu, H., Zhou, Y.: Rulf: Rust library fuzzing via api dependency graph traversal. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 581–592. IEEE, Melbourne, Australia (Nov 2021). https://doi.org/10.1109/ASE51524.2021.9678813
28. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–34 (Jan 2018). https://doi.org/10.1145/3158154
29. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**(3), 305–317 (1977). https://doi.org/10.1007/BF00290339
30. King, D., Hicks, B., Hicks, M., Jaeger, T.: Implicit flows: Can't live with 'em, can't live without 'em. In: Sekar, R., Pujari, A.K. (eds.) Information Systems Security, vol. 5352, pp. 56–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7_4
31. Lamba, A., Taylor, M., Beardsley, V., Bambeck, J., Bond, M.D., Lin, Z.: Cocoon: Static information flow control in rust. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 166–193 (Apr 2024). https://doi.org/10.1145/3649817
32. Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P., Levis, P.: Multiprogramming a 64kb computer safely and efficiently. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 234–251. ACM, Shanghai China (Oct 2017). https://doi.org/10.1145/3132747.3132786
33. Li, Z., Wang, J., Sun, M., Lui, J.C.: Mirchecker: Detecting bugs in rust programs via static analysis. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 2183–2196. ACM, Virtual Event Republic of Korea (Nov 2021). https://doi.org/10.1145/3460120.3484541
34. Liu, P., Zhao, G., Huang, J.: Securing unsafe rust programs with xrust. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 234–245. ACM, Seoul South Korea (Jun 2020). https://doi.org/10.1145/3377811.3380325
35. Mine, A.: The octagon abstract domain. In: Proceedings Eighth Working Conference on Reverse Engineering. pp. 310–319. IEEE Comput. Soc, Stuttgart, Germany (2001). https://doi.org/10.1109/WCRE.2001.957836

36. Muchnick, S.S., Jones, N.D.: Program Flow Analysis: Theory and Applications. Prentice-Hall Software Series, Prentice-Hall, Englewood Cliffs, N.J (1981)
37. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software
38. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, Massachusetts (2002)
39. Pullicino, K.: Jif: Language-based information-flow security in java (Dec 2014). https://doi.org/10.48550/arXiv.1412.8639
40. Qin, B., Chen, Y., Liu, H., Zhang, H., Wen, Q., Song, L., Zhang, Y.: Understanding and detecting real-world safety issues in rust. IEEE Transactions on Software Engineering **50**(6), 1306–1324 (Jun 2024). https://doi.org/10.1109/TSE.2024.3380393
41. Saha, S., Ghentiyala, S., Lu, S., Bang, L., Bultan, T.: Obtaining information leakage bounds via approximate model counting. Proceedings of the ACM on Programming Languages **7**(PLDI), 1488–1509 (Jun 2023). https://doi.org/10.1145/3591281
42. Sakamoto, S., Okuda, K., Nakatsuka, R., Yamauchi, T.: Droidtrack: Tracking information diffusion and preventing information leakage on android. In: Park, J.J., Ng, J.K.Y., Jeong, H.Y., Waluyo, B. (eds.) Multimedia and Ubiquitous Engineering, vol. 240, pp. 243–251. Springer Netherlands, Dordrecht (2013). https://doi.org/10.1007/978-94-007-6738-6_31
43. Wang, F., Song, F., Zhang, M., Zhu, X., Zhang, J.: Krust: A formal executable semantics of rust. In: 2018 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 44–51. IEEE, Guangzhou, China (Aug 2018). https://doi.org/10.1109/TASE.2018.00014
44. Wei, S., Ryder, B.G.: Practical blended taint analysis for javascript. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 336–346. ACM, Lugano Switzerland (Jul 2013). https://doi.org/10.1145/2483760.2483788
45. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.: Memory-safety challenge considered solved? an in-depth study with all rust cves. ACM Transactions on Software Engineering and Methodology **31**(1), 1–25 (Jan 2022). https://doi.org/10.1145/3466642
46. Yakovenko, A.: Solana: A new architecture for a high performance blockchain
47. Yang, Z., Yang, M.: Leakminer: Detect information leakage on android with static taint analysis. In: 2012 Third World Congress on Software Engineering. pp. 101–104. IEEE, Wuhan, China (Nov 2012). https://doi.org/10.1109/WCSE.2012.26
48. Yin, X., Feng, Y., Shi, Q., Liu, Z., Liu, H., Xu, B.: Fries: Fuzzing rust library interactions via efficient ecosystem-guided target generation. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1137–1148. ACM, Vienna Austria (Sep 2024). https://doi.org/10.1145/3650212.3680348