# 2021 年全国大学生信息安全竞赛

# 作品报告

作品名称：C2SafeRust 半自动化转换工具

电子邮箱：3045465496@qq.com

提交日期：2021 年 6 月 6 日

# 填写说明

1. 所有参赛项目必须为一个基本完整的设计。作品报告书旨在能够清晰准确地阐述（或图示）该参赛队的参赛项目（或方案）。

2. 作品报告采用A4纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5倍行距。

3. 作品报告中各项目说明文字部分仅供参考，作品报告书撰写完毕后，请删除所有说明文字。(本页不删除)

4. 作品报告模板里已经列的内容仅供参考，作者可以在此基础上增加内容或对文档结构进行微调。

5. 为保证网评的公平、公正，作品报告中应避免出现作者所在学校、院系和指导教师等泄露身份的信息。

# 目　　录

# 摘要

Rust是由Mozilla在2014年发布的一种多范式系统编程语言,因其能够在保持高性能的同时提供更好的内存安全性，成为了越来越多大型项目的首选语言。然而因为Rust的一些安全特性以及语法上的限制，它的学习曲线很不友好，从入门到熟练掌握可能要花上几个月的时间。对于那些想将C语言的工程转为Rust的用户，尽管已经存在c2rust等工具，但因为其仅仅转换为了unsafe rust，后续仍需要大量手工修改才能得到safe rust。

为此，我们设计了一个C2SafeRust的半自动化转换器，它通过在LLVM IR的层面上分析C语言代码,获取包含了诸如所有权、生命周期等的Rust安全特性的策略(policy)，然后通过在c2rust的转换工具得到的unsafe Rust的基础上，施加前一步得到的策略，从而生成safe Rust。若原本的C代码没有内存漏洞，则能顺利通过编译；否则将原本C语言中隐藏的内存安全漏洞，通过Rust编译器暴露出来。

我们实现这个工具的目的在于减轻从unsafe Rust到safe Rust的转换过程中所需要的负担，尽可能减少手动修改的工作量。

# 第一章 作品概述

## 1. 背景分析

Rust是一门系统编程语言，专注于安全，尤其是并发安全，支持函数式和命令式以及泛型等编程范式的多范式语言。Rust语言语法与C++语言相似，执行速度快且内存利用率极高，由于没有运行时和垃圾回收，它能够胜任对性能要求特别高的服务，可以在嵌入式设备上运行，还能轻松和其他语言集成。

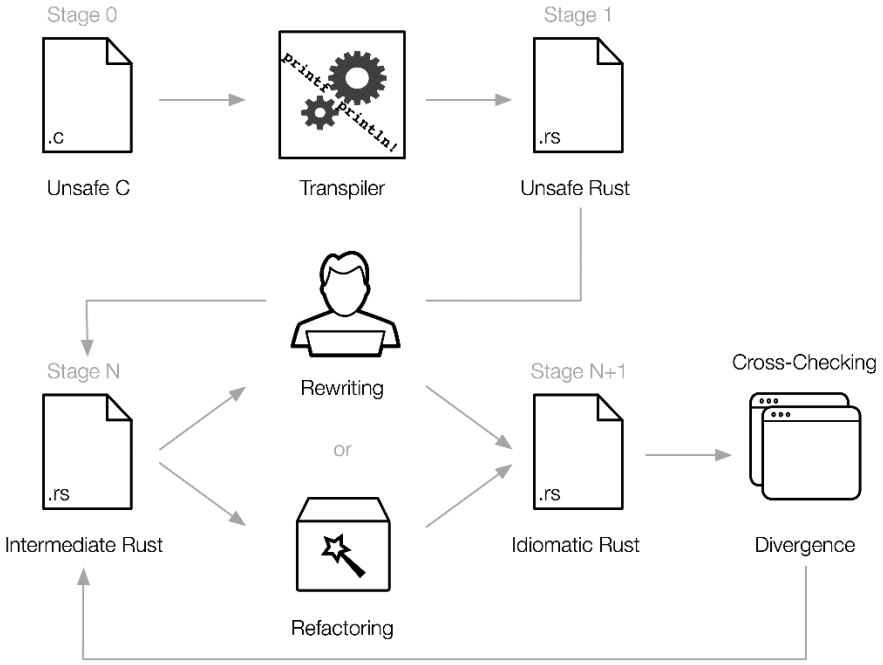
Rust语言最早在2015年上线，已连续五年在StackOverflow开发者调查中荣获“最受喜爱编程语言”。Rust语言同样具有广阔的应用前景和良好的社区生态，目前微软，脸书，谷歌，阿里等公司都在积极拥抱Rust语言，上线相关项目如使用Rust语言重写部分AOSP代码等。

Rust语言在安全性方面最大的特点是通过编译时的细粒度检查将潜在的安全问题扼杀。具体来说Rust语言与C语言有着本质上的不同，Rust语言更像高级语言，不能直接操作指针与内存，只能对分配好的一个个对象进行操作。另一方面，Rust语言引入了生存周期，所有权，引用的概念，在一些情况下程序员需要手动标明这些信息，通过这些附加的属性使得编译器可以做更细粒度的检查，一旦发现可能的安全隐患则无法通过编译。举例来说，Rust语言的引用概念规定：一个变量只能有一个可变引用或者多个不可变引用，从而解决了并发时的同步问题。

逐步使用Rust语言重写过去的C语言代码是一个趋势，可以在性能不变的前提下提升安全性，但是Rust语言与C语言有很大不同，因此我们尝试开发一个C2SafeRust的工具，辅助Rust重写工作。

## 2. 相关工作

c2rust是一个开源项目，通过C语言程序编译出的抽象语法树从语法层面生成对应的Rust语言代码，但是从语法层面实现语言转译有它的问题，例如C中的指针正常是不应该存在于Rust代码之中的。因此如下图所示，c2rust转换后的unsafe-rust代码仍然需要人工编辑，才能修改为可用的Rust安全代码。




**图表** 1c2rust **流程图**

Rust语言也考虑到了从C代码迁移过来的难度，因此设计了unsafe-rust，允许程序员使用一些无法通过安全检查的代码进行编译，其中就兼容了C的指针操作，也就是说c2rust工具生成的是unsafe的Rust代码，编译器并不会对其进行严格的安全性检查，经过我们的测试，转移后的代码执行结果与C语言基本一致，无法利用Rust语言的安全特性。另一方面，直接由c2rust工具转换生成的代码可读性很差，为了实现C语言的同等效果，生成的Rust语言中大量地调用了unsafe-libc中的声明和函数，甚至要将C语言中调用的宏一层层展开。因此c2rust工具目前可用，但是实用性并不高，仅可作为一个小型项目的辅助工具，并且需要叫大量的人工编辑。

## 3. 特色描述

根据我们的尝试，想要建立一个完全自动化的c2rust转换工具几乎不可能，如果可以从C语言中静态分析出足够Rust语言的额外信息，那么便完全可以在C语言层面上发现安全问题，这显然是目前无法做到的。

因此我们致力于完成一个Rust语言重构C语言项目的辅助工具，针对一系列特定安全问题进行自动化或者半自动化的代码生成，而对于其他没有指定要求的目前认为“安全”的代码不做额外处理。这样可以一定程度上减轻重构代码的任务量，尤其是当针对可能发生的，特定的安全问题，可以对涉及到的部分代码引入Rust语言的编译

检查机制，保证其安全性。

## 4. 应用前景分析

使用安全的Rust语言重构C语言项目目前来看是一个趋势，但是基于上述分析这将不亚于构建项目时的工程量，这是Rust语言目前面临的一个难题与挑战。因此，如果可以实现我们的C2SafeRust，那么可以针对特定类型的安全问题生成特定的Rust代码对其进行保护，就可以减少一部分重构的工程量。例如C语言项目中广泛存在的内存泄漏问题，就可以通过我们的C2SafeRust生成unsafe-rust与Rust夹杂的代码，对堆上分配的内存进行特定的保护。在未来可能会有一系列类似于C2SafeRust的工具来辅助小型项目，甚至于是大型项目的Rust重构工作。

# 第二章 作品设计与实现

## 1. 系统方案

正如前一章节所述，现有的c2rust工具只能将C语言转换到unsafe Rust，并没有用到任何Rust安全特性，没有实际应用价值，因此后续还是需要大量人工手动修改，将unsafe Rust转换为safe Rust。

为了减轻人工工作量，我们将整个转换过程（C到safe Rust）分为如下三个步骤：

1. 利用LLVM，将C语言源代码转换为对应的LLVM IR语言，然后通过自动化脚本分析IR代码，获得能够描述Rust中的所有权、生命周期等安全特性的策略（policy）
2. 利用c2rust工具，将C语言源代码转换为unsafe Rust。
3. 通过半自动化脚本（需要一些人工识别），将第一步获得policy实施到unsafe Rust上，从而获得了safe Rust

因为时间原因，我们选择将一个存在简单的UAF（use after free）漏洞的C源代码，转换为safe Rust作为结果展示，具体可参见下一章“作品测试与分析”。完整的项目可以在https://github.com/RainWang6188/C2SafeRust.git上查看

## 2. 实现原理

我们将从转换过程的三个步骤分别描述实现原理

### a） 生成policy

- 首先通过如下命令，将C源代码转为以.ll后缀的LLVM IR代码。

```
clang -emit-llvm -S -fno-discard-value-names <source.c> -o <target.ll>
```

- 然后，通过python脚本，利用LLVM IR的SSA特性（static single assignment），将IR代码的语句划分为如下7种情况，并建立依赖树（多叉树结构，子节点的值从父节点获取，即子节点依赖于父节点）

1）alloca

```
%retval = alloca i32, align 4
```

LLVM IR种alloca语句用于声明变量，因此当判断该行为alloca类型，

即将赋值号左边的操作数（此处为%retval）创建为依赖树的根节点

2）@malloc

```
%call = call noalias i8* @malloc(i64 16) #3
```

@malloc类型是函数调用libc中的malloc()，此时因为malloc的参数是一个常量，因此我们也将左操作数创建为依赖树根节点

3）bitcast

```
%0 = bitcast i8* %call to %struct.A*
```

LLVM中的bitcast相当于强制类型转换，如上图中将%call变量从i8*转换为了%struct.A*，并赋值给了变量%0。因此，我们将左操作数作为右操作数的子节点，加入依赖树中，并设置对应的类型。此例中，（%0， %struct.A*）被设置为了（%call，i8*）的子节点。

4）load

```
%1 = load %struct.A*, %struct.A** %a, align 8
```

LLVM IR中向变量赋值均通过load/store操作实现。Load操作左操作数获取右操作数中的值，因此，我们将左操作数作为右操作数的子节点，加入依赖树中，并设置对应类型。此例中，%1被设置为了%a的子节点，并赋予类型%struct.A*。

5）store

store操作要分为两种情况

- 有两个变量操作数

```
store %struct.A* %0, %struct.A** %a, align 8
```

这里，将左操作数%0的值存入了右操作数%a中，因此，%a依赖于%0，我们将右操作数作为左操作数的子节点放入依赖树中,并赋予对应的类型。

- 仅有一个变量操作数，另一个是常量

```
store i32 0, i32* %retval, align 4
```

此时左操作数为常量，因此这种情况仅会改变右操作数的值或类型，我们仅需要在依赖树中找到该节点，并修改类型即可。

6）getelementptr(GEP)

```
%B1 = getelementptr inbounds %struct.B, %struct.B* %4, i32 0, i32 0
```

GEP 指令用于获取组合变量中某一成员变量的地址。简单来说，上例中的左操作数%B1 依赖于右操作数%4。因此，我们将左操作数作为右操作数的子节点，类型设置为 elementptr，后续往往会因为 store 指令修改类型。

7）其他情况

其他类型的指令不会对依赖树产生影响，因此不做考虑。

依赖树显式地描述了变量的生命周期：因为子节点依赖于父节点，因此，在Rust中作为引用时的子节点的生命周期必须不长于父节点的生命周期。

**b） 获取 unsafe Rust**

接下来，我们就可以通过github上的c2rust工具，将C语言源代码转换为unsafe Rust代码。其实现原理是从LLVM AST层面分析C代码，生成Rust语法格式的不安全Rust。所有的变量类型均与C相同，因此完全不包含引用、所有权、生命周期等概念（如C中的指针经过c2rust转换后成为了不安全的裸指针），仍存在内存安全问题。

**c） 实施 policy 获得 safe Rust**

有了第一步得到的 policy 与第二步的 unsafe Rust，接下来，我们就可以生成 safe Rust 了。这一步仍需要一定的人工分析。

在 Rust 中，基本类型（如数字类型，布尔类型等）均是具有 Copy 属性（Copy Trait）的，因此在变量绑定、函数参数传递、函数返回值传递等场景下，它都是 copy 语义，会自动拷贝一个新的变量，而不会发生所有权转移。所以，我们没有必要考虑这些类型的变量，分析它们的所有权、生命周期是没有意义的。在简单情况下，我们只需要考虑指针类型，而这也是 C 语言中最容易引起内存安全漏洞的类型。

对于 C 语言中的指针类型，我们选择将其转换为 Rust 中的智能指针 Box<T> 类型。Box<T>类型的变量具有如下特性：将数据存储在堆上；且不具有 Copy 属性，能够传递所有权；具有 Deref 属性，可以自定义解引用运算符的行为，方便地访问复杂组合对象的成员。这些特性完美契合了我们的需要，所以我们在生成 safe Rust 的过程中，首先通过分析 policy 中的依赖树，找到所有指针类型变量，然后通过半自动化脚本，将 unsafe Rust 中的这些变量转换为Box<T>类型：将malloc()从内存中分配空间转变为Box<T>的new;而free()则改为通过将对象放入一个空的 free_box<T>()函数，将所有权释放。这样修改后，如果在 free_box(a)后仍引用 a 变量，Rust 编译器就会报错。

## 3. 软件流程

(a)通过c2rust工具生成初步的unsafe-rust代码

(b)对LLVM编译得到的中间码进行分析，生成变量依赖关系树

(c)对得到的依赖树和代码进行一定程度的人工分析，生成修改policy

(d)依照policy执行python脚本得到修改后的Rust代码

## 4. 功能

通过C2SafeRust可以半自动化地针对一些安全问题生成特定的Rust代码，利用Rust安全特性解决这些安全问题，C2SafeRust作为Rust项目重构的辅助工具可以减少重构的任务量。

# 第三章 作品测试与分析

## 1. 测试方案

我们通过一个存在简单的UAF(use after free)漏洞的C语言程序来测试我们的C2SafeRust转换工具。

运行C语言程序可以发现存在UAF漏洞，而C2Rust转换后的unsafe Rust代码也存在UAF漏洞；而经过我们的C2SafeRust工具得到的safe Rust代码则会在编译阶段报错，告知程序员存在内存漏洞。

## 2. 测试环境搭建

建议系统为Ubuntu-18.04

a) C2Rust （建议参考https://github.com/immunant/c2rust）

i. 需要安装llvm 6，Python3.4,CMake 3.4.3和openssl 1.0（及之后版本）

```
apt install build-essential llvm-6.0 clang-6.0 libclang-6.0-dev cmake libssl-dev pkg-config python3
```

ii. 安装rustup

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

iii. 添加rustfmt与rustc-dev

```
rustup install nightly-2019-12-05
rustup component add --toolchain nightly-2019-12-05 rustfmt rustc-dev
```

iv. 从crates.io中安装c2rust

```
cargo +nightly-2019-12-05 install c2rust
```

或通过git安装

```
cargo +nightly-2019-12-05 install --git https://github.com/immunant/c2rust.git c2rust
```

b) 安装anytree(https://anytree.readthedocs.io/en/latest/)

```
pip install anytree
```

c) 安装LLVM12（建议参考https://github.com/banach-space/llvm-tutor）

d) 安装Python 3.8.2及以上版本 （参考https://www.python.org/downloads/）

我们也在Docker Hub上创建了一个镜像，可以通过

```
docker pull rainwang6188/c2saferust:devel-version-1
```

拉取镜像，内部已经配置好了环境。

## 3. 测试过程与结果

a) 首先分析uaf.c代码

```c
#include<stdio.h>
#include<stdlib.h>

struct A {
    void (*fnptr)(char* arg);
    char* buf;
};

struct B {
    int B1;
    int B2;
};

void vuln(void){
    printf("In vuln function!\n");
}

int main()
{
    struct A *a = (struct A*) malloc(sizeof(struct A));
    free(a);
    struct B *b = (struct B*) malloc(sizeof(struct B));
    b->B1 = &vuln;
    a->fnptr(a->buf);

    return 0;
}
```

可以看到，在main函数中，free(a)之后仍引用了a对象的fnptr成员，存在UAF漏洞。表现形式为进入了vuln()函数，控制台有“In vuln function!”的输出。

b) 接下来通过c2rust将uaf.c转换为uaf_org.rs（具体使用过程参考README.md），这是unsafe Rust，且存在一些语法问题，无法直接通过编译。我们对其进行修改后（即uaf_mod.rs），使其通过编译，可以看到仍存在UAF漏洞，控制台仍有“In vuln function!”的输出

```
    Finished dev [unoptimized + debuginfo] target(s) in 18.37s
     Running `target/debug/uaf`
In vuln function!
```

c) 为了生成safe Rust，我们通过如下命令利用llvm获得uaf.ll的IR代码

```
export LLVM_DIR=<installation/dir/of/llvm>
$LLVM_DIR/bin/clang  -emit-llvm uaf.c -S -fno-discard-value-
names -o uaf.ll
```

然后，通过如下命令利用我们编写的gen_deptree.py生成依赖树

```
cp uaf.ll uaf.ll.txt
python test.py
```

最终的依赖树可视化结果如下：

```
rain@LAPTOP-Q1OU3987:/mnt/d/c2rust/C2SafeRust/src/uaf_ir$ python3 gen_deptree.py
%retval                         i32
%call                           i8*
└── %0                          %struct.A*
    └── %a                      %struct.A*
        ├── %1                  %struct.A*
        │   └── %2              i8*
        ├── %5                  %struct.A*
        │   └── %fnptr          element ptr
        │       └── %6          void (i8*)*
        └── %7                  %struct.A*
            └── %buf            element ptr
                └── %8          i8*
%call1                          i8*
└── %3                          %struct.B*
    └── %b                      %struct.B*
        └── %4                  %struct.B*
            └── %B1             i32
```

通过分析依赖树，可以知道变量a，b，fnptr和B1是我们需要重点关注的指针类型。

d) 有了上一步的依赖树，就可以通过半自动化脚本对uaf_org.rs这个unsafe rust进行修改，获得uaf_safe.rs，用cargo +nightly编译结果如下：

```
error[E0382]: use of moved value: `a`
  --> src/main.rs:37:48
   |
28 | let mut a = Box::<A>::new_uninit().assume_init();
   |     ----- move occurs because `a` has type `Box<A>`, which does not implement the `Copy` trait
29 | free_box(a);
   |          - value moved here
...
37 | (*a).fnptr.expect("non-null function pointer")((*a).buf);
   |                                                ^^^^^^^^ value used here after move

error: aborting due to previous error; 2 warnings emitted

For more information about this error, try `rustc --explain E0382`.
error: could not compile `uaf`
```

可以看到，程序无法通过编译，也准确判断出对象a在free处失去了所有权，后续的uaf漏洞因为没有所有权被暴露出来。

## 4. 结果分析

通过整个过程，可以看到 unsafe rust 阶段的 Rust 代码与 C 代码是基本等价的，仍存在安全漏洞。而通过我们的 C2SafeRust 工具，分析依赖树获取 policy，并对 unsafe Rust 施加 policy，即可将 unsafe Rust 转换为 safe Rust，从而将 C 中的内存安全漏洞暴露出来。

# 第四章 创新性说明

在本文中，我们详细说明了C2SafeRust工具针对UAF问题的半自动化解决方案，其中创新性主要体现于以下两个方面。

一、半自动化的Rust代码生成

Rust语言目前还处于起步阶段，远远不到大规模应用阶段，而现有的Rust语言转换工具还都处于转换成unsafe-rust后人工修改的程度，我们的C2SafeRust可以针对特定的安全问题将一小部分unsafe-rust代码，依赖于分析产生的policy半自动化地生成修改后的unsafe-rust代码，其中利用了Rust语言本身的安全特性。对于有较高安全要求，却苦于重构任务量的中小型项目，C2SafeRust可以半自动化地直接生成Rust代码。

二、变量生存周期的分析

分析变量的生存周期一直以来都是静态分析的难点，我们使用LLVM生成中间码来对C语言代码进行依赖关系分析，构建一棵变量之间的依赖关系树，从而得到Rust语言中变量的生存周期这一概念。在传统的静态分析中，由于地址与数据之间并没有做一个较好地区分，一般难以对变量的依赖关系做一个全面完整的分析。而在我们的解决方案中，由于针对的是部分与安全问题密切相关的几个变量，具有一定的特征，因此可以完整地分析出这些变量的生存周期，并据此重构Rust语言代码，利用其安全的特性。

# 第五章 总结

Rust因其独特的安全性成为越来越多系统工程的首选语言，但由于它较高的学习门槛，新手入门很困难。尽管网络上已经存在诸如c2rust的转换工具，便于用户将工程从C转换为Rust，但因为生成的仍是unsafe Rust，后续改为safe Rust还需要大量人工编辑。为此我们设计了一个半自动化的C2SafeRust转换器，减轻手工修改的负担。转换过程分为三步：在LLVM IR层面分析C语言代码，生成能够描述Rust中的所有权、生命周期等安全特性的策略（policy）；通过c2rust将.c文件转换为.rs的unsafe Rust；通过半自动化脚本对unsafe Rust施加policy，获得safe Rust。我们也用了一个包含简单UAF漏洞的C语言代码展示了整个过程，在最终获得了safe Rust后，Rust编译器就能将这个UAF漏洞暴露给用户。

# 参考文献

[1]韩心慧,魏爽,叶佳奕,等.二进制程序中的use-after-free漏洞检测技术[J].清华大学学报(自然科学版),2017,57(10):1.

[2]Steve Klabnik, Carol Nichols.Rust权威指南[M].电子工业出版社:北京,2020:112-467

[3]AndrzejWarzyński.LLVM教程[EB/OL].https://github.com/banach-space/llvm-tutor,2019-5-26.

[4]Kevin Wang. Rust 适合用来写 linux 内核模块吗 [EB/OL]. https://zhuanlan.zhihu.com/p/137907908, 2020-5-5