RUSPATCH: Towards Timely and Effectively Patching Rust Applications

Yufei Wu, and Baojian Hua*
School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
wuyf21@mail.ustc.edu.cn, bjhua@ustc.edu.cn
* Corresponding author.

Abstract—Despite the fact that Rust is designed to be a secure programming language for system programming, it is still vulnerable and exploitable due to its inclusion of an `unsafe` sub-language. However, existing studies on Rust security only focus on static detection or rectification of vulnerability, but ignore the problem of timely and effective rectifications of vulnerabilities dynamically.

In this paper, to fill this gap, we present RUSPATCH, the *first* infrastructure to timely and effectively patch vulnerable Rust applications. RUSPATCH consists of two main phases: static partitioning and dynamic patching. In the static partitioning phase, RUSPATCH divides the candidate program into target code and patch candidates via a customized compiler. During the patching phase, RUSPATCH dynamically validates and applies the security patch once a vulnerability is detected. To realize the whole process, we tackled three technical challenges of language discrepancy, efficiency issues, and security threats. We have designed and implemented a software prototype for RUSPATCH, and have conducted extensive experiments to evaluate its effectiveness, performance, overhead, and usefulness. Experimental results demonstrated that RUSPATCH is effective in patching off-the-shelf Rust applications including real-world Rust CVEs, and the extra overhead RUSPATCH introduced is less than 3.28% and thus insignificant. Furthermore, RUSPATCH is easy to incorporate into existing Rust applications without any manual interventions.

Keywords–Rust, Vulnerability, Dynamic patching

1. INTRODUCTION

Programming languages directly impact the reliability, safety, and correctness of software, and their features impact the prevalence of bugs in actual systems. Rust [1] is an emerging programming language designed for constructing safe system software. To achieve its design goals of both *security* and *efficiency*, Rust introduced a group of novel language features (*e.g.*, ownership [2], borrow [3], reference [4], and explicit lifetime [5]), to guarantee security without sacrificing execution efficiency. Due to its technical advantages, Rust has gained popularity in the past several years, and has been used successfully in a wide spectrum of domains such as operating system kernels [6], cloud services[7], network protocol stacks [8], runtimes [9], databases [10], and blockchains [11].

While Rust makes an important step towards secure system programming, it is, unfortunately, still vulnerable and exploitable, due to its inclusion of an `unsafe` sub-language [12] to support arbitrary low-level operations and to offer more programming flexibility. Specifically, the `unsafe` Rust programs might be vulnerable as they not only disable compiler static checking but also bypass necessary runtime checking, thus may defeat Rust's security guarantees leading to security issues. For example, in Rust CVE-2020-35879 [13], the unsafe `raw_slice_mut` method caused a data race, due to its incorrect use of lifetime in unsafe code. As another example, in Rust CVE-2018-1000810 [14], the buffer access `buf[i]` triggered out-of-bounds (OOBs) memory issues, as `unsafe` Rust does not check accesses against buffer range, leading to memory corruptions [15] [16]. Therefore, given the increasingly important roles of Rust in system programming, a comprehensive study of Rust security is essential.

Recognizing this need, researchers have recently conducted a significant amount of research efforts (*e.g.*, empirical security study [17] [18] [19], vulnerability detection [20] [21] [22] [23], security enhancement [24] [25], and formal verification [26] [27]), to address Rust security issues. While these research efforts made considerable progress in securing Rust applications, they, unfortunately, have severe limitations: they only focus on static detection or rectification of vulnerabilities, but ignore the problem of *dynamic software updating* (DSU), in which a *running* vulnerable Rust application is rectified with a security patch without stopping or restarting the target application. DSU is indispensable and important in today's 7/24 world, such as online services (*e.g.*, Firecracker [28], Amazon AWS newly deployed virtualization service in Rust) cannot be stopped or restarted for the unexpected delay that an offline patching strategy might incur. For example, according to Gartner [29], the average cost of IT downtime is $5,600 per minute and the average enterprise downtime can reach $300,000 per hour. Even in non-online service scenarios, DSU is also important and valuable, given the normal urgency to rectify vulnerabilities [30].

Challenges. Unfortunately, while DSU has been extensively studied and has shown promising potential [30] [31] [32], to the best of our knowledge, no such systems exist for Rust. Yet developing an effective DSU infrastructure for Rust faces three key challenges: **C1:** *language discrepancy* caused by Rust's

unique language features absent from other languages; **C2:** *efficiency issues* brought by existing studies to the rectified systems, which make these studies infeasible to Rust designed with the goal of runtime efficiency; and **C3:** *security threats* of the patches in binary forms in existing studies, which might defeat Rust's security guarantees.

Our work. In this paper, to fill the gap, our goal is to investigate techniques and infrastructures to solve the Rust dynamic software updating problems. To achieve this goal, we propose the *first* framework dubbed RUSPATCH, to patch vulnerable Rust applications timely and effectively. Specifically, the workflow of RUSPATCH consists of two phases: a compile-time partitioning phase, and a dynamic patching phase. In the first phase, RUSPATCH splits the candidate Rust program into two components: target code and patch candidates, both of which are then compiled to binaries and deployed on the cloud. In the second phase, once a vulnerability is encountered or detected, RUSPATCH validates the target patch for vulnerability rectification, and then compiles the patch to loadable modules, which are sent to the cloud to substitute the existing vulnerable modules.

To realize the whole process, we tackle the three aforementioned technical challenges. **C1:** *language discrepancy*: to address **C1**, we designed and implemented a delegatecall proxy pattern generator by modifying Rust's official `rustc` compiler extensively to add customized compiler passes. **C2:** *efficiency issues*: to address **C2**, we designed and implemented an automatic and transparent multithreading code injector to generate and inject dynamic updating thread, by utilizing Rust's concurrency mechanism. **C3:** *security threats*: to address **C3**, we designed a dynamic code validator to automatically validate the patches before compiling and deploying them, by utilizing both the official Rust borrow checker [33] and state-of-the-art checking tools such as Clippy [34], and Miri [35].

We have implemented a software prototype for RUSPATCH, and have conducted extensive experiments on CloudLab [36] to evaluate it. To conduct the evaluation, we also created two datasets: a microbenchmark **RusBench** consisting of vulnerable Rust programs adapted from real-world Rust CVEs, and a macrobenchmark containing large and real-world Rust applications.

With this prototype and datasets, we evaluate RUSPATCH in terms of effectiveness, performance, overhead, and usefulness. We first applied RUSPATCH to RusBench, and experimental results demonstrated that RUSPATCH is effective in patching all Rust vulnerabilities effectively, showing a 100% success rate. Second, to evaluate the performance of RUSPATCH, we conducted experiments on RusBench, which showed that RUSPATCH is efficient in processing Rust programs in line with `rustc`. Third, to evaluate the overhead RUSPATCH introduced, we measure the execution time of the target Rust application before and after using RUSPATCH, and experimental results demonstrated that the runtime overhead RUSPATCH introduced is less than 3.28%. Finally, a developer study shows RUSPATCH is easy to be incorporated into real-world Rust applications, without any developer intervention or manual code rewriting in applying patches.

Contributions. To the best of our knowledge, this work represents the *first* step towards addressing Rust dynamic software updating problems. To summarize, this work makes the following contributions:

- **Infrastructure design.** We proposed the first infrastructure dubbed RUSPATCH, to patch Rust applications dynamically.
- **Prototype implementation.** We implemented a prototype for RUSPATCH, to deploy patches for Rust applications timely and effectively.
- **Extensive evaluations.** We conducted extensive experiments to evaluate RUSPATCH in terms of effectiveness, performance, overhead, and usefulness, on real-world Rust CVEs and practical Rust applications.

Outline. The rest of this paper is organized as follows. Section 2 presents the background for this work. Section 3 introduces the motivation for this work as well as the threat model. Section 4 presents the overall design and challenges. Sections 5 and 6 present the design and implementation of RUSPATCH, respectively. Section 7 presents the experiments to evaluate RUSPATCH. Section 8 discusses the limitations of this work as well as directions for future work. Section 9 discusses the related work, and Section 10 concludes.

2. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work, by introducing the Rust programming language (§ 2-A), the `unsafe` Rust (§ 2-B), and dynamic software updating (§ 2-C).

2.1. Rust

Brief history. Rust [1] is an emerging and rapidly growing programming language. Initially designed in 2006, Rust was first publicly released in 2010 [37]. Designed to be a safe system programming language, Rust has grown into a production quality language after its first stable version 1.0 in 2015, and has shown promising potential in building secure system applications.

Advanced features. Rust emphasizes both *safety* and *efficiency*. On the one hand, Rust guarantees safety by introducing a group of novel language features such as ownership [2], borrow [3], reference [4], and explicit lifetime [5], which are checked at compile-time. On the other hand, Rust achieves high efficiency by embracing a zero-abstraction design philosophy and by inheriting most language designs from C. Specifically, Rust eliminates the potential runtime penalty of automatic memory management (*e.g.*, garbage collectors [38]), by incorporating an ownership-based memory management mechanism ensuring each value has a unique owner [39].

Wide applications. Due to its safety and efficiency advantages, Rust is gaining more popularity with wide applications in recent years. Rust was not only rated as the "most popular programming language" on Stack Overflow in 2022 [40], but also gaining more adoptions in the industry (*e.g.*, Microsoft [41], Google [42], and Linux [43]), to build secure applications

in a large spectrum of domains. Specifically, Rust is increasingly important in building online services (*e.g.*, Mononoke [44], Firecracker [28], Discord [45], and Azure IoT Edge [46]). In the future, a desire to secure cloud infrastructures without sacrificing efficiency will make Rust a more promising language.

2.2. Unsafe Rust

Necessity. Unsafe Rust [12] is a sub-language of Rust and serves as a security loophole to bypass Rust's static and dynamic security checking. The unsafe Rust is indispensable for two key reasons: first, it is used to bypass Rust's strict static analysis, which is often overly conservative (*e.g.*, binding an IP address to a socket is *unsafe* in Rust, as it is generally impossible to determine *statically* whether the given IP address is valid). Second, unsafe Rust is extensively used in low-level system programming (*e.g.*, converting an integer to a driver address), in which security is sacrificed to achieve programming feasibility.

Scenarios. Unsafe Rust has five usage scenarios [12]: 1) *raw pointer dereference*, where raw pointers might point to invalid memory, breaking Rust borrowing checking rules; 2) *unsafe functions or methods*, when they contain unsafe operations; 3) *mutable static variables*, which may introduce data races and thus are `unsafe`; 4) *unsafe traits*, when at least one of its methods is `unsafe`; and 5) *unions*, as Rust cannot guarantee the correct types of data in a specific union instance.

Ubiquity. The `unsafe` code is ubiquitous in the Rust ecosystem. For example, in Rust's official package repository `crates.io`, 23.6% of the crates contain `unsafe` code [19]. As another example, 54% of Servo's code [47] (a flagship Rust project of Web browser engine from Mozilla) is `unsafe` [18]. Despite the fact that `unsafe` Rust is ubiquitously used, it breaks the security guarantees of Rust and might lead to security vulnerabilities [48].

2.3. Dynamic Software Updating

Concept. Dynamic software updating (DSU) technology dynamically updates the functionality of a running application, without stopping or restarting the application. In today's 7/24 world, DSU is critical for non-stoppable and high-available systems.

Key techniques. Existing DSU techniques can be classified into two categories: source- and binary-level, depending on whether or not the source code is required. On the one hand, a source-level DSU [49] [50] first converts the target source program into updatable forms, by inserting indirection jumps with the aid of a specialized compiler. At runtime, these jumps are updated to point to the newest version of the code. On the other hand, a binary-level DSU [51] [52] inserts the aforementioned indirection jumps directly into the target binaries by binary rewritings [53], which is preferable when source code is absent.

Wide applications. Due to its advantages of both high available guarantees and timely rectifications, DSU has been widely used in recent years (*e.g.*, Linux kernel [54] [32], Android kernel [55], and IoT devices [56]). Specifically, DSU is being deployed on clouds. For example, Orthus [57] is a novel DSU infrastructure to update KVM/QEMU instances.

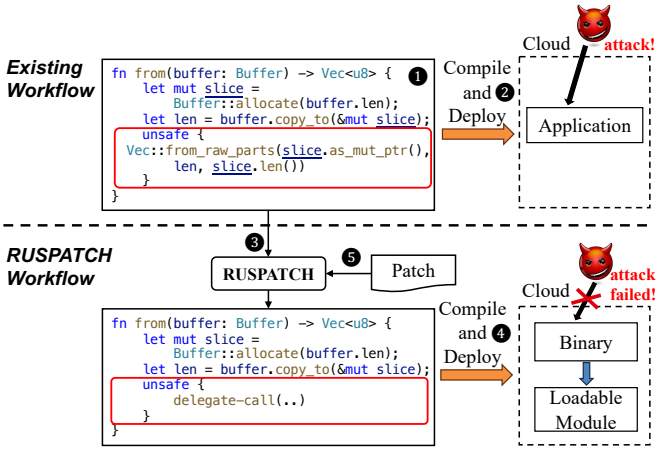

Figure 1: A motivating example demonstrating the technical overview of RUSPATCH, with a real-world CVE-2019-16140.

3. MOTIVATION AND THREAT MODEL

In this section, we first present the motivation for this study via a concrete example (§ 3-A), then the threat model (§ 3-B).

3.1. Motivation

High availability is one of the most important goals of cloud service. However, while cloud service providers make tremendous efforts to avoid downtime, security updates on cloud are frequent which might introduce unexpected downtime, reducing the availability of cloud service. Worse yet, the unexpected downtime might lead to a significant loss [29].

To put the discussion in perspective, in Fig. 1, we present the existing process of how vulnerabilities might be rectified in a cloud scenario and also challenges during this process, with a running example on CVE-2019-16140. We then show how a DSU system, like RUSPATCH in this work, addresses these challenges.

In the original process, the source code (❶) was compiled into binaries and then deployed to the cloud (❷). The Rust source code might be vulnerable due to its inclusion of the `unsafe` sub-language. Indeed, the source code demonstrated a use-after-free (UAF) vulnerability we adapted from a previously reported CVE-2019-16140 [58]. As a result, a malicious adversary can inject arbitrary code into the cloud for execution by exploiting the buffer overflows (*i.e.*, `slice` buffer in this example). To fix such vulnerabilities, the *source* program is first modified by applying a security patch, then compiled and deployed on the cloud to substitute the old binaries, which might lead to unexpected downtime.

To address the downtime challenge, RUSPATCH first compiles the (vulnerable) source code into partitioned sources, via a customized compiler we built (❸). The partitioned sources are then compiled and deployed to the cloud as two components: the binary and loadable module (❹). Once a vulnerability is

detected, RUSPATCH takes both the source code and the security patch, then validates the patch for security and functional correctness, before compiling them to binaries and deploying them on the cloud to substitute the outdated loadable modules (❺).

3.2. Threat Model

This work focuses on the problem of dynamical patching Rust applications timely and effectively. Therefore, we make the following assumptions in the threat model for this work.

We assume that the cloud execution environment, running the Rust applications, has standard protections. For example, the underlying hardware or operating systems provide standard protections such as Data Execution Prevention (DEP) [59], Stack Canaries [15], and Address Space Layout Randomization (ASLR)[60]. Furthermore, the Rust compilers (both the original and the one we customized) have not been compromised by malicious adversaries so the binaries generated from the compilers are trustworthy. It should be noted that although operating systems and compiler security studies are very important, they are independent of and thus orthogonal to the study in this work. Furthermore, those research fields can also benefit from the research progress in this work.

We assume that the safe sub-language of Rust (*i.e.*, without the `unsafe` keyword) is safe and will not pose a security threat to the application being investigated. For example, safe Rust code does not trigger out-of-bounds buffer access, as every buffer access is checked against the buffer length. Existing Rust security studies demonstrated that *all* reported memory bugs in Rust are related to `unsafe` code. Thus, such an assumption is reasonable in reality.

We assume that the `unsafe` sub-language of Rust is vulnerable and susceptible to attacks. For example, as Fig. 1 shows, an attacker may exploit a vulnerability in `unsafe` code to trigger buffer overflows due to the lack of buffer range checking.

We assume that the network between the host and cloud is protected [61] [62] and trustworthy so that the security binary patches as network traffic will not be sniffed or spoofed [63]. Although network security is an important research field, it is independent of and thus orthogonal to the study in this work.

4. OVERALL DESIGN AND CHALLENGES

In this section, we present an overview and design goals of RUSPATCH (§ 4-A), its architecture (§ 4-B), and existing challenges and our solutions (§ 4-C).

4.1. Overview and Design Goals

We have two goals guiding the overall design of RUSPATCH architecture: 1) full program or vulnerability support; and 2) fully automated. First, the architecture of RUSPATCH should support all language features and different kinds of vulnerabilities. This goal is important for RUSPATCH to be applied to any Rust programs, as well as existing or potential vulnerabilities. Otherwise, the usefulness of RUSPATCH is significantly reduced, if it can only be applied to a limited set of Rust programs or specific vulnerabilities. Second, RUSPATCH should be fully automated in converting Rust programs, validating and applying patches. Human interventions are only required to supplement the automated process (*e.g.*, for failure root cause analysis).

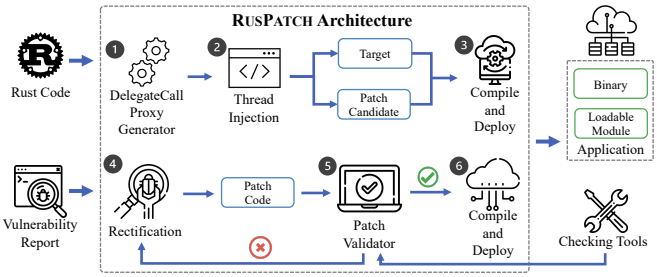

Figure 2: RUSPATCH architecture.

4.2. The Architecture

Guiding by these goals, we present, in Fig. 2, the overall architecture of RUSPATCH, consisting of four key modules. First, the delegatecall proxy pattern code generator (❶) takes as input the original (and potentially vulnerable) Rust source code, and outputs a modified code by rewriting the original code with delegatecall proxy patterns. Second, the thread injection module (❷) takes as input the delegatecall proxy injected code, and outputs both target and patch candidates by generating and injecting updating threads automatically. Third, RUSPATCH compiled (❸) both the target and patch candidates into binaries and loadable modules, respectively, and then deployed them to the cloud as an online service. Fourth, the vulnerability rectification module (❹) takes as input the original Rust code as well as security patches, then automatically rectifies vulnerabilities, and outputs patched source code. Finally, the patch validator (❺) validates the security and functional correctness of the application after the vulnerability has been rectified, then compiles the patches into loadable modules and deploys (❻) them on the clouds to substitute the outdated loadable modules.

In section 5, we will discuss the design of each module in detail, respectively.

4.3. Challenges and Our Solutions

Developing an effective DSU infrastructure for Rust needs to address the following three challenges (§ 1).

C1: Language discrepancy. Rust's unique features which are absent from other languages (*e.g.*, ownership [2], and explicit lifetime [5]), pose challenges to existing source transformation-based DSU techniques, due to the nontrivial code transformation efforts required. For example, the most extensively evaluated DSU Ginseng [49] for C program requires substantial modifications (*i.e.*, 60k LoC) to the C compiler toolchain, which is nontrivial and complex to adapt to Rust's compiler `rustc` [64] directly.

Solution. To address this challenge, we propose a delegatecall proxy pattern approach to support the Rust programming

language, by modifying and extending Rust's official compiler `rustc` via a customized compiler pass on abstract syntax trees. This process partitions the Rust source code into the target and the patch candidates, in which the latter ones can be timely updated when vulnerabilities are detected. We will present the design details in section 5-B.

C2: Efficiency issues. Rust's efficiency design goal makes it challenging to leverage indirection-based DSU techniques [65] [66] directly, due to their considerable runtime penalties. For example, the widely evaluated DSU Upstare [50] incurs a significant overhead of 38.2%, due to the unwinding and rewinding code that its compiler adds to all functions.

Solution. To address this challenge, we propose a concurrent programming-based approach. Specifically, we have designed and implemented a multithreading injector that injects a dynamic updating thread into the target application automatically and transparently. To offer maximum efficiency benefits and programming flexibility, we adaptively support concurrent programming in RUSPATCH. Although message passing-based concurrency [67] is another choice, we focus on shared memory-based concurrency [68]. We will present the design details in section 5-C.

C3: Security threats. Rust's safety design goal makes existing binary rewriting-based DSU techniques [32] infeasible, for two key reasons. First, it is intrinsically difficult to validate the security patches in *binary forms*. Specifically, binary patches do not have the type information of Rust (*e.g.*, ownership[2]), hence, violations of type safety are difficult to detect at the binary level. While typed binaries (*e.g.*, typed assembly language [69], proof-carrying code [70], or recent WebAssembly [71]) are promising in providing type safety, no typed binaries have been investigated for Rust (to the best of our knowledge). Second, prior studies [72] have demonstrated that patches themselves written by developers might be faulty or buggy, leading to unintentional damages to the system being patched [73]. Therefore, to guarantee the security and trustworthiness, the target patches should be validated before being applied. Unfortunately, to the best of our knowledge, no such techniques have been thoroughly investigated for Rust.

Solution. To mitigate the security threats, we propose an automatic *source-level* validation approach, to validate the security patches before applying them. On the one hand, the validator validates the security patch in terms of security guarantees, by leveraging Rust's official checkers (*e.g.*, borrow checkers [33], and type checker [74]) as well as state-of-the-art checking tools (*e.g.*, Clippy [34], Rudra [22], and Miri[35]). On the other hand, we leverage differential testings and regressions to validate the functional correctness of the target application, preventing the introduction of potential bugs. We present the design details in section 5-D.

5. RUSPATCH DESIGN

In this section, we present the design of RUSPATCH in detail, by introducing the patching candidate analysis algorithm (§ 5-A), the delegatecall proxy pattern generation (§ 5-B), automated thread injection (§ 5-C), and patch validation (§ 5-D),

Constant	c		
BasicBlock	b	$\in$	$\mathbb{Z}$
Variable	x	$\in$	$\{x_0, x_1, x_2, \ldots\}$
Type	τ	::=	`bool` | `i8` | `u8` | `i16` | ...
BinOp	$\oplus$	::=	$+ \mid - \mid \times \mid / \mid < \mid == \mid \ldots$
Operand	o	::=	`const` c | `move` p | `copy` p
Place	p	::=	$x \mid *p \mid p.n \mid p[x]$
Rvalue	r	::=	$o \mid \&o \mid o_1 \oplus o_2 \mid o \ as \ \tau$
Statement	s	::=	$p = r \mid$ `storageLive`(x)
		$\mid$	`storageDead`(x)
Terminator	t	::=	$f(x) \mid$ `return` | `drop`(p) | `assert`(o)
		$\mid$	`goto`$(b) \mid$ `switch`$(o, (b_1, \ldots, b_n))$
Function	f	::=	[`unsafe`] `fn` $y(\vec{x : \tau})\{\vec{s}\ t\}$
Program	a	::=	$\vec{f}$

Figure 3: Core syntax of the Rust language.

respectively.

5.1. Patch Candidate Analysis

Rust core syntax model. To partition the source code into target code and patch candidates, we first need to identify patch candidates and then perform partitions by rewriting Rust programs. We conduct such an identification and partition on Rust abstract syntax tree (AST) data structures via customized compiler passes. To describe this process rigorously, we first present, in Fig. 3, a core syntax of Rust we designed, following prior work [21]. Specifically, a Rust program a consists of a list of function f, each of which has an optional `unsafe` key word indicating its explicit declaration of safety, followed by a list of statements $\vec{s}$ terminated with a terminator t.

A statement s may be an assignment $p = r$, or Rust-specific `storageLive` and `storageDead` marking the start and end of a variable x's lifetime, respectively.

A terminator t can be of distinct syntactic forms: 1) a function invocation $f(x)$; 2) a function `return`; 3) a deletion `drop`; 4) an assertion `assert`; 5) an unconditional jump `goto`; or 6) a conditional branch `switch`.

A place p stands for a location that can be assigned to, which includes a variable x, pointer references, tuple field selection, or array elements. A right value r consists of operands o, reference $\&$, binary operations, and type castings. Both right values r and operands o have Rust-specific syntactic features. For example, an operand o can be marked by either `move` or `copy`, representing the move or copy semantics of Rust [75] [76], respectively.

To simplify the presentation, we have omitted some features, such as pattern matching or control flow. However, these features can be added without any technical difficulty.

Patch candidate analysis algorithm. With this core syntax for Rust, we present, in Algorithm 1, how a Rust program is analyzed and then converted to use a delegatecall proxy pattern. The key idea for this algorithm is to: 1) identify all

Algorithm 1 : Delegatecall proxy pattern generation

Input: P: The Rust program
Output: (P', U): P' is Delegatecall proxy injected Rust program; and U is a set of patch candidates

1: **procedure** GEN-DELEGATECALL(P)
2: $P' = P$
3: $U, C =$IDENTIFY-PATCH-CANDIDATE(P)
4: **for** each function $f \in U$ **do**
5: $P' -= f$
6: **for** each call site $h \in C$ **do**
7: P'= ADDDELEGATECALL(P', h)
8: **return** P', U
9: **procedure** IDENTIFY-PATCH-CANDIDATE(P)
10: $U, C = \emptyset$
11: A = BUILDAST(P)
12: G = BUILDCALLGRAPH(P)
13: **for** each `unsafe` function $f \in A$ **do**
14: $U \cup= f$
15: **for** each call graph edge $(h \to f) \in G$ **do**
16: $C \cup= h$
17: **return** U, C

`unsafe` functions f and their corresponding call sites; and 2) rewrite all such call sites to use delegatecall proxy patterns. To implement this key idea, the GEN-DELEGATECALL() function takes as input a Rust program P, calculates and returns delegatecall proxy injected Rust program P' as well as a set of patch candidates U. This function consists of three key steps: first, this function calls IDENTIFY-PATCH-CANDIDATE() to generate a set of `unsafe` functions U and a set of their call sites C (line 3). Second, for each function f in the set U, the algorithm removes the function from the program, obtaining a result program P' (line 4 to 5). Third, for each call site h in the set C, the algorithm rewrites this call h with a corresponding delegatecall proxy pattern (line 6 to 7), whose details will be discussed next (§ 5-B).

The function IDENTIFY-PATCH-CANDIDATE() takes as input the Rust program P, and calculates a set of patch candidates U with their corresponding call sites U. Specifically, the procedure builds an AST A as well as a call graph G from the Rust program P (line 10 to 12), which are then traversed to calculate and return a patch candidate set U and a call site set C (line 13 to 17).

This algorithm is efficient, for an AST of n functions and m call sites, the computational complexity is $O(n * m)$ for patch candidate analysis, and is $O(m)$ for delegatecall proxy generation.

5.2. Delegatecall Proxy Pattern Code Generator

The addDelegateCall() function in Algorithm 1 rewrites the original Rust AST, to introduce proxy patterns. To formalize this process of AST traversal and rewriting, we introduce a group of syntax-indexed translation functions $[\![\cdot]\!]_f$, $[\![\cdot]\!]_s$, and $[\![\cdot]\!]_t$ to translate a function f, a statement s, and a terminator t, respectively.

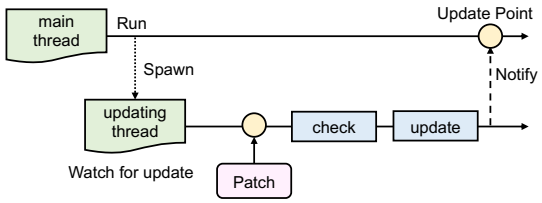

Figure 4: The updating process of RUSPATCH, with the aid of an auxiliary updating thread.

First, the translation $[\![\cdot]\!]_f$ of a function f is defined by recursively invoking the translation $[\![\cdot]\!]_s$ and $[\![\cdot]\!]_t$.

$$[\![\texttt{[unsafe]}\ y(\tau\ x)\{\vec{s}\ t\}]\!]_f \Rightarrow \texttt{[unsafe]}\ y(\tau\ x)\{\vec{[\![s]\!]}_s\ [\![t]\!]_t\} \tag{1}$$

Second, the translation $[\![\cdot]\!]_s$ is defined on a statement s, which does not change the syntactic forms of s.

$$[\![p = r]\!]_s \Rightarrow p = r \tag{2}$$
$$[\![\texttt{storageLive}(x)]\!]_s \Rightarrow \texttt{storageLive}(x) \tag{3}$$
$$[\![\texttt{storageDead}(x)]\!]_s \Rightarrow \texttt{storageDead}(x) \tag{4}$$

Finally, the translation function $[\![\cdot]\!]_t$ translates a terminator t, for which only rules for function invocation, drop, and goto are presented while others are omitted for clarity.

$$[\![f(x)]\!]_t \Rightarrow \begin{cases} f' = \texttt{findSym}(\text{“}f\text{”}); f'(x) & f \in U \\ f(x) & f \notin U \end{cases} \tag{5}$$
$$[\![\texttt{drop}(p)]\!]_t \Rightarrow \texttt{drop}(p) \tag{6}$$
$$[\![\texttt{goto}(b)]\!]_t \Rightarrow \texttt{goto}(b) \tag{7}$$

To translate a patch candidate function invocation $f(x)$ (*i.e.*, $f \in U$), we load the function's value dynamically with "`findSym`" and then invoke it; otherwise, for other functions (*i.e.*, $f \notin U$), the invocation remains unchanged. We have deliberately left the concrete implementation of the function `findSym` abstract here, to allow for flexibility in practical implementation. In the next section (§ 6), we will showcase an implementation strategy based on dynamically loaded modules.

5.3. Updating Thread Injection

Dynamic software updating may introduce potential runtime overhead to the target program being updated, due to the dynamic monitoring and applying of available patches. To mitigate this issue, we have designed a strategy to automatically inject an updating thread into the target program.

We present, in Fig. 4, the overall process of injecting an updating thread. When the main thread starts executing, it spawns a new updating thread watching for potential vulnerability patches. After identifying such patches, the updating thread validates (to be discussed in § 5-D) and updates the patches, before notifying the main thread. Finally, the main thread loads and executes the updated function upon receiving the notifications.

As the main thread executes concurrently with the updating thread, only thread spawn and notification may incur extra runtime overhead to the main thread. Furthermore, as dynamic software updating might not occur very frequently, the extra overhead is low in practice as our experimental results demonstrated (§ 7-F).

5.4. Patch Validation

A patch itself may be vulnerable or buggy [72] [77]. Hence, we designed a patch validator to validate and test the patches before deploying them. The patch validator performs: 1) security verification; and 2) functional correctness checking. First, RUSPATCH utilized an architecture of security plugins to guarantee the security of the target patch. Specifically, the plugins include not only the official borrow checker from the Rust compiler, but also third-party state-of-the-art vulnerability detection tools, to determine whether vulnerabilities have been successfully fixed without introducing new ones. Second, to guarantee the functional correctness of the target application, RUSPATCH utilized both a differential testing approach to guarantee that the functionality of the application has remained unchanged before and after the patch is applied. Furthermore, RUSPATCH also utilized a regression testing approach to ensure the normal functionality has not been altered by the patch, by leveraging the test cases distributed with the project (if any). If either test fails, the patch is rejected and test results are returned to developers for further investigation.

6. IMPLEMENTATION

We have implemented a prototype system for RUSPATCH using Rust, which has been distributed in our open source.

Delegatecall proxy generator and thread injector. We implemented the delegatecall proxy pattern generator and thread injector by developing a customized Rust compiler leveraging an open source parsing library `syn` [78]. We implemented the `VisitMut` trait in the `syn` package to borrow and visit its nodes while traversing the abstract syntax tree. We implement the `visit_item_mut` method to detect whether a Rust function is unsafe (*i.e.*, marked by `unsafe` or contains an unsafe code block). Unsafe functions are then extracted and encapsulated into a Rust submodule as patch candidates, which are further compiled into a dynamic link library. We also implemented the `visit_expr_mut` method to iterate over each expression and check whether the function call in it invokes a patch candidate. We implemented the thread injector using the `modified` method from the Rust standard library.

Dynamic loading and concurrency. We implemented dynamic loading at runtime by leveraging a Rust open source library `libloading` [79], which provides a group of APIs to load dynamic libraries to use the functions and static variables in them. Specifically, we use the API `Library::new<P: AsRef<OsStr>>(f: P)` to find and load a dynamic library from the file `f`. Our prototype utilized the Rust shared memory concurrency model to implement both the dynamic library and patched function pointers. To guarantee the correctness of concurrent accesses, we utilized the Rust's `RwLock` synchronization primitives [80], to protect the shared variables between the main and the updating thread.

Patch validator and testing. We have made use of an architecture of security plugins to implement the patch validator. Specifically, we have incorporated diverse security checking tools into our prototype implementation including not only the official `rustc` borrow checker [33] and Clippy [34], but also open-source state-of-the-art checking tools including Miri [35] and Rudra [22]. Among these tools, Clippy is a static analysis tool that detects common security issues, whereas Miri detects potential memory safety issues by symbolically executing the code. We have leveraged the tools cargo-referendum [81] and Rust regression [82], for differential and regression testings, respectively. It should be noted that the plugin architecture is extensible, making the incorporation of other tools straightforward.

7. EVALUATION

In this section, we conduct experiments to evaluate RUSPATCH.

7.1. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. As RUSPATCH is proposed to automatically deploy patches for Rust applications, is it effective in fixing vulnerabilities?

RQ2: Compiling performance. What is the performance of RUSPATCH to compile Rust projects?

RQ3: Runtime overhead. As RUSPATCH is designed to rectify vulnerabilities dynamically, does RUSPATCH incur additional runtime overhead to the Rust applications being patched?

RQ4: Usefulness. As RUSPATCH is introduced to help end Rust developers, is it useful to help them deploy patches?

7.2. Experimental Setup

We conduct all experiments on the publicly available CloudLab cloud infrastructure testbed [36] (we include in our open source the CloudLab [83] profile that automatically instantiates the software environment used in this evaluation). Our experiments utilize a CloudLab c220g5 server configured with two Intel Xeon Silver 4114 10-core CPUs running at 2.20 GHz, 192 GB RAM, and a dual-port Intel X520 10Gb NIC. The machine runs 64-bit Ubuntu 20.04 Linux with kernel version 5.4.0. In all experiments, we disable hyperthreading, turbo boost, and frequency scaling to reduce the variance in benchmarking.

7.3. Dataset

To conduct the evaluation, we selected and created two datasets: 1) microbenchmarks; and 2) real-world and large Rust applications, which are included in our open source.

Microbenchmarks. Answering the above research questions requires a vulnerability dataset as a ground truth, however,

TABLE I: The RusBench benchmark, and its experimental results.

CVE number	Package	Type	Description	Patched?	Original(s)	RUSPATCH(s)
CVE-2017-1000430	base64	OOB	Buffer overflow in calculating the buffer size.	✓	0.538	0.582
CVE-2018-1000810	std	OOB	Integer overflows leading to an out of bounds write.	✓	0.553	0.559
CVE-2019-15551	smallvec	DF	Double free for certain grows with the current capacity.	✓	0.844	0.897
CVE-2019-16140	isahc	UAF	`From` function returns freed memory.	✓	0.522	0.557
CVE-2019-16880	linea	DF	Double free in trait implementation of panic.	✓	0.838	0.906
CVE-2020-25794	sized-chunks	UNINIT	Clone can have a memory-safety issue upon a panic.	✓	0.596	0.611
NA	Servo	UAF	Object dropped earlier and pointer used later.	✓	0.591	0.593
NA	Servo	UNINIT	Panic when font face name is not available.	✓	0.543	0.572
NA	Tock	UNINIT	Allocator dropping uninitialized memory.	✓	0.876	0.882
NA	Redox	UNINIT	Using uninitialized memory.	✓	0.552	0.563

TABLE II: Macro benchmarks of 5 real-world projects.

Project	Domain	LoC	#Files	Github Stars (k)
RisingWave [86]	Database	291,017	1,471	4.4
Polars [87]	Data Process	138,764	832	17.3
Wasmer [88]	Wasm runtime	350,040	762	15.3
Diem [89]	Blockchain	300,399	1,570	16.7
Rocket [90]	Web	31,902	315	20.7

such a dataset does not exist (to the best of our knowledge). To this end, we took the first step of manually creating **RusBench**, a dataset containing real-world vulnerabilities of diverse types, which we created by inspecting Rust CVE [84], RustSec [85] as well as previous papers [17] [48].

As shown in TABLE I, our focus in creating this dataset is on the diversity of vulnerabilities. We selected ten representative micro-benchmarks by analyzing various vulnerability behavior patterns, including out-of-bounds access (OOB), use-after-free (UAF), double-free (DF), and uninitialized memory (UNINIT) (shown in the 3rd column **Type**). As these microbenchmarks were derived from prior research papers, they have undergone thorough analysis and we contend that our dataset embodies a broad spectrum of vulnerability types, encompassing critical ones. Furthermore, we are continuing to expand it by including more benchmarks covering other vulnerability types.

Real-world applications. For better benchmark representativeness, we choose real-world Rust projects that: 1) have 1,000+ stars, which indicates popularity—a criterion used in prior work [91]; 2) are frequently updated and maintained; 3) are cloud service related; and 4) contain `unsafe` code.

According to these selection criteria, we present, in TABLE II, five real-world applications from diverse domains: 1) RisingWave [86]: a distributed SQL *database* for stream processing; 2) Polars [87]: a *data processing* library for Rust; 3) Wasmer [88]: a lightweight WebAssembly (Wasm) *runtime*; 4) Diem [89]: a distributed *blockchain* system; and 5) Rocket [90]: an async *Web* framework for Rust.

7.4. RQ1: Effectiveness

To answer **RQ1** by demonstrating the effectiveness of RUSPATCH, we conduct experiments by evaluating RUSPATCH against the microbenchmark RusBench.

We first compiled and executed the vulnerable programs in **RusBench**, whose execution caused program crashes for specific input triggering the bugs. Next, we processed these benchmarks with RUSPATCH, then compiled and executed them for a second time. During program execution, RUSPATCH patched the target Rust program with the security patches we supplied. The experimental results demonstrated that RUSPATCH successfully patched all these benchmarks (column "**Patched?**" in TABLE I), without terminating or restarting the vulnerable programs.

Summary. These experiment results demonstrated that RUSPATCH is effective in deploying patches for real-world CVEs and program bugs automatically.

7.5. RQ2: Compiling Performance

To answer **RQ2** by investigating RUSPATCH's performance in compiling Rust projects, we conducted experiments to measure the time RUSPATCH spent in compilation. Each program is compiled for 10 rounds to calculate an average.

We present, in TABLE I, the average compiling time to process each Rust program (column **RusPatch**). The maximum compiling time for RUSPATCH is 0.906 seconds. We further

TABLE III: Execution time overhead.

Operations	Original (ms)	RUSPATCH(ms)	Overhead
`unchecked`	1709.97 ± 3.19	1722.36 ± 14.05	+0.72%
`offset`	1424.64 ± 3.01	1434.14 ± 2.15	+0.67%
`copy`	252.61 ± 1.30	260.92 ± 0.58	+3.28%

compared with the compiling time with the original `rustc` compiler (column **Original**), and observed the difference is negligible. Furthermore, the static compiling time not only introduces no runtime penalties, but also is in par with prior work [21] on Rust static analysis.

Summary. RUSPATCH is efficient in compiling Rust programs, in line with the official compiler `rustc`.

7.6. RQ3: Runtime Overhead

To answer **RQ3** by investigating the overhead RUSPATCH introduced to the target program, we chose microbenchmarks to measure the performance changes of the target program before and after deploying RUSPATCH. Following prior work [92], we selected microbenchmarks to measure overhead, as performance benefits or losses would likely be small and difficult to observe due to the ambient noise present in an application-level benchmark.

We testified micro benchmarks for three usage scenarios where `unsafe` is extensively used to improve performance [17]: 1) traversing an array with `unsafe` memory access `slice::get_unchecked()` with no boundary checking (abbr. as `unchecked` in TABLE III); 2) traversing an array by raw pointers `ptr::offset()`; and 3) `unsafe` memory copy `ptr::copy_nonoverlapping()` (abbr. as `copy`). Benchmarks 1) and 2) each include 100 million operations, while benchmark 3) includes a billion operations. Compiler optimizations were disabled to ensure that these operations were not optimized to distort performance.

We present, in TABLE III, the results. We ran each benchmark 10 times and reported the median time for the original program, as well as for RUSPATCH. The last column **Overhead** is calculated by $Overhead = \text{RUSPATCH}/Original - 1$.

Summary. The experimental results demonstrated that the overhead RUSPATCH introduced ranges from 0.67% to 3.28%, and thus is insignificant.

7.7. RQ4: Usefulness

Developer Background. To quantify the manual effort needed to patch Rust programs and evaluate the usefulness of RUSPATCH, we conducted a developer study. We hired six Rust developers to conduct this study, and all of them have extensive experience in using Rust: they have been using Rust as their primary developing language for more than 3 years. However, they are not familiar with developing or using a DSU system. Therefore, we can quantify the effort required for a normal Rust developer to use a DSU system like RUSPATCH in this work.

TABLE IV: Average time (in minutes) to finish each task, without and with RUSPATCH.

CVE	Task1 (m)		Task2 (m)	
	$\mathbf{M}^1$	$\mathbf{R}^2$	$\mathbf{M}^1$	$\mathbf{R}^2$
CVE-2017-1000430[93]	39.3	4.2	12.3	2.6
CVE-2019-16140[58]	35.7	2.8	12	1.8
CVE-2019-16881[94]	47.7	2.8	39.3	2.1
CVE-2020-25794 [95]	26.1	2.1	23.3	1.7

[1] and [2] are abbreviations of Manual, and RUSPATCH, respectively.

Methodology. Throughout our study, we asked the developers to finish two tasks: 1) converting three vulnerable programs to DSU programs, manually or using RUSPATCH; 2) deploying patches for running Rust programs, manually or using RUSPATCH. The two tasks target different scenarios: the first task measures the efforts required to convert a Rust program to a dynamically updatable version, while the second task demonstrates efforts in validating and deploying patches. For all tasks, we measured the time required by the developer to finish the task.

We present, in TABLE IV, the time used to finish the two tasks, respectively.

Converting to a DSU program. To conduct the evaluation, we wrote four programs by adapting four real-world CVEs with diverse complexity and sizes (in terms of lines of code): 1) CVE-2017-1000430 [93] (356 LoC); 2) CVE-2019-16140 [58] (212 LoC); 3) CVE-2019-16881 [94] (534 LoC); and 4) CVE-2020-25794 [95] (136 LoC).

We first provided developers with a short description of the delegatecall proxy pattern and asked them to convert the given vulnerable programs into delegatecall proxy pattern. The developers required an average of 43 minutes to convert the source code into a delegatecall proxy pattern code.

Next, we asked to finish this task using RUSPATCH. Since RUSPATCH does not require any prior knowledge of delegatecall proxy pattern, developers were able to convert a correct DSU program in a maximum of 4.4 minutes.

Dynamic deployment of patches. We provided developers with both the already converted DSU Rust program and a security patch, and asked them to deploy the patch to the running DSU program.

In manual deployment, the developers needed to manually check whether the patch was correct and whether the functionality of the source program had changed, then compiled it into a dynamic loadable module and deployed it onto the CloudLab. They spent 23.3 minutes on average to finish this task.

Next, developers used RUSPATCH to automate the patch validation and deployment. Due to the RUSPATCH's advantage of full automation, the time used to finish this task is less than 2.6 minutes.

Summary. These experimental results demonstrated the practical usefulness of RUSPATCH to end Rust developers, who

TABLE V: Build time and generated binary sizes for real-world projects, without and with RUSPATCH.

Program	Build Time (s)		Size (MB)	
	Original	RUSPATCH	Original	RUSPATCH
RisingWave	280.41	281.44	1809.31	1810.08
Polars	36.37	36.76	927.38	927.98
Wasmer	587.99	598.88	11.54	11.62
Diem	186.18	188.82	477.61	470.62
Rocket	43.85	46.08	111.69	112.58

might even have no prior knowledge or experience with dynamic software updating.

7.8. Real-world Applications

To further investigate the usefulness of RUSPATCH to real-world Rust applications, we conducted experiments on large and in-the-wild Rust applications, as presented in TABLE II. As these benchmarks come with no ground truth, we thus took the fault injection [96] approach, a technique of software testing by introducing faults to the code being tested.

To realize this process, we intentionally introduced a piece of `unsafe` code into the project's original source code, making it vulnerable. We then compiled it to obtain an unsafe version of binary: `prog.unsafe`. We then utilized RUSPATCH to transform the unsafe source code into a DSU-enabled binary: `prog.dsu`. We then deployed the two versions onto CloudLab, our evaluation platform, for evaluation.

We first evaluated the effectiveness of RUSPATCH on the real-world applications, and the results demonstrated that RUSPATCH patched the introduced vulnerabilities successfully. We then measure the build time of the original `rustc` compiler and RUSPATCH, and the generated binary sizes, for these real-world applications. We present, In TABLE V, a detailed comparison of build time and binary sizes. The experimental results demonstrated that the difference in build time between `rustc` and RUSPATCH is negligible, and the difference between generated binary sizes is insignificant.

Summary. RUSPATCH is effective and useful to real-world Rust programs, by introducing insignificant build time and binary size differences.

8. DISCUSSION

In this section, we discuss some limitations of this work, along with directions for future work. It should be noted that this work represents the first step towards timely and effectively patching Rust applications.

Binary patching. RUSPATCH is designed to support source-level patching, but does not support binary patching when the source code is absent. However, binary patching Rust is difficult, as Rust, being a young language, does not have a standard application binary interface. In the meanwhile, as current Rust compiler uses LLVM [97] as its backend, hence it is interesting to investigate intermediate representation level patching techniques, by leveraging LLVM instrumentations (*e.g.*, Instrew [98], or Dbill [99]). We leave it for future work.

Human interventions. Although RUSPATCH is designed to be fully automated, manual interventions are still needed in two scenarios: 1) vulnerability rectification; and 2) failure inspections. Specifically, RUSPATCH requires a patch to rectify a vulnerability, which is often crafted by a developer manually. While manual bug fixing is an established software engineering practice, recent studies have shown the promising potential of automatic program rectifications [23], which we leave as an important direction for future exploration.

Patching granularity. RUSPATCH applies patches on a function granularity. In the meanwhile, prior studies (*e.g.*, stack reconstruction [50], or binary rewriting [51]) showed promising potentials of fine-grained patching strategies on basic block granularities. We leave this as a future direction to explore.

9. RELATED WORK

In recent years, there have been a significant amount of studies on Rust security and dynamic software updating. However, the work in this paper stands for a novel contribution to these fields.

Rust security. The use of unsafe Rust has been extensively studied. Evans et al. [18] conducted a large-scale empirical study on the use of the unsafe mechanism in real-world Rust applications. Qin et al. [17] conducted an empirical study of 850 examples of unsafe code in real Rust programs to propose common unsafe patterns. Astrauskas et al. [19] studied empirically unsafe code usage scenarios in practice by analyzing a large corpus of Rust projects.

Existing studies have made extensive use of static or dynamic analysis for vulnerability detection. SafeDrop[20] detects memory corruption by performing alias analysis and taint analysis on Rust MIR. MirChecker [21] statically detects runtime crashes and memory safety vulnerabilities caused by dangling pointers. Rudra [22] detects memory vulnerabilities by running dataflow analysis and send/sync difference checking algorithms on the Rust ecosystem. Rupair [23] automatically repair Rust buffer overflow vulnerabilities caused by integer overflows.

However, all of the above studies focus only on static vulnerability detection and rectification, but do not consider dynamic vulnerability rectification issues like our work.

Dynamic software updating. Ginseng [49] rewrote the program's C source code to support dynamic updates, by using function indirection and type wrapping. Ksplice [54] analyzes runtime updates at an object-code level, enabling automated live-patch generation in many cases. UpStare [50] performs source code-level conversion and uses the stack reconstruction technique to update versions. DUSC [66] used proxy classes to update Java programs at runtime by substituting, adding, and removing classes. Orthus [57] copies the whole state of all active VMs from the running hypervisor to a new hypervisor to conduct hypervisor-level runtime patching.

However, all these studies cannot be applied to the Rust language, due to the feature discrepancies between Rust and

other languages.

10. CONCLUSION

In this work, we present RUSPATCH, the first framework to timely and effectively patch Rust applications. RUSPATCH consists of two main phases: static code partitioning and dynamic patch deployment. We implemented a prototype for RUSPATCH and conducted extensive experiments with it. Experimental results demonstrated that RUSPATCH is effective, efficient, and useful. Overall, our work is a call to arms for further hardening the Rust ecosystem, making the promise of a secure programming language a reality.

REFERENCES

[1] "Rust Programming Language," https://www.rust-lang.org/.
[2] "What Is Ownership?" https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.
[3] "Borrowing," https://doc.rust-lang.org/rust-by-example/scope/borrow.html.
[4] "References and Borrowing," https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.
[5] "Lifetimes - The Rust Programming Language," https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html.
[6] "Tock Embedded Operating System," https://www.tockos.org/.
[7] "TTstack," https://github.com/rustcc/TTstack.
[8] "Smoltcp: A smol tcp/ip stack," https://github.com/smoltcp-rs/smoltcp.
[9] "Tokio - An asynchronous Rust runtime," https://tokio.rs/.
[10] "TiKV: Distributed transactional key-value database, originally created to complement TiDB," https://github.com/tikv/tikv.
[11] "Parity-ethereum: The fast, light, and robust client for Ethereum-like networks," https://github.com/openethereum/parity-ethereum.
[12] "Unsafe Rust," https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.
[13] "CVE-2020-35879," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35879.
[14] "CVE-2018-1000810," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000810.
[15] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th USENIX Security Symposium (USENIX Security 98)*, 1998.
[16] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.
[17] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. London UK: ACM, Jun. 2020, pp. 763–779.
[18] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 246–257.
[19] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, Nov. 2020.
[20] M. Cui, C. Chen, H. Xu, and Y. Zhou, "SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–21, Oct. 2023.
[21] Z. Li, J. Wang, M. Sun, and J. C. Lui, "MirChecker: Detecting Bugs in Rust Programs via Static Analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 2183–2196.
[22] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. Virtual Event Germany: ACM, Oct. 2021, pp. 84–99.
[23] B. Hua, W. Ouyang, C. Jiang, Q. Fan, and Z. Pan, "Rupair: Towards Automatic Buffer Overflow Detection and Rectification for Rust," in *Annual Computer Security Applications Conference*. Virtual Event USA: ACM, Dec. 2021, pp. 812–823.
[24] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with XRust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 234–245.
[25] E. Rivera, S. Mergendahl, H. Shrobe, H. Okhravi, and N. Burow, "Keeping Safe Rust Safe with Galeed," in *Annual Computer Security Applications Conference*. Virtual Event USA: ACM, Dec. 2021, pp. 824–836.
[26] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, Jan. 2018.
[27] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, Oct. 2019.
[28] "Firecracker," https://firecracker-microvm.github.io/.
[29] "Ensure Cost Balances With Risk in High-Availability Data Centers," https://www.gartner.com/en/documents/3906266.
[30] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, General-Purpose Dynamic Software Updating for C," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 4, pp. 1–38, Oct. 2014.
[31] Z. Zhao, Y. Jiang, C. Xu, T. Gu, and X. Ma, "Synthesizing Object State Transformers for Dynamic Software Updates," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1111–1122.
[32] L. Zhou, F. Zhang, J. Liao, Z. Ning, J. Xiao, K. Leach, W. Weimer, and G. Wang, "KShot: Live Kernel Patching with SMM and SGX," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Valencia, Spain: IEEE, Jun. 2020, pp. 1–13.
[33] "Rust borrow checker," https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html.
[34] "Rust-clippy," https://github.com/rust-lang/rust-clippy.
[35] "Miri," https://github.com/rust-lang/miri.
[36] "CloudLab," https://cloudlab.us/.
[37] G. Hoare, "Project Servo," http://venge.net/graydon/talks/intro-talk-2.pdf, 2010.
[38] R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook: The Art of Automatic Memory Management*, 1st ed. Boca Raton, FL: CRC Press, 2016.
[39] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Nov. 2014.
[40] "Stack Overflow Developer Survey 2022," https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022.
[41] C. Catalin, "Microsoft to explore using Rust," https://www.zdnet.com/article/microsoft-to-explore-using-rust/, 2019.
[42] "Rust in the Android platform," https://security.googleblog.com/2021/04/rust-in-android-platform.html.
[43] "Rust in the Linux kernel," https://security.googleblog.com/2021/04/rust-in-linux-kernel.html.
[44] "Mononoke," https://github.com/facebookexperimental/eden.
[45] "Why Discord is switching from Go to Rust," https://discord.com/blog/why-discord-is-switching-from-go-to-rust.
[46] "IotEdge," https://github.com/Azure/iotedge/tree/main/edgelet.
[47] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the Servo Web Browser Engine Using Rust," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 81–89.
[48] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1–25, Jan. 2022.
[49] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 72–83, Jun. 2006.
[50] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. USA: USENIX Association, Jun. 2009, p. 31.
[51] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "POLUS: A POwerful Live Updating System," in *29th International Conference on Software*

Engineering (ICSE'07). Minneapolis, MN: IEEE, May 2007, pp. 271–281.
[52] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1289–1306.
[53] M. Prasad, "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks," in *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, 2003.
[54] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the Fourth ACM European Conference on Computer Systems - EuroSys '09*. Nuremberg, Germany: ACM Press, 2009, p. 187.
[55] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive Android Kernel Live Patching," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1253–1270.
[56] C. Niesler, S. Surminski, and L. Davi, "HERA: Hotpatching of Embedded Real-time Applications," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021.
[57] X. Zhang, X. Zheng, Z. Wang, Q. Li, J. Fu, Y. Zhang, and Y. Shen, "Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 93–105.
[58] "CVE-2019-16140," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16140.
[59] S. Andersen and V. Abella, "Data execution prevention. changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies," 2004.
[60] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: Association for Computing Machinery, Oct. 2004, pp. 298–307.
[61] S. Lai, X. Yuan, J. K. Liu, X. Yi, Q. Li, D. Liu, and S. Nepal, "OblivSketch: Oblivious Network Measurement as a Cloud Service," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021.
[62] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "DDoS attack protection in the era of cloud computing and Software-Defined Networking," *Computer Networks*, vol. 81, pp. 308–319, Apr. 2015.
[63] L. De Carli, R. Sommer, and S. Jha, "Beyond Pattern Matching: A Concurrency Model for Stateful Deep Packet Inspection," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. Scottsdale Arizona USA: ACM, Nov. 2014, pp. 1378–1390.
[64] "Rust Compiler Development Guide," https://rustc-dev-guide.rust-lang.org/.
[65] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A VM-centric approach," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 1–12, Jun. 2009.
[66] A. Orso, A. Rao, and M. Harrold, "A technique for dynamic updating of Java software," in *International Conference on Software Maintenance, 2002. Proceedings.* Montreal, Que., Canada: IEEE Comput. Soc, 2002, pp. 649–658.
[67] "Using Message Passing to Transfer Data Between Threads - The Rust Programming Language," https://doc.rust-lang.org/book/ch16-02-message-passing.html.
[68] "Shared-State Concurrency - The Rust Programming Language," https://doc.rust-lang.org/book/ch16-03-shared-state.html.
[69] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system F to typed assembly language," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 3, pp. 527–568, May 1999.
[70] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97*. Paris, France: ACM Press, 1997, pp. 106–119.
[71] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, Jun. 2017, pp. 185–200.
[72] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, Jul. 2015, pp. 24–36.
[73] H. Kim, M. O. Ozmen, Z. B. Celik, A. Bianchi, and D. Xu, "PatchVerif: Discovering Faulty Patches in Robotic Vehicles," in *32th USENIX Security Symposium (USENIX Security 23)*, 2023.
[74] "Type checking - Rust Compiler Development Guide," https://rustc-dev-guide.rust-lang.org/type-checking.html.
[75] "The move semantics in Rust," https://doc.rust-lang.org/std/keyword.move.html.
[76] "The copy semantics in Rust," https://doc.rust-lang.org/std/marker/trait.Copy.html.
[77] M. Böhme and A. Roychoudhury, "CoREBench: Studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 105–115.
[78] "Syn - Parser for Rust source code," https://crates.io/crates/syn.
[79] "Libloading," https://crates.io/crates/libloading.
[80] "RwLock in std::sync - Rust," https://doc.rust-lang.org/std/sync/struct.RwLock.html.
[81] "Cargo referendum," https://crates.io/crates/cargo-referendum.
[82] "Cargo regression," https://crates.io/crates/regression.
[83] R. Ricci, E. Eide, and C. Team, "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications," *;login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.
[84] "Rust CVE," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust.
[85] "RustSec Advisory Database," https://rustsec.org/.
[86] "RisingWave," www.risingwave.dev.
[87] "Polars," https://github.com/pola-rs/polars.
[88] "Wasmer: The leading WebAssembly Runtime supporting WASIX, WASI and Emscripten," https://github.com/wasmerio/wasmer.
[89] "Diem: A decentralized, programmable distributed ledger," https://github.com/diem/diem.
[90] "Rocket - Simple, Fast, Type-Safe Web Framework for Rust," https://rocket.rs/.
[91] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Communications of the ACM*, vol. 60, no. 10, pp. 91–100, Sep. 2017.
[92] F. Rommel, C. Dietrich, M. Rodin, and D. Lohmann, "Multiverse: Compiler-Assisted Management of Dynamic Variability in Low-Level System Software," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, Mar. 2019, pp. 1–13.
[93] "CVE-2017-1000430," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000430.
[94] "CVE-2019-16881," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16881.
[95] "CVE-2020-25794," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25794.
[96] Mei-Chen Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.
[97] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86.
[98] A. Engelke and M. Schulz, "Instrew: Leveraging LLVM for high performance dynamic binary instrumentation," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 172–184.
[99] Y.-H. Lyu, D.-Y. Hong, T.-Y. Wu, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew, "DBILL: An efficient and retargetable dynamic binary instrumentation framework using llvm backend," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 141–152.