RPATCH: Towards Timely and Effectively Patching Rust Applications

Yufei Wu Baojian Hua*
School of Software Engineering
University of Science and Technology of China
wuyf21@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Rust is an emerging programming language designed for system software by providing security guarantees and runtime efficiency, and has been used widely in many software infrastructures such as OS kernel, Web browsers, cloud services, and blockchains. However, as Rust introduce the `unsafe` sublanguage, Rust applications are fragile and vulnerable. Existing studies on Rust security focus on static vulnerability detection or rectification, but ignore the problem of dynamic vulnerability rectifications. In this paper, we present RPATCH, an infrastructure for automatically patching Rust applications. RPATCH consists of three key components: 1) a *delegatecall-proxy pattern generator* ; 2) an *automatic multithreading generator*; and 3) a *modular testing and vulnerability replaying component*. We designed and implemented a prototype for RPATCH, and conducted extensive experiments to evaluate its effectiveness, cost, and usefulness on micro and macro-benchmarks. Experimental results demonstrate that RPATCH is effective to patch real-world Rust applications, the extra cost RPATCH introduced is low and insignificant. Futhermore, we conducted a developer study to show that RPATCH is easy to use.**

***Index Terms*—Rust, Security**

I. INTRODUCTION

Rust [1] is an emerging programming language for constructing safe system software. With its design goals of both security and efficiency, Rust introduced a group of novel language features such as *ownership* [2], *borrow* [3], *reference* [4], and explicit *lifetime* [5]. Due to the expressiveness and convenience of these advanced programming features, Rust has gained popularity in the past several years, and has been used successfully in the development of many system software like operating system kernels [6] [7] [8] [9] [10], Web browsers [11], file systems [12] , cloud services [13], network protocol stacks [14], language runtimes [15], databases [16], and blockchains [17].

To support arbitrary low-level operations and offer more flexibility to developers, Rust introduced an unsafe sublanguage via the `unsafe` [18] feature. Essentially, the `unsafe` Rust code not only bypasses compiler static checks but also disables runtime checks, thus may break Rust's security guarantees and may (sadly) lead to severe vulnerabilities. For example, for buffer access `buf[i]`, the unsafe Rust sublanguage does not check the index `i` against the buffer range, which may trigger out-of-bounds (OOBs) memory accesses. Such OOBs may further lead to memory corruptions such as arbitrary address overwriting [19] [20]. As a result, existing Rust security studies demonstrated that *all* reported memory-safety bugs are caused by Rust's `unsafe` feature [21] [22].

To address the Rust security issues, there have been a significant amount of research efforts on Rust security, such as empirical security study [22] [23] [24], vulnerability detection [25] [26] [27] [28] [29] [30] [31], security enhancement [32] [33] [34] [35] [36] [37], and formal verification [38] [39] [40] [41]. Although these research efforts made considerable progress in securing Rust applications, they, unfortunately, have severe limitations: they only focus on static vulnerability detection or rectification, but ignore the problem of *dynamic vulnerability rectifications* (DVR). Here, DVR represents rectifying vulnerabilities in a running Rust application without stopping or restarting the target application. Given the design goals of Rust to be a safe system language, DVR is important to achieve safety in an automated and timely manner. For example, Amazon AWS introduced Firecracker [42], a new virtualization service written in Rust. For online services like Firecracker, once a vulnerability is detected, it would be time-consuming, labor-intensive and error-prone to rectify and patch them offline. Worse yet, online services have to stopped or restarted for an unexpected delay, which are difficult or even impossible to perform in cloud computing scenarios.

To this end, we propose the use of dynamic software updating (DSU) technologies [43] to solve the DVR problem. Although there have been a significant amount of studies on dynamic software updating, they cannot solve the Rust DVR problem for three key challenges: 1) *language feature discrepancy*, 2) *performance issues*, and 3) *incompleteness*, hindering its practical usefulness. First, the novel language features of Rust, such as ownership [2] and explicit lifetime [5], do not exist in other languages such as C or Java; hence, it is unclear whether or not it is possible and how to apply existing DSU to Rust. On the contrary, we believe a Rust semantics-aware technique should be investigated.

Second, due to the high performance requirements of Rust applications, existing DSU techniques is not feasible, due to their considerable runtime penalties. For example, the overhead of DynSec [44] is more than one hundred seconds.

Finally, DSU alone is not a panacea. Instead, to guarantee

* Corresponding author.

the correct rectification of vulnerabilities as well as the normal functionalities of the rectified Rust applications, DSU should be accompanied by buddy systems such as vulnerabilities replaying. Unfortunately, to the best of our knowledge, no such systems have been thoroughly investigated for Rust.

To address the aforementioned challenges, our goal, in this paper, is to design and implement the *first* infrastructure to solve the Rust DVR problems timely and effectively. To achieve this goal, we propose a framework dubbed RPATCH, which consists of three key components: 1) a *delegatecall-proxy pattern generator* to split the Rust code into delegatecall proxy code and updatable candidate code effectively and automatically; 2) an *automatic multithreading generator* to generate dynamic updating thread; and 3) a *modular testing and vulnerability replaying component* to automate the process of testing, validating and deploying patches dynamically.

We implemented a prototype for RPATCH, and conducted extensive experiments to evaluate its effectiveness, cost, and usefulness. First, to evaluate the effectiveness of RPATCH, we applied RPATCH to 14 vulnerable Rust programs adapted from real-world Rust CVEs, and experimental results demonstrate that RPATCH is effective in fixing all these Rust vulnerabilities automatically. Second, to evaluate the cost of RPATCH, we applied RPATCH to micro-benchmarks, experiments results demonstrate that the runtime overhead RPATCH introduced is less than 140 milliseconds. Finally, a developer study shows RPATCH is easy to use: as RPATCH deploys patches for the target Rust applications automatically, no developer intervention or manual code rewriting is required.

To the best of our knowledge, this work represents the first step towards solving Rust DVR problems. To summarize, this work makes the following contributions:

- We presented the first infrastructure dubbed RPATCH to solve Rust DVR problems.
- We implemented a prototype of RPATCH, to deploy patches for Rust applications effectively and dynamically.
- We conducted extensive experiments to demonstrate the effectiveness, cost, and usefulness of RPATCH to real-world Rust CVEs and applications.

The rest of this paper is organized as follows. Section II presents an overview of the background for this work. Section III introduces the motivation for this work as well as the threat model. Section IV presents the design of RPATCH and Section V presents the implementation. Section VI presents the experiments we performed, and answers to the research questions based on the experimental results. Section VII discusses directions limitations of this work as well as future work. Section VIII discusses the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work. We first introduce the Rust programming language (Section II-A), then the `unsafe` Rust (Section II-B). Finally, we discuss dynamic software updating (Section II-C).

A. Rust

Rust [1] is an emerging and rapidly growing programming language. Rust was initially designed by Graydon Hoare of Mozilla Research Institute in 2006, and was first publicly released in 2010 [45]. Rust released its first stable version 1.0 in 2015, and the latest stable version is now 1.62.1 (as of this study).

Rust emphasizes both performance and language safety, by introducing a group of novel language features such as *ownership* [2], *borrow* [3], *reference* [4], and explicit *lifetime* [5]. Rust also provides a set of strict compile-time safety checking rules to guarantee type safety. Furthermore, Rust's ownership-based memory management mechanism ensures that each value in the program has a unique owner, and memory are managed safely, according to the owner's lifetimes.

Due to its safety and efficiency advantages, Rust is gaining more popularity. Rust was rated as the "most popular programming language" on Stack Overflow in 2022 [46]. Rust is also widely used in industry, such as Microsoft [47], Google [48], and even Linux [49]. Specifically, Rust are also used to build online services, such as Mononoke [50], Firecracker [42], Discord [51], and Azure IoT Edge [52].

B. Unsafe Rust

Unsafe Rust is a security loophole for Rust, to bypass Rust's static and dynamic security checking. The unsafe Rust is indispensable for two key reasons: first, Rust's static analysis, like any static program analysis, is conservative and thus may incorrectly reject well-behaved Rust programs. For example, it is `unsafe` to bind an IP address to a socket, even if the IP address is valid. Second, unsafe operations are ubiquitous in low-level system programming (i.e., Rust's primary target scenarios), which demand programming feasibility beyond safe Rust. For example, reading value from a piece of raw memory.

Unsafe Rust [18] has five usage scenarios: 1) *raw pointer dereference*, the raw pointers are allowed to ignore Rust borrowing check rules, are not guaranteed to point to valid memory, and do not implement automatic cleanup; 2) *unsafe function or method invocation*, if a Rust function or method is marked unsafe explicitly, the invocations of such functions are dangerous and should be placed in an `unsafe` block; 3) *mutable static variable accessment*, globally accessible mutable static variables may have data race, so it is `unsafe`; 4) *unsafe trait implementation*, a trait is `unsafe` when at least one of its methods is `unsafe`; 5) *fields of unions accessment*, Rust cannot guarantee the type of data currently stored in the union instance.

The `unsafe` code is widely used in Rust developing. For example, in the Rust community's package repository, 23.6% of the crates contain `unsafe` code [24]. As another example, in Servo [53], a web browser engine from Mozilla, 54% of its code is `unsafe` [23].

Although `unsafe` Rust provides developers with more flexibility, it breaks the security guarantees of Rust and may

leads to security vulnerabilities. Existing research [21] demonstrates that all of the reported memory security vulnerabilities are related to improper use of the `unsafe` code, which can be classified into four categories: (1) automatic memory reclamation errors associated with side effects of Rust OBRM; (2) vulnerabilities caused by the use of unsafe functions and FFIs; (3) vulnerabilities caused by the use of advanced features of Rust, such as trait [54]; and (4) other common memory errors, such as arithmetic overflows and boundary checking issues.

C. Dynamic Software Updating

Dynamic software updating (DSU) represents updating the functionality of a running application dynamically, without stopping or restarting the application. DSU is critical for non-stop systems such as air traffic control, financial transaction, and cloud services, which must provide continuous service but nonetheless be updated to fix bugs and add new features. Additionally, DSU avoids stopping and starting non-critical systems every time a patch must be applied.

The key technique of DSU is state transformation. The idea of state transformation aims to convert the currently running code along with the associated resources, data or state into a new version. Existing studies have proposed many techniques for state transformation in DSU, such as state mapping [55] [56] [57], indirection [58] [59] [60], stack reconstruction [61] [62] [63] [64] , transformation function [55] [65] [66], stack mapping [67] [68], and lazy state transfer [56] [69] [70] .

DSU has three key advantages: *high availability*, *quick vulnerability mitigation* and *user experience improvement* [71]. First, in some critical areas such as defense, medical, industrial control systems and cloud systems, it is critical to provide uninterrupted high availability services [72] [73] [74]. In these systems, any disruptions caused by applying patch are unacceptable. Second, DSU allows quick fixes for vulnerabilities, reducing the time window in which software is vulnerable and making it safe from attack [75] [76]. Third, DSU can reduce the number of system and service restarts to improve the user experience [77].

Due to these advantages, DSU technique has been widely used in recent years, such as Linux kernel [75] [78], Android kernel [76] [77], VMM on cloud [74] and IoT devices [79]. For example, Orthus [74] is a novel DSU system deployed on Alibaba Cloud. It can upgrade a running KVM/QEMU to an updated version with new features or security fixes.

III. Motivation and Threat Model

Before elaborating on the details of RPatch, we first present the research motivation for this study (Section III-A), by introducing a concrete example of how RPatch works. Finally, we give the threat model for this work (Section III-C).

A. Motivation

Cloud service is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing. It is designed

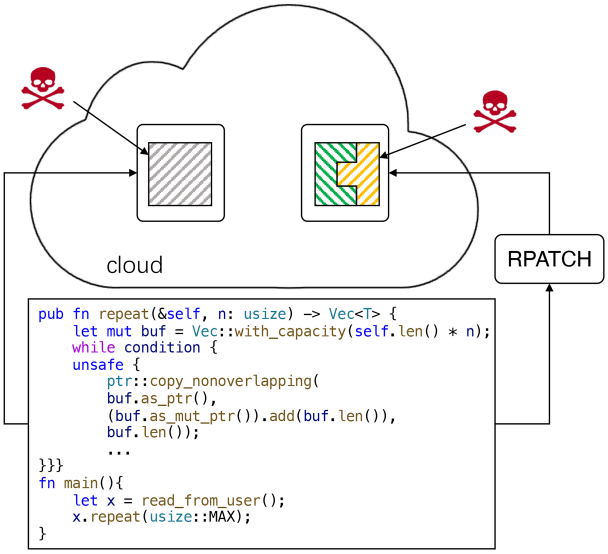


Fig. 1: A technical overview of RPatch

to provide users with uninterrupted access to infrastructure, platforms or software. High availability is the most important goal of cloud service. It requires that the service can be accessed anywhere and at anytime with no perceptible downtime. Due to the high availability, security and performance of cloud service, it has become the preferred choice for current Internet service and the future trend of IT industry development.

However, although cloud service providers avoid downtime to impact user experience, security updates on cloud are frequent and such updates require downtime, which reduces the availability of cloud service. This unexpected downtime causes a significant loss and causes a bad experience for users. For example, according to Gartner [80], the average cost of IT downtime is $5,600 per minute and the average enterprise downtime can reach $300, 000 per hour. Although there have been a lot of studies [55] [75] [64] to mitigate this issue, they still have some limitations: 1) *language feature discrepancy*, 2) *performance issues*, and 3) *incompleteness*, hindering its practical usefulness.

TODO: revise It is challenging to apply dynamic software updating on cloud platform. As shown in Fig. 1, the application being deployed on the cloud platform as a whole, changing any part of the program requires stopping the program from running.

B. Methodology

To address the aforementioned challenges, we propose a novel DSU technology that consists of three key features: 1) delegatecall-proxy pattern, 2) multithread injection, and 3) modular system.

First, the delegatecall-proxy pattern separates insecure functionality from secure functionality by providing proxies. The advantage of the proxy pattern is that the proxy separates the proxy object from the real invoked target object, allowing insecure code to be separated from the main program logic

and to be updated at runtime. As shown in Fig. 1, RPATCH splits the program into two parts, so the vulnerable parts of the program can be replaced.

Second, the multithread injection improves performance for proxy access. Proxy access to relevant functions is performed in multi threads, thus causing less runtime overhead.

Third, we propose a modular system that makes the entire dynamic updating process easy to perform. The system has a modular testing and vulnerability replaying component to automate the process of testing, validating and deploying patches dynamically.

Due to the novel language feature, there are still many challenges faced by Rust, which will be addressed in Section IV.

C. Threat Model

This work focuses on the problem of patching Rust applications dynamically and timely, once application vulnerabilities were detected. Therefore, we make the following assumptions in the threat model for this work.

We assume that the network environment of the cloud service is protected from network attacks by firewalls, so the process of deploying patches is secure. For example, the patches we apply to the cloud will not be replaced with insecure patches by malicious attacks. In addition, the operating system and underlying libraries in the cloud environment are trustworthy and have not been replaced.

We assume that the host 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) [81], Stack Canaries [19], and Address Space Layout Randomization (ASLR) [82]. Furthermore, the Rust compiler has not been compromised by malicious attackers so that the binaries generated from the compiler are trustworthy. It should be noted that although operating systems and compilers security studies are very important, they are independent of and thus orthogonal to the study in this work. Furthermore, these research fields can also benefit from the research progress in this work.

We assume that the safe sub-language of Rust (i.e. not containing 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 buffers access, as every buffer access is checked against the buffer length. Existing Rust security studies demonstrated that *all* reported memory-safety bugs in Rust require `unsafe` code, except for compiler errors [21] [22]. 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, if `unsafe` is used incorrectly, an attacker may exploit this vulnerability to trigger a buffer overflow since the compiler does not perform boundary checks on it.

IV. DESIGN

In this section, we present the design of RPATCH in detail. We first introduce the architecture of RPATCH (Section IV-A), and the patching candidate analysis algorithm (Section IV-B). Then we present the design of delegatecall proxy pattern code generator (Section IV-C), as well as the thread injection (Section IV-D). Finally, we present the vulnerability rectification (Section IV-E), patch testing (Section IV-G), and the human intervention (Section IV-H).

A. The Architecture

We have two principles guiding the design of RPATCH architecture: 1) program and vulnerability neural; and 2) modularity. First, the architecture of RPATCH should be neutral to different language features and vulnerabilities. This principle is important for RPATCH to be applied to any potential Rust programs, as well as different vulnerabilities. Otherwise, the usefulness of RPATCH is significantly reduced, if it can only be applied to a limited set of Rust programs or specific vulnerabilities. Second, the architecture of RPATCH should be modular, thus each module can be easily extended or even replaced separately.

Fig. 2 presents the overall architecture of RPATCH, which consists of several key modules. First, the delegatecall proxy pattern code generator (❶) takes Rust source code as input, and outputs the code with proxy pattern injected and a set of patch candidates. The patch candidates are compiled into loadable modules that are invoked by the main application, and then the entire application is deployed to the cloud platform as an online service. Second, the thread injection module (❷) automatically generates threads for the application that can monitor updates of the loadable modules. Third, the vulnerability rectification module (❸) automatically rectify vulnerabilities at the source code level, and outputs patch source code. Forth, the patch generation module (❹) compiles patch source code into patch binaries. Finally, the patch validator (❺) takes as input the patch binaries and the proxy binaries, to validate whether the vulnerability has been fixed, and whether the program semantics has been modified.

In the following sections, we discuss the design of each module in detail, respectively.

B. Patch Candidate Analysis

In order to separate the source code into proxy module and patch candidates, we first need to identify patch candidates and their call sites. The key idea for this analysis algorithm is to: 1) identify all `unsafe` functions f which can be performed by traversing through the AST; 2) identify all the call sites of the `unsafe` functions using the reversed call graph, which can be built by MIR. The AST is a tree representation of the source program, AST contains the type information about each node of the program, for analysis in later stages. The MIR is a control-flow graph (CFG) representation, in which each block is a sequence of statements. Blocks are connected by directed edges, which represents possible control transfers. MIR contains the necessary semantic information needed by RPATCH such as function calls.

Algorithm 1 takes as input a Rust program P, and generate a program P' with delegatecall-proxy pattern and a set of unsafe

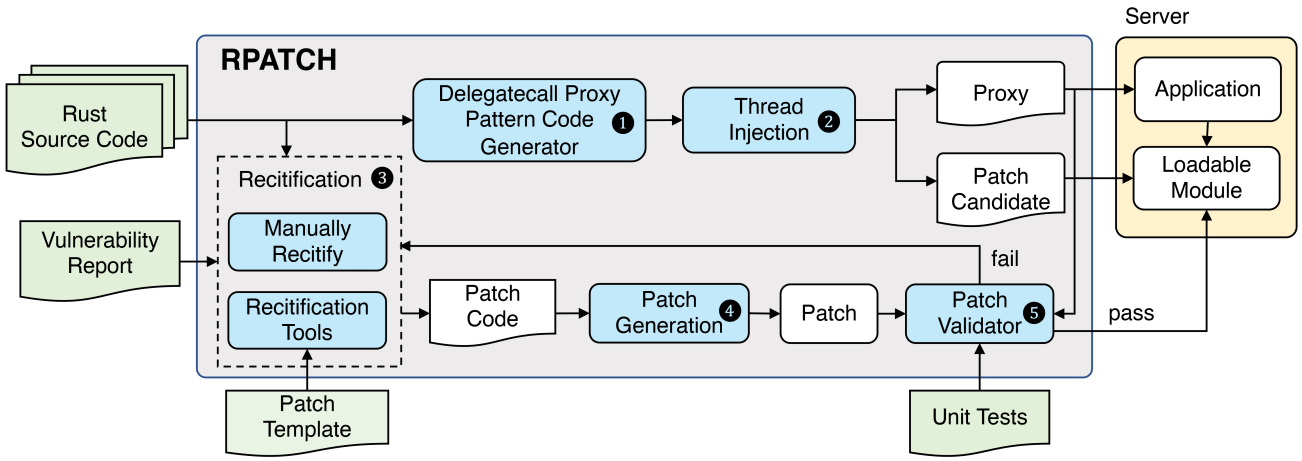


Fig. 2: RPATCH architecture.

functions U. The algorithm has two procedures: 1) GEN-DELEGATECALL() and 2) IDENTIFY-PATCH-CANDIDATE().

The procedure GEN-DELEGATECALL() takes the original Rust program P as input, returns a program P' with delegatecall-proxy pattern and a set of `unsafe` functions U. First, it call IDENTIFY-PATCH-CANDIDATE() to generate a set of `unsafe` functions U and a set of its call sites C. Second, for each function f belonging to the set U, delete the function in the program P'. Third, for each function h belonging to the set C, add a delegatecall in its unsafe call.

The procedure IDENTIFY-PATCH-CANDIDATE() consists of the following key steps. First, the procedure builds an AST A and a MIR M. Second, it builds a call graph G from the MIR M and then reverse it, in order to get the function call sites. Third, the procedure visits each unsafe function f in the AST A, and add f to the set U. Since there is relevant type information on AST, it is easy to determine whether the function is `unsafe`. Then, it looks for each successor h of the function f in the call graph G', and add h to the set C.

This algorithm is efficient, it will build the AST, MIR, and call graph just once, and the traversal of the successor nodes of f in the directed call graph G' is quite fast, as the call graph for the program is of modest size, compared to the size of the AST or MIR.

Algorithm 1 : Delegatecall-proxy pattern generation

Input: P: The Rust program
Output: The original Rust program that removes unsafe functions P'; A set of patch candidates U

procedure GEN-DELEGATECALL(P)
  $P' = P$
  $U, C =$IDENTIFY-PATCH-CANDIDATE(P)
  for each function $f \in U$ **do**
    $P' -= f$
  for each function $h \in C$ **do**
    P'= addDelegatecall(P', h)
  return P', U
procedure IDENTIFY-PATCH-CANDIDATE(P)
  $U, C = \emptyset$
  A, M = buildAstAndMir(P)
  G = buildCallGraph(M)
  G' = Reverse(G)
  for each unsafe function $f \in A$ **do**
    $U \cup = f$
    for each successor h of $f \in G'$ **do**
      $C \cup = h$
  return U, C

C. Delegatecall Proxy Pattern Code Generator

To make the Rust program can be updated at runtime, we use the delegatecall proxy pattern. This is a structural design pattern that provides a substitute or placeholder for another object. Proxies control access to the loadable modules, making them independent of the other parts.

The delegatecall proxy pattern code generator takes as input the Rust marked AST as aforementioned in Section IV-B, generates the AST with proxy pattern injected. The generator split the original Rust source code into two parts, one for proxying calls and one for implementing unsafe functionalities: i) immutable proxy code, which contains function invocations, but does not implement any unsafe functionality; ii) updatable code, which implements actual unsafe functionalities. The main part of the Rust code remains unchanged, and where `unsafe` is used, a proxy pattern is injected to make it possible to mask the use of `unsafe` by calling a proxy. All the `unsafe` code is extracted and becomes a patch candidate, this part of the code may be vulnerable and thus needs to be called by the proxy.

Since the relevant type information and location information are available on the Rust AST, the operation to modify the code is performed by modifying the relevant AST node.

D. Thread Injection

Due to the increased runtime overhead of calling through a proxy, we designed the thread injection module to improve performance.

To inject patching loading thread, the multi-thread generator automatically generates multithreads for the Rust program. The thread is responsible for monitoring the dynamic loadable module, and sharing the corresponding data with the main program. It reads the loadable module at a certain frequency and monitors whether it is updated. If it is updated, the corresponding data will be updated, as well as the path executed by the main program.

It should be noted that there is thread overhead only at application startup. At runtime, the child threads and the main thread can work concurrently.

The two kinds of code generated by the generator are compiled together into a Rust application, where the patch candidates are compiled into loadable module for dynamic loading by the main program that the proxy code is compiled into.

E. Vulnerability Rectification

The vulnerability rectification module takes as input the vulnerability report, the original Rust source code, and user-supplied patch templates, to rectify the vulnerability and generates the patched source code. The patched source code will be further processed by RPATCH's subsequent phases.

To achieve rectify the vulnerabilities automatically, the vulnerability rectification module of RPATCH is designed to leverage the state-of-the-art and open source vulnerability rectification tools, such as Rupair [31]. Furthermore, RPATCH is extensible and can be extended and supported with other up-to-date rectification tools.

For each supported vulnerability type, a patch template is integrated into RPATCH, thus reducing manual effort. This patch template is automatically adapted to the Rust program being patched. For example, for integer overflow vulnerabilities, RPATCH integrates a patch template so that it can be fixed automatically along with the rectification module without manual intervention.

For some vulnerabilities that can not be automatically rectified, we require developers to give patched source code.

F. Patch Generation

The patch generation module takes as input the patch source code, and generates the patch binary. Since Rust does not currently have a stable Application Binary Interface (ABI), the patch generation module has integrated analysis of Rust's name mangling rules and generating patches. The generated patches are able to replace the original loadable module without additional configuration.

G. Patch Validator

In a DSU system, as soon as a patch is deployed, the original application adopts the functionality of the patch. Therefore, we designed a patch validator to test the patch before deployment.

The patch validator is designed with two strategies: security verification and functional verification. For security verification, the patch validator utilizes the state-of-the-art vulnerability detection tools to detect whether vulnerabilities have been successfully fixed and no new vulnerabilities are introduced. For functional verification, the patch validator uses regression testing. Regression testing uses test cases provided by the developer and is designed to verify that the original functionality of the software remains intact after rectification. If either of the two tests fails, the patch will return to the rectification module for re-fix.

H. Human Intervention

Although RPATCH is fully automated, there are still some possibilities that require manual intervention: 1) manual rectification, when there is a vulnerability type that is not yet supported by RPATCH, it needs to be fixed manually; and 2) manual inspection, when the patch validator fails, the cause needs to be checked manually.

If a vulnerability type cannot be rectified automatically, RPATCH notifies the developer of the unsupported vulnerability type. The developer will need to fix it manually. Since RPATCH is extensible and integrates the latest vulnerability fixing tools, the manual intervention required for this part is minimal.

Additionally, if either of the processes in the patch validator report an error, the developer needs to manually analyze the reason for the error and then the patch will return to the rectification module for re-fixing. This part of the manual intervention requires only a quick code review. We believe that even moderately experienced Rust developers can perform these tasks, as no detailed knowledge about the DSU system is required.

V. IMPLEMENTATION

We have implemented a prototype system for RPATCH using Rust. The delegatecall proxy pattern generator and thread injection module are implemented as a custom Rust compiler. It works as an unmodified Rust compiler when compiling dependencies, and injects patch candidate analysis algorithms when compiling target programs. We implemented dynamic loading at runtime based on a third-party library called `libloading`, which provides a series of API to load dynamic libraries and use the functions and static variables they contain. The compiler will ensure that the loaded function will not outlive the library from which it comes, preventing the most common memory-safety issues. We use the rectification tool Rupair [31] to automatically fix buffer overflow vulnerabilities.

VI. EVALUATION

In this section, we present experimental results to evaluate RPATCH, by answering research questions (Section VI-A). We first testify the effectiveness of RPATCH by fixing vulnerabilities in Rust applications and real-world CVEs (Section VI-C). Next, we evaluated the cost of RPATCH (Section VI-D). Then

TABLE I: Experimental results on CVEs and real-world Rust projects bugs.

CVE number	Package	Type	Description	Patched?	Performance(s)
CVE-2017-1000430	base64	OOB	When encoding base64, there is a buffer overflow problem when calculating the buffer size.	✓	0.136
CVE-2018-1000810	std	OOB	When passed a large number this function has an integer overflow which can lead to an out of bounds write.	✓	0.132
CVE-2018-20996	crossbeam	DF	There is a double free because of destructor mishandling.	✓	0.151
CVE-2019-15551	smallvec	DF	There is a double free for certain grow attempts with the current capacity.	✓	0.154
CVE-2019-16140	isahc	UAF	The From implementation for Vec was not properly implemented, returning a vector backed by freed memory.	✓	0.149
CVE-2019-16144	generator	DF	Uninitialized memory is used by Scope, done, and yield_ during API calls.	✓	0.146
CVE-2019-16880	linea	DF	There is double free when the given trait implementation might panic.	✓	0.144
CVE-2019-16881	portaudio-rs	UAF	If the user-provided closure panic, it will lead to use-after-free.	✓	0.143
CVE-2020-25794	sized-chunks	UNINIT	Clone can have a memory-safety issue upon a panic.	✓	0.142
CVE-2020-36210	autorand-rs	UNINIT	Impl Random on arrays, uninitialized memory can be dropped when a panic occurs, leading to memory corruption.	✓	0.144
NA	Servo	UAF	Object dropped earlier and pointer used later.	✓	0.164
NA	Servo	UNINIT	Panic when font face name is not available.	✓	0.150
NA	Tock	UNINIT	Allocator dropping uninitialized memory.	✓	0.146
NA	Redox	UNINIT	Using uninitialized memory.	✓	0.142

we conducted a developer study to evaluate the usefulness of RPATCH (Section VI-F). Finally, we conducted experiments on real-world Rust applications to understand how RPATCH solve real-world Rust DVR problems (Section VI-G).

A. Research Questions

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

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

RQ2: Cost. As RPATCH is designed to rectify the newly detected vulnerabilities timely, does RPATCH incur additional cost to the patched Rust applications?

RQ3: Performance. As RPATCH is designed to solve the Rust DVR problems timely and effectively, what's the performance of RPATCH?

RQ4: Usefulness. As RPATCH is introduced to help Rust developers update Rust applications timely, is it useful to help Rust developers to deploy patches?

B. Experimental Setup

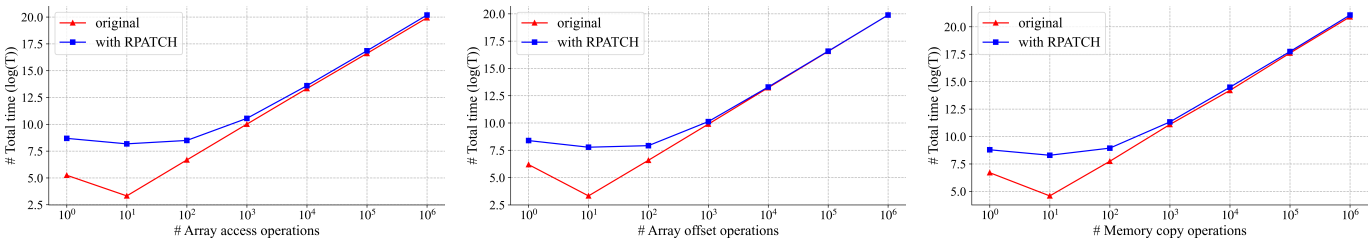
All the experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 16 GB of RAM running Ubuntu 20.04.

C. Effectiveness

To answer **RQ1** by demonstrating the effectiveness of RPATCH, we conducted experiments on Rust applications and real-world CVEs.

Table I shows a micro benchmark of our dataset, consisting 14 buggy Rust programs. These test cases are created in two different ways: 1) we collected 10 typical Rust CVEs with different vulnerability types reported by CVE [83] (the first 10 rows); and 2) we collected 4 bugs in real-world Rust projects (the next 4 rows). These programs are containing four kinds of bugs including out-of-bounds access (OOB), use-after-free (UAF), double-free (DF), and access/free uninitialized memory (UNINIT). Since most program vulnerabilities and CVEs require specific environments, specific inputs, or specific crate versions to trigger, We have to spend considerable manual effort to write a runnable minimum test benchmark for these vulnerabilities.

We apply RPATCH on these benchmarks, the results are shown in Table 1. The column "Patched?" indicates whether RPATCH can patch the vulnerability without terminating the program. In total, RPATCH successfully patch the vulnerabilities in all benchmarks. This experiment demonstrates that RPATCH is effective in deploying patches for Rust applications and real-world CVEs automatically.



(a) Array Access (b) Array Offset (c) Memory Copy

Fig. 3: Execution Time for Programs before and after the RPATCH for three unsafe usage scenarios: array access without boundary checking, raw pointers dereference, and memory copy.

D. Cost

To answer **RQ2**, for each RPATCH-generated program, we measure the runtime overhead RPATCH introduced into it. Since the number of executions of `unsafe` functionalities by real-world applications is immeasurable, we testified micro benchmarks for three usage scenarios where people use `unsafe` to improve performance [22]: 1) traversing arrays with `unsafe` memory access (`slice::get_unchecked()`) without boundary checking; 2) traversing an array by pointer computing (`ptr::offset()`) and dereferencing, and 3) `unsafe` memory copy (`ptr::copy_nonoverlapping()`).

We compare the execution time for each DSU program with its corresponding original program, which are showed in Fig. 3.

As Fig. 3 shows, we executed two different versions for each benchmark: 1) the original one (red); 2) the one with RPATCH (blue). In each sub-figure, the x-axis stands for the number of operations performed, from 1 to 10^6; and the y-axis gives the average running time for the corresponding operations, in microseconds. Furthermore, to make the difference between running time clearer, we have normalized the average running time by $y = \log(T)$, where T is the original absolute running time.

We observed that, for the three scenarios, the runtime overhead RPATCH introduced was less than 140 milliseconds on average. This overhead is practical and acceptable, because the main time overhead occurs when the application is first started, and when the application is patched.

This experiment shows that the extra cost RPATCH introduced is low and insignificant.

E. Performance

To understand RPATCH's performance, we conducted experiments to measure the time spent by RPATCH in converting each Rust program. In Table I, we presented the time to modify the Rust source code to inject the proxy pattern.

As shown in the last column of Table I, the maximum analysis time for RPATCH is 0.164s, which depends on the complexity of the program. This experiment demonstrates that RPATCH is efficient to solve the Rust DVR problem.

F. Usefulness

Developer Background. To quantify the manual effort needed to patch programs and evaluate the usefulness of RPATCH, we conducted a developer study. We hired five Rust developers to conduct this study, 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 very familiar with developing DSU systems. None of the developers had ever developed a DSU system before. Therefore, we can quantify the effort required for a Rust developer to learn and develop a DSU system.

Methodology. Throughout our study, we asked the developers to perform two tasks: (1) manually converting three vulnerable programs to DSU programs and with RPATCH; (2) manually deploying patches for running Rust programs and with RPATCH. The two tasks cover different scenarios, where RPATCH can be useful to a developer. The first task measured the manual effort required to convert a Rust program to a dynamically updatable program. The second task demonstrates the ease of use of RPATCH in automating the conversion, validating and deploying of patches. For all tasks, we measured the time required by the developer to perform the task, as shown in Table II.

We then performed both a manual code review and a crosscheck with RPATCH to analyze mistakes made by the developers. The experimental results show that it takes a lot of time and effort to perform these tasks manually, whereas RPATCH offers simple, efficient and user-friendly operations. The time measurements show that the developers with no prior experience with RPATCH, were able to complete complex tasks within minutes using RPATCH.

Converting to a DSU program. We collected 3 typical CVEs and wrote three vulnerable programs accordingly. To provide a representative set of programs, we wrote the three programs with different complexity (in terms of lines of code). 1) CVE-2017-1000430 [84] (356 lines of code), 2) CVE-2019-16140 [85] (212 lines of code), and 3) CVE-2019-16881 [86] (534 lines of code). We provided developers with a short description of the delegatecall-proxy pattern and asked them to convert the given source code of vulnerable programs into two pieces of code: a piece of delegatecall proxy code and a piece of updatable candidate code. The developers required an average

of 125 minutes to convert the source code into an delegatecall-proxy pattern code.

TABLE II: Average time (Minutes) for each task, without and with RPATCH

CVE	Convert to DSU		Patch deployment	
	Manual	RPATCH	Manual	RPATCH
CVE-2017-1000430	104	6	40	2
CVE-2019-16140	120	5	32	1
CVE-2019-16881	150	4	55	2

Next, we asked the developer to convert the source code of the Rust program to a DSU program using RPATCH. Since RPATCH does not require any prior knowledge of delegatecall-proxy pattern, developers were able to deploy a correct DSU program in a maximum of 6 minutes.

Dynamic deployment of patches. We provided developers with the patch source code of the programs in the first task and asked them to deploy it on the corresponding running program. The developers needed to manually check whether the patch is correct and whether the functionality of the source program has changed, and then compile it into a dynamic loadable module and deployed it on the cloud platform.

All developers manually and correctly deployed the patches for all three programs, which shows their expertise in Rust development. However, compared to the long time it took for developers to patch manually (42 minutes on average), RPATCH automated the patching process and was able to help developers deploy patches for all three Rust programs in less than 2 minutes.

Summary. Our study provides confirmation that RPATCH offers a high degree of automation, efficiency, and usability thereby freeing developers from manual and error-prone tasks.

G. Real-world Applications

To understand how RPATCH solve real-world Rust DVR problems, we conducted experiments on real-world applications of two fields: database and data processing. Both applications maintain potentially large amounts of in-memory data that are either lost or expensive to restore after an update. We took the fault injection [87] approach, which is a technique of software testing by introducing faults to test code paths.

1) **Database:** RisingWave [88] is a cloud-native streaming database that uses SQL as the interface language. It is designed to reduce the complexity and cost of building real-time applications.

We inserted a set of `unsafe` code into the original source code of RisingWave, making it vulnerable to attacks. We modified RisingWave by introducing the vulnerable code and recompiled it to obtain a unsafe version `RisingWave.unsafe`. Then, we applied RPATCH on the unsafe version `RisingWave.unsafe` to modify it to a updatable version `RisingWave.update`. Next, we started each of the two versions of RisingWave server and used a PostgreSQL terminal (psql) to connect to them. We used scripts to insert and query data in bulk, recording the time spent respectively. We then wrote patch code for the inserted vulnerable code, entered the patch into RPATCH, and deployed the patch in the running environment of the application.

Our experimental results show that the application utilizing the `RisingWave.update` can be dynamically updated at runtime, while the other cannot. The time for RPATCH to modify `RisingWave.unsafe` to `RisingWave.update` is 0.557s, which is trivial in application compilation (270 seconds). The runtime overhead RPATCH introduced is less than 20 milliseconds.

2) **Data Processing:** Polars [89] is a DataFrame library for Rust. It is based on Apache Arrow's memory model and designed for parallelization of queries on DataFrames.

We inserted a set of `unsafe` code to Polars and compiled with and without RPATCH to two versions `Polars.unsafe` and `Polars.update`, respectively. Then, we developed two identical applications that utilize the two versions of the Polars library for querying data. We then wrote patch code for the inserted vulnerable code, entered the patch into RPATCH, and deployed the patch in the running environment of the application.

Experimental results show that RPATCH can deploy the patch successfully, and the runtime overhead RPATCH introduced is less than 5 milliseconds.

The experimental results demonstrated that RPATCH is useful to patch real-world large Rust applications, and the extra cost RPATCH introduced is insignificant.

VII. Discussion

In this section, we discuss some possible enhancements 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.

A. Binary Level

RPATCH is designed to support source-level patching. Binary patching is useful when the source code is absent. This is difficult work because Rust doesn't have a stable application binary interface. Even with identically defined structs the layout may be different. As Rust compiler uses LLVM [90] [91] to generate code, it's feasible to make use of the LLVM level instrumentation technology (Instrew [92], Dbill [93], etc.), to perform binary-level patching. We leave it for future work.

B. Consistency Issues

Some complex patches may change the semantics of the target function, which may affect other functions that are not patched. For example, a patch may change the order in which multiple functions get locks at the same time, or some patches may change the global data used by multiple functions. Currently, RPATCH cannot handle these cases.

Existing empirical studies suggest this is rare. For example, the use of global variables occurs in only 3.7% of `unsafe` unsages [23]. One way to address this problem is to construct a consistency model and safely choose patch tasks, identify

and patch all relevant functions. We leave this consideration for future work.

VIII. Related Work

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

A. Rust security

Unsafe Mechanism. Evans et al. [23] conducted a large-scale empirical study on the use of the unsafe mechanism in real-world Rust library functions and applications. Qin et al. [22] conducted an empirical study of 850 examples of unsafe code in real Rust programs to summarize common unsafe operations and the purpose of using unsafe. Astrauskas et al. [24] studied empirically how unsafe code is used in practice by analyzing a large corpus of Rust projects to assess the validity of the Rust hypothesis and to classify the purpose of unsafe code. These studies demonstrate the necessity and widespread use of unsafe code.

Security Vulnerabilities. Rust security vulnerabilities can be divided into two categories: memory security vulnerabilities and concurrency security vulnerabilities. Xu et al. [21] conducted an empirical analysis and study of 186 Rust memory security vulnerabilities that have been reported as of December 31, 2020. The results of the study show that all the reported memory security vulnerabilities are related to the improper use of unsafe mechanism. Yu et al. [94] studied three software systems developed based on Rust, and empirically investigated 18 of them for concurrent security vulnerabilities. Qin et al. [22] conducted an empirical study of 70 memory security vulnerabilities and 100 concurrency security vulnerabilities in real Rust applications to expose the causes of security vulnerabilities.

Vulnerability Detection and Rectification. Existing studies usually use static or dynamic analysis tools for vulnerability detection using the three intermediate representations provided by the Rust compiler: AST, HIR, and MIR. Huang et al. [25] constructed a tool UnsafeFencer, to help developers detect unsafe raw pointer dereference. Cui et al. [26] proposed SafeDrop, which focuses on deallocation of heap memory and detects memory corruption by performing alias analysis and taint analysis on Rust MIR. Switzer et al. [27] proposed a type obfuscation vulnerability detection algorithm by performing static data flow analysis on MIR and then inserting runtime inspection code on LLVM IR. MirChecker [28] is a vulnerability detection framework that can detect both runtime crashes and memory security vulnerabilities caused by dangling pointers. Rudra [29] detects memory vulnerabilities by running dataflow analysis and send/sync difference checking algorithms on the Rust ecosystem. Stuck-me-not [30] detects common deadlock security vulnerabilities in blockchain systems developed on Rust by performing data flow analysis on MIR. Rupair [31] is currently the only work that involves automatic repair of Rust vulnerabilities. It detects buffer overflow vulnerabilities caused by integer arithmetic overflows by performing data flow analysis on AST and MIR.

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.

B. Dynamic Software Updating

KSHOT [78] is used to patch Linux kernel security vulnerabilities in real time, it uses System Management Mode (SMM) to store runtime state and Intel Software Guard eXtensions (SGX) as a trusted environment for patch preparation. Zhang et al. [74] proposed a new method, VMM live upgrade, that can quickly upgrade the entire VMM without disrupting the guest VMs. They built Orthus, which features three key techniques: dual KVM, VM grafting, and device handover to effectively ”cut and paste” VMs from their running VMM to the upgraded VMM. Xu et al. [77] developed an automatic hotpatch generation tool, Vulmet, which produces semantic preserving hot patches by learning from the official patches. The key idea of Vulmet is to use the weakest precondition reasoning to transform the changes made by the official patches into the hot patch constraints. Romme et al. [95] proposed Multiverse, a C and GNU C compiler extension that provides runtime support for efficient function-level binary patches in both user and kernel modes. LEE et al. [96] dynamically patch Android applications by using the Appwrapping technique (also known as bytecode rewriting), a technique that uses Java reflection to dynamically invoke security functions by inserting them into the Android application to execute code. Mvedsua [97] is a novel DSU system that employs Multiple Version Execution (MVE) to provide a solution that both masks update pauses and tolerates a variety of failed updates. It applies the update to the previous copy while the original system continues to run and keeps the status of the two versions synchronized. If the new version shows no problem after the warm-up period, the operator can make it permanent and discard the original version.

However, all these studies cannot be applied to the Rust language, due to the dramatic feature difference between Rust and other languages.

IX. Conclusion

In this work, we present a framework RPatch, to automatically detect and rectify vulnerabilities in the native code of Rust. We implemented a prototype for RPatch and conducted extensive experiments with it. Experimental results demonstrated the effectiveness, efficiency, and usefulness of RPatch. The work in this paper is a first step towards timely and effectively patch Rust applications. 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] "The Rust Programming Language," https://doc.rust-lang.org/stable/book/.

[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," https://doc.rust-lang.org/rust-by-example/scope/lifetime.html.

[6] "Tock Embedded Operating System," https://www.tockos.org/.

[7] S. Lankes, J. Breitbart, and S. Pickartz, "Exploring Rust for Unikernel Development," in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. Huntsville ON Canada: ACM, Oct. 2019, pp. 8–15.

[8] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The Case for Writing a Kernel in Rust," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*. Mumbai India: ACM, Sep. 2017, pp. 1–7.

[9] A. Light, "Reenix: Implementing a Unix-Like Operating System in Rust," p. 28.

[10] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with Intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Lausanne Switzerland: ACM, Mar. 2020, pp. 143–156.

[11] "Servo," https://servo.org/.

[12] "TFS," https://github.com/redox-os/tfs.

[13] "TTstack," https://github.com/rustcc/TTstack.

[14] "Smoltcp: A smol tcp/ip stack," https://github.com/smoltcp-rs/smoltcp.

[15] "Tokio - An asynchronous Rust runtime," https://tokio.rs/.

[16] "TiKV: Distributed transactional key-value database, originally created to complement TiDB," https://github.com/tikv/tikv.

[17] "Parity-ethereum: The fast, light, and robust client for Ethereum-like networks," https://github.com/openethereum/parity-ethereum.

[18] "Unsafe Rust," https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[19] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard," p. 11.

[20] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Z. Huang, Y. J. Wang, and J. Liu, "Detecting Unsafe Raw Pointer Dereferencing Behavior in Rust," *IEICE Transactions on Information and Systems*, vol. E101.D, no. 8, pp. 2150–2153, Aug. 2018.

[26] M. Cui, C. Chen, H. Xu, and Y. Zhou, "SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-Flow Analysis," Apr. 2021.

[27] J. F. Switzer, "Preventing IPC-facilitated type confusion in Rust," p. 60, 2020.

[28] 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.

[29] 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.

[30] P. Ning and B. Qin, "Stuck-me-not: A deadlock detector on blockchain software in Rust," *Procedia Computer Science*, vol. 177, pp. 599–604, 2020.

[31] 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.

[32] 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.

[33] E. E. Rivera, "Preserving Memory Safety in Safe Rust during Interactions with Unsafe Languages," p. 66.

[34] W. Ouyang and B. Hua, "RusBox: Towards Efficient and Adaptive Sandboxing for Rust," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Wuhan, China: IEEE, Oct. 2021, pp. 1–2.

[35] D. Dominik, "Visualization of Lifetime Constraints in Rust," p. 35.

[36] D. Blaser, "Simple Explanation of Complex Lifetime Errors in Rust," p. 63.

[37] P. Lindgren, N. Fitinghoff, and J. Aparicio, "Cargo-call-stack Static Call-stack Analysis for Rust," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Helsinki, Finland: IEEE, Jul. 2019, pp. 1169–1176.

[38] 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.

[39] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "RustBelt meets relaxed memory," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, Jan. 2020.

[40] A. Weiss, D. Patterson, and A. Ahmed, "Rust Distilled: An Expressive Tower of Languages," Aug. 2018.

[41] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for Rust," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–32, Jan. 2020.

[42] "Firecracker," https://firecracker-microvm.github.io/.

[43] "Dynamic software updating - Wikipedia," https://en.wikipedia.org/wiki/Dynamic_software_updating.

[44] M. Payer, B. Bluntschli, and T. R. Gross, "DynSec: On-the-fly Code Rewritingg andRepair," in *5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13)*, 2013.

[45] G. Hoare, "Project Servo," http://venge.net/graydon/talks/intro-talk-2.pdf, 2010.

[46] "Stack Overflow Developer Survey 2022," https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022.

[47] C. Catalin, "Microsoft to explore using Rust," https://www.zdnet.com/article/microsoft-to-explore-using-rust/, 2019.

[48] "Rust in the Android platform," https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[49] "Rust in the Linux kernel," https://security.googleblog.com/2021/04/rust-in-linux-kernel.html.

[50] "Mononoke," https://github.com/facebookexperimental/eden.

[51] "Why Discord is switching from Go to Rust," https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

[52] "IotEdge," https://github.com/Azure/iotedge/tree/main/edgelet.

[53] 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.

[54] "Trait," https://doc.rust-lang.org/book/ch10-02-traits.html.

[55] S. An, X. Ma, C. Cao, P. Yu, and C. Xu, "An Event-Based Formal Framework for Dynamic Software Update," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. Vancouver, BC, Canada: IEEE, Aug. 2015, pp. 173–182.

[56] A. R. Gregersen and B. N. Jørgensen, "Dynamic update of Java applications—balancing change flexibility vs programming transparency," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 81–112, 2009.

[57] F. Boyer, O. Gruber, and D. Pous, "A robust reconfiguration protocol for the dynamic update of component-based software systems," *Software: Practice and Experience*, vol. 47, no. 11, pp. 1729–1753, 2017.

[58] 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.

[59] W. Cazzola and M. Jalili, "Dodging Unsafe Update Points in Java Dynamic Software Updating Systems," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, 2016, pp. 332–341.

[60] W. Tang and M. Zhang, "PyReload: Dynamic Updating of Python Programs by Reloading," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. Nara, Japan: IEEE, Dec. 2018, pp. 229–238.

[61] 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.

[62] D. Zou, H. Wang, and H. Jin, "StrongUpdate: An Immediate Dynamic Software Update System for Multi-threaded Applications," in *Human Centered Computing*, ser. Lecture Notes in Computer Science, Q. Zu, B. Hu, N. Gu, and S. Seng, Eds. Cham: Springer International Publishing, 2015, pp. 365–379.

[63] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of Java applications," *Information and Software Technology*, vol. 56, no. 9, pp. 1086–1098, Sep. 2014.

[64] Z. Chen and W. Qiang, "ISLUS: An Immediate and Safe Live Update System for C Program," in *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, Jun. 2017, pp. 267–274.

[65] A. R. Gregersen, B. N. Jørgensen, Hadaytullah, and K. Koskimies, "Javeleon: An Integrated Platform for Dynamic Software Updating and Its Application in Self-*Systems," in *2012 Spring Congress on Engineering and Technology*, 2012, pp. 1–9.

[66] 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.

[67] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, General-purpose Dynamic Software Updating for C," p. 16.

[68] F. Chen, W. Qiang, H. Jin, D. Zou, and D. Wang, "Multi-version Execution for the Dynamic Updating of Cloud Applications," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, Jul. 2015, pp. 185–190.

[69] E. Wernli, D. Gurtner, and O. Nierstrasz, "Using first-class contexts to realize dynamic software updates," in *Proceedings of the International Workshop on Smalltalk Technologies*, ser. IWST '11. New York, NY, USA: Association for Computing Machinery, Aug. 2011, pp. 1–11.

[70] A. R. Gregersen, D. Simon, and B. N. Jørgensen, "Towards a dynamic-update-enabled JVM," in *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, ser. RAM-SE '09. New York, NY, USA: Association for Computing Machinery, Jul. 2009, pp. 1–7.

[71] C. Islam, V. Prokhorenko, and M. A. Babar, "Runtime Software Patching: Taxonomy, Survey and Future Directions," Mar. 2022.

[72] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[73] Z. Zhao, T. Gu, X. Ma, C. Xu, and J. Lü, "CURE: Automated Patch Generation for Dynamic Software Update," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 2016, pp. 249–256.

[74] 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.

[75] 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.

[76] 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.

[77] Z. Xu, Y. Zhang, L. Zheng, and L. Xia, "Automatic Hot Patch Generation for Android Kernels," p. 19.

[78] 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.

[79] M. Weißbach, N. Taing, M. Wutzler, T. Springer, A. Schill, and S. Clarke, "Decentralized coordination of dynamic software updates in the Internet of Things," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 2016, pp. 171–176.

[80] "Ensure Cost Balances With Risk in High-Availability Data Centers," https://www.gartner.com/en/documents/3906266.

[81] alvinashcraft, "Data Execution Prevention," https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[82] "PaX (http://pageexec.virtualave.net)," p. 37.

[83] "Rust CVE," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust.

[84] "CVE-2017-1000430," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000430.

[85] "CVE-2019-16140," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16140.

[86] "CVE-2019-16881," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16881.

[87] "Fault injection - Wikipedia," https://en.wikipedia.org/wiki/Fault_injection.

[88] "RisingWave," www.risingwave.dev.

[89] "Pola-rs/polars: Fast multi-threaded DataFrame library in Rust — Python — Node.js," https://github.com/pola-rs/polars.

[90] 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.

[91] ——, "The LLVM Compiler Framework and Infrastructure Tutorial," in *Languages and Compilers for High Performance Computing*, ser. Lecture Notes in Computer Science, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds. Berlin, Heidelberg: Springer, 2005, pp. 15–16.

[92] 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.

[93] 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.

[94] Z. Yu, L. Song, and Y. Zhang, "Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software," Feb. 2019.

[95] 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: Association for Computing Machinery, Mar. 2019, pp. 1–13.

[96] S.-H. Lee, S.-H. Kim, J. Y. Hwang, S. Kim, and S.-H. Jin, "Is Your Android App Insecure? Patching Security Functions With Dynamic Policy Based on a Java Reflection Technique," *IEEE Access*, vol. 8, pp. 83 248–83 264, 2020.

[97] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 573–585.