Preventing IPC-facilitated type confusion in Rust

by

Jennifer F. Switzer

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2020

Certified by .....
Howard Shrobe
Principal Research Scientist
Thesis Supervisor

Certified by .....
Hamed Okhravi
Senior Staff, MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Nathan Burow
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Preventing IPC-facilitated type confusion in Rust

by

Jennifer F. Switzer

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Type-safe languages undertake to prevent the type confusion vulnerabilities that arise in type-unsafe languages such as C++. One such type-safe language is Rust, which provides powerful type safety guarantees [1]. However, these guarantees are valid only for a single compilation unit. That is, they may not hold when multiple separately compiled processes communicate. In this work, we explore how type confusion vulnerabilities can still arise when multiple separately compiled, internally type-safe processes share information through inter-process communication (IPC).

We propose safeIPC, a tool for eliminating IPC-facilitated type confusion in Rust. safeIPC is a Rust compiler extension that detects communications over IPC and inserts runtime checks to ensure that type safety is maintained. Programs instrumented with safeIPC throw a runtime error if the type of any data received over IPC is not equivalent to the type expected. Our analysis shows that safeIPC is effective in preventing type confusion vulnerabilities not prevented by Rust alone.

Thesis Supervisor: Howard Shrobe
Title: Principal Research Scientist

Thesis Supervisor: Hamed Okhravi
Title: Senior Staff, MIT Lincoln Laboratory

Thesis Supervisor: Nathan Burow
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to extend my sincere thanks to my advisors Dr. Howie Shrobe, Dr. Hamed Ohkravi, and Dr. Nathan Burow for their guidance and encouragement. Also to Claire Nord, Dr. Bryan Ward, Dr. Richard Skowrya, and the others at Lincoln Labs for providing support and sharing their expertise.

I would also like to thank those who have supported me personally throughout the last six years: mom, dad, Cleo, Scott, Miguel, Mackenzie, Nadia, Kayla, Selam, Virginia, Margaret, and Aneeqa. And a special thanks to Terri Blacker and Corey Reynolds for being the best and most encouraging high school teachers I could ever ask for; I wouldn't be here without you.

Contents

1 Introduction

Type confusion is a well-studied security vulnerability that has been exploited in a wide variety of ways, including control-flow hijacking (CVE-2019-8019), bypassing authentication (CVE-2019-9816), and memory corruption (CVE-2019-9813). Type confusion occurs when one data type is mistaken for another, leading to an incorrect reinterpretation of the data [2]. Type confusion often leads to memory safety violations, which account for approximately 70% of all exploitable security vulnerabilities [3].

Type confusion vulnerabilities typically arise in type-unsafe languages such as C++, where there are little to no language-provided type-safety guarantees [4]. Several mechanisms have been proposed for addressing type confusion vulnerabilities in C++, including HexType [4], TypeSan [2], and CaVer [5]. Although these tools differ in implementation and performance, they all perform a similar function: at compile-time, the code is instrumented and runtime checks are inserted to check for any casts that may cause type confusion. If any of these so-called bad casts is detected at runtime, the program is halted. Other techniques for addressing type confusion include the use of general debugging tools such as fuzzing [6], and manual checks performed by the programmer.

An alternatively strategy is the use of type-safe languages such as Rust [7]. Rather than attempting to add type safety on top of an existing language, these languages have it built in. Rust, for instance, provides strong type safety guarantees by requiring that all variables have a well-defined type, and by strictly limiting the conversions that can occur between types. It provides no implicit type conversion, and allows only well-defined explicit type conversions [7].

Rust's type safety guarantees are provided in large part by static analyses

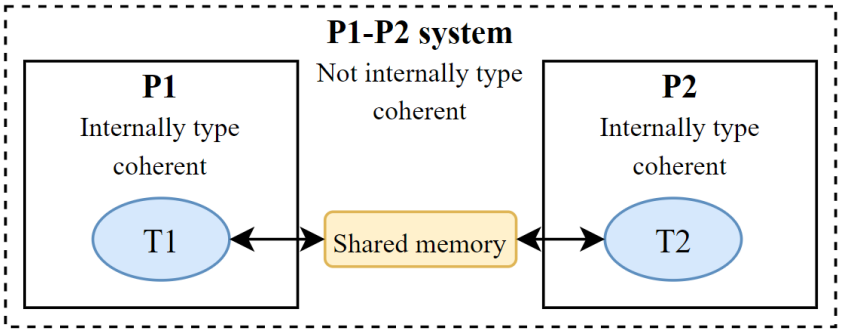

Figure 1: Processes *P1* and *P2* each have their own type definition for the shared data, which may not be equivalent.

performed at compile-time. These analyses track the type and lifetimes of all objects, and throw a compiler error if the programmer attempts to perform a bad type cast. Rust's guarantees may not hold when separately-compiled programs communicate, however. When information is shared between two or more programs, the type information of the shared data is not available for the compiler to check statically. As a result, the Rust compiler is not able to guarantee that all programs share the same type definition for the shared data, and is not able to guarantee that type safety is maintained for these objects. This is a problem, since many computer systems today consist of multiple communicating compilation units [8].

Inter-process communication (IPC) is the mechanism by which multiple processes running on the same machine communicate and share data. When two separately-compiled processes send data over IPC, the shared data's associated type information is lost. Whether these processes are written in a type-safe or type-unsafe language, this loss of type information, coupled with the fact that the compiler does not see both processes, may allow type confusion to occur.

Figure 1 indicates how IPC may facilitate type confusion. When two

separately-compiled processes, *P1* and *P2*, communicate over IPC, there is no shared concept of type for the shared data. *P1* writes data of type *T1* to the shared memory, and *P2* interprets the received data as type *T2*. If these types are not equivalent, type confusion occurs.

Although multiple tools [4, 2, 5] exist for addressing type confusion, there currently aren't any widespread strategies for addressing type confusion between separately-compiled processes. Furthermore, the tools that do exist were created with type-unsafe languages in mind, with many implemented specifically for C++. Type-safe languages such as Rust require an alternate approach that leverages the type information already available at compile-time.

In this work, we contrast the type confusion vulnerabilities that arise in C++, a type-unsafe language, and Rust, a type safe language. We contribute (1) a taxonomy of type confusion vulnerabilities, (2) an analysis of how IPC may facilitate type confusion, even in so-called type safe languages, and (3) safeIPC, a tool for detecting and eliminating IPC-facilitated type confusion in Rust.

2 Background

In this section, we provide an overview of IPC, and categorize common security vulnerabilities related to IPC. Furthermore, we contrast how type confusion arises in type-safe and type-unsafe languages, and provide motivating examples of type confusion in real-world systems. Finally, we discuss how IPC can facilitate type confusion, and provide two case studies for this phenomena.

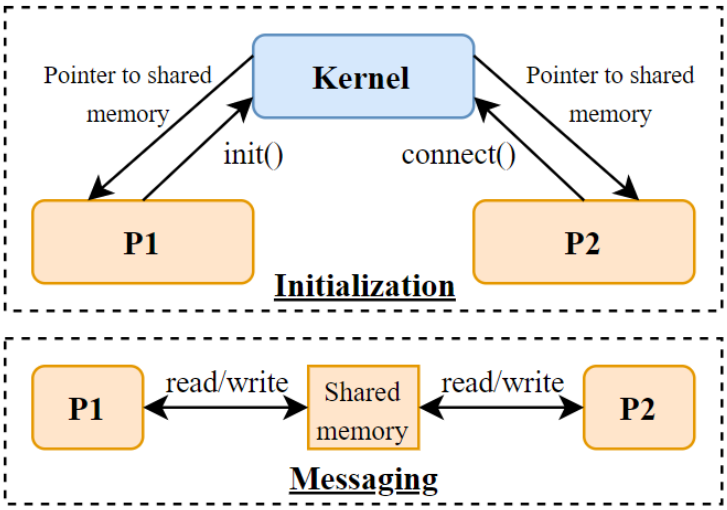

Figure 2: IPC via shared memory requires coordination with the kernel only during initialization; afterwards, both processes can read and write directly to the shared memory region.

2.1 IPC

IPC is the mechanism by which two or more distinct processes communicate. There are two major approaches for implementing IPC: shared memory, in which each process reads and writes directly to the same shared memory region; and message passing, in which the processes make send and receive calls to some shared IPC library that facilitates their communication. In either case, there is typically some coordination with the kernel or another privileged component, either to set up the shared memory region, or to pass messages between processes.

In shared memory implementations, system calls are only required to set up the shared memory region, while message passing typically requires a system call every time a message is sent. As a result, shared memory implementations are typically faster. However, when multiple processes are operating on the same data, concurrency issues may arise that would not occur in a message

passing system. IPC-facilitated type confusion may arise in either shared memory or message passing systems. For the purpose of simplicity, for this project we assume a shared memory system, such as that shown in Figure 2. However, our analysis generalizes to message passing systems as well.

The security guarantees enforced by the kernel (or other privileged components) and the IPC library itself may vary greatly between implementations [9]. For instance, some implementations may automatically encrypt the data sent while others may expect the programmer to do so manually.

IPC-related vulnerabilities are an important consideration for operating systems engineers. Previous research into IPC vulnerabilities in Linux and MacOS indicates that developers are oftentimes unaware of IPC-related risks and best practices [9]. It is therefore desirable to provide an IPC interface that provides security guarantees even when the developer is ignorant of security considerations. This is an especially important consideration for microkernel operating systems. Microkernels are a popular choice for secure operating system design, since they reduce the size of the OS' trusting computing base (TCB) [10], but their fine-grained isolation necessitates increased IPC [11]. IPC security is therefore an important consideration for the design of secure operating systems.

IPC-related vulnerabilities are also an important consideration for browsers. Browsers typically contain many interacting components, and therefore employ a high volume of IPC. Many of these components are exposed to user input or potential adversaries. Exploitable IPC-related vulnerabilities have been discovered in browsers such as Chrome (CVE-2018-6067, CVE-2017-15415), Firefox (CVE-2018-5129), and Safari (CVE-2017-1000121). Although these vulnerabilities are more easily exploitable in part because these browsers are largely written in C++, a type- and memory-unsafe language, switching to

Rust would not fully mitigate them, as described in Section 2.4.2.

2.2 Categorizing IPC Vulnerabilities

There are many classes of IPC-related vulnerabilities, not all of which are related to type confusion. Table 1 categorizes common IPC vulnerabilities, with those that may lead to type confusion indicated.

For this classification, we include value-dependent type confusion as a form of type confusion. Value-dependent type confusion occurs when a value is not within the range of expected values. If a process expects to receive a value in a particular range, and does not perform a manual check to ensure that this expectation is met, undefined behavior may result. Value-dependent type confusion has been the cause of multiple exploitable IPC vulnerabilities (CVE-2017-1000121; CVE-2018-5129).

As shown in Table 1, there are three classes of vulnerabilities that might lead to type confusion:

1. Lack of consistent type definitions between processes (type incoherence)

2. Lack of coherent range expectations between processes (value-dependent type incoherence)

3. Lack of coherent serialize/deserialize definitions between processes (incorrect serialization)

Type incoherence occurs when the sending and the receiving processes do not share the same type definition for the information being shared. For instance, the sending process may send an instance of a struct `cat` for which it has the following type definition:

```
{ struct cat { dinner:  i32, snacktime:  i32 }}
```
[1]

While the receiving process has the definition:

```
{ struct cat { dinner:  i32, snacktime:  String }}
```
[2]

These type definitions are incoherent because they have different types for the field of `snacktime`. The receiving process will attempt to interpret the received data according to the definition it holds for `cat`. As a result, it will attempt to interpret the value of field `snacktime` as a `String` when it is in fact an `i32`. This could lead to unexpected behavior.

Value-dependent type incoherence occurs when the sending and the receiving processes have different expectations for the range of values sent. For instance, a receiving process might expect an array index within some range $[start, end)$. If the sending process sends a value that is outside of this range, for instance $i > end$, an out-of-bounds read or write may occur.

Incorrect serialization occurs when there is some bug in the serialize/dezerialize procedures of one or both of the processes such that:

```
deserialize(serialize(m)) != m
```
[3]

This may cause the data to be corrupted, which could lead to type confusion, for instance if one field of the struct `m` is mistaken for another.

2.3 Type confusion: The safe and the unsafe

One strategy for mitigating type confusion errors is to use a programming language that is considered type-safe. Although there is no universal definition for what makes a language type-safe, in general type-safe languages are those

Vulnerability	Attacks	Defenses	Examples	In scope?
Insufficient authentication	Impersonation	Robust authentication	CVE-2019-3948	No
Lack of or vulnerable encryption	Man-in-the-middle	Robust encryption	Man-in-the-machine [9]	No
Race conditions	Memory corruption	Locks, semaphores	CVE-2017-9687	No
Value-dependent type incoherence	Memory corruption, data-oriented programming (DOP)	Sanitize received data	CVE-2018-5129; CVE-2017-1000121	**Yes**
Type incoherence	Type confusion, DOP	None	Tock; CVE-2018-5130 (Section 2.4)	**Yes**
Incorrect serialization	Denial of service, data corruption	None	CVE-2018-6067; CVE-2017-15415	**Yes**

Table 1: IPC-related vulnerabilities, with those that may lead to type confusion indicated as in-scope

that discourage type errors through a variety of means: disallowing or vetting dangerous operations such as downcasting; ensuring that every object has a well-defined type (i.e. no void pointers); and requiring that functions explicitly declare the type of data that they expect.

Type-safe languages may not prevent all type confusion errors, however. Specifically, language-provided type safety guarantees may not hold when separately compiled processes communicate. Since these guarantees are provided by the compiler [1], they are contingent on the compiler having a complete view of the system. When two such processes communicate over IPC, for instance, type information may be lost across the IPC boundary. There is therefore the possibility for type confusion to occur even if all communicating processes are written in a type-safe language such as Rust.

This section explores type safety in a type-unsafe language (C++), a type-safe language (Rust), and a type-safe language across compilation units (Rust with IPC). The results of this analysis are summarized in Table 2; type confu-

	C++	Rust	Rust w/IPC
Type confusion	✗	✓	✗
Value-dependent type confusion	✗*	✗*	✗
Incompatible serialize/deserialize	✗	✓	✗

Table 2: Covered IPC-facilitated type confusion vulnerabilities (✓= covered, cannot occur; ✗= not covered)
* Although value-dependent type confusion vulnerabilities can technically occur within a single compilation unit (and are therefor marked as not covered), they are much more likely to occur in systems with multiple compilation units. This is because it is more likely for separate components to have incompatible range definitions for shared objects.

sion vulnerabilities that are covered by a given language, and therefore cannot occur, are marked by ✓; while those that are not covered and therefore can occur are marked by a ✗.

2.3.1 C++

Type confusion is the main attack vector for compromising modern C++ applications such as browsers [4]. It has been exploited in a variety of ways, including control-flow hijacking (CVE-2019-8019), bypassing authentication (CVE-2019-9816), and memory corruption (CVE-2019-9813). Even in cases where control-flow integrity is maintained, data corruption facilitated by type confusion can have serious consequences; it may allow an attacker to mount a data-oriented programming (DOP) attack, which have been shown to be realistic security threats [12] [13].

These vulnerabilities are rampant in C++ applications because the language does not provide any sort of built-in type safety guarantees; that is, it allows potentially unsafe typecasts. For example, in C++ an instance of a parent class may be downcast to a descendant class with which it does not share every field. Subsequent attempts to access fields that exist in the de-

Conversion type	Use	Safety
(1) `as` keyword	Converts between primitive types	Safe
(2) `std::convert::{From, Into}`	Traits that can be implemented for any type T to convert from/into T.	Safe
(3) `std::mem::transmute`	Casting between arbitrary types of the same size	Unsafe

Table 3: Type conversion in Rust

pendant class but not the parent may lead to type confusion; for instance, the program may attempt to use a data field as a pointer.

C++ is therefore vulnerable to type confusion vulnerabilities, as indicated in 2. Despite these shortcomings, it is still widely used today, thanks to its speed and the low-level control it provides.

2.3.2 Rust

The Rust language is a popular type-safe language for systems programming that attempts to combine the type and memory-safety guarantees of a high-level language with the low-level control and performance of C/C++ [14]. It provides its type and memory safety guarantees through its ownership model, which allows exactly one unique reference to each live object to exist at a time [15]. Ownership allows Rust to statically track the lifetime of the object, and deallocate it when its owner goes out of scope.

Rust enforces strict rules for type conversion. No implicit type conversion exists, but explicit type conversion is possible. There are three ways to perform type conversion in Rust, summarized in Table 3.

The behavior of (1) and (2) are well-defined, and therefore should not cause type confusion. They function within Rust's strict type system, which prevents bad type casts.

It is sometimes necessary for the programmer to break out of Rust's strict type system, however. In order to allow for the low-level functionality of languages such as C, Rust provides a keyword called `unsafe`. The `unsafe` keyword allows programmers to perform low-level actions that Rust cannot guarantee the safety of but that are sometimes necessary for systems programmers, such as pointer arithmetic or direct memory accesses. One such unsafe function is the `transmute` function, indicated in (3). The behavior of transmute is undefined [16]; it converts between any two arbitrary types of the same size by reinterpreting data of one type as another. It is an unsafe function that, according to Rust's documentation, "should be the absolute last resort" [1].

One important use-case for transmute is the serialization of arbitrary data types. This is oftentimes necessary when sending information over IPC, as described in Section 2.4.

2.3.3 Separately compiled Rust programs

Rust's type safety guarantees hold when a single Rust program is considered. However, they may not hold when multiple separately compiled Rust programs share data, since Rust enforces its type and memory safety guarantees through static analyses performed at compile-time [1] [7]. As a result, although two separately compiled Rust processes may each be internally type safe, a system composed of both these processes communicating over IPC may not. This is because when these processes communicate, the Rust compiler doesn't see the full picture; it doesn't know, for instance, how the information sent by one process will be interpreted by other. That is, the receiving process sees only the data sent, and must reconstruct its type information, leading to possible type confusion. This is an important consideration because most of today's

software systems consist of separately compiled components [17]. Section 2.4.1 provides a case study for IPC-facilitated type confusion in Rust.

2.4 Case studies

This section provides two case studies for IPC-facilitated type confusion, which can occur even if all communicating processes are written in a safe language.

2.4.1 Case study 1: TockOS

We first investigate TockOS, an operating system for embedded devices written in Rust [18]. We discovered that it was indeed possible for type confusion to occur between two separately compiled processes, even if both are written in Rust.

IPC-facilitated type confusion occurs whenever the communicating processes have different definitions for the type of the information being communicated. The proof-of-concept shown in Figure 1 provides one example of this. This code snippet shows one process (the sender) issuing a password request to another (the receiver). On the client side, the sender defines the message to be sent, which is of type `Pwr`. It then writes the value of the `user_id` field to the console, which will output a value of 4. To send the message over IPC, the sender must first serialize it. It does so using `transmute`, which allows for conversion between any two arbitrary types of the same size [7]. Finally, the sender sends the serialized bytes to the server. In Tock, this means writing the bytes to a shared memory region that is initialized at the beginning of the communications.

On the receiver side, the receiver must define and register a callback, which will be triggered upon receipt of a message from the sender. In the code above,

Sender

```
struct Pwr {
    timestamp: u64,
    user_id: u64,
    website_id: u64,
    master_pw: u64
};
let msg = Pwr {
    timestamp: 5,
    user_id: 4,
    website_id: 12,
    master_pw: 6
};
console.write(msg.user_id);
let payload = unsafe {
    transmute::<Pwr,[u8; 32]>(msg)
};
server.write_bytes(&payload);
```

Receiver

```
struct Pwr {
    timestamp: u64,
    master_pw: u64,
    website_id: u64,
    user_id: u64
};
let mut cb =
    Cb::new(|msg: &mut [u8; 32]| {
    let recovered: Pwr = unsafe {
        transmute::<[u8;32],Pwr>(msg)
    };
    console.write(recovered.user_id);
});
IpcServer::start(&mut cb);


.
```

Listing 1: It's possible to exploit Tock's IPC mechanism to cause type confusion

the receiver callback, `cb`, first recovers the message by using `transmute` to convert from `[u8; 32]` back into `Pwr`. Note that the receiver does not share the exact same definiton of `Pwr` with the sender; the order of fields `master_pw` and `user_id` are flipped. Therefore, the values of these two fields will be flipped as compared to the sender's intended message. Rather than outputting 4 as the value of the `user_id` field, the server will output 6.

This simple example illustrates that the lack of shared type definitions between two separately compiled programs can indeed lead to type confusion, even when those programs are written in a type-safe language such as Rust.

2.4.2 Case study 2: Firefox

IPC-facilitated type confusion was the cause of an exploitable vulnerability in Firefox 59 (CVE-2018-5130). The bug was triggered by RTP (Real-time transport protocol) packets with an incorrectly-specified payload type.

Each RTP packet consists of an RTP header and a payload. The header contains metadata about the object, including the payload type. The pay-

load contains serialized data. The receiving process uses the specified payload type in order to correctly reinterpret the payload. If the payload type is not correctly specified, undefined behavior may occur.

Researchers determined that it was possible to deliberately exploit this vulnerability in order to crash Firefox, as well as cause invalid reads or writes and divide by zero exceptions. The vulnerability has since been patched.

Sender

```
d = Audio { ... };
payld = transmute::<Audio,[u64]>(d);
header = RtpHeader {
    payloadType: Video
};
message = RtpPacket {
    header: header,
    payload: payload
};
send(message);
```

Receiver

```
m = receive();
dt = m.header.payloadType; //Video
rd = transmute::<[u64],dt>(m.payload);
.
```

Listing 2: Type incoherence in Firefox led to data being parsed incorrectly

Listing 2 demonstrates how this vulnerability can be exploited. The sender incorrectly sets the `payloadType` field in the RTP header to `Video`, when the data is in fact of type `Audio`. As a result, the receiving process will attempt to interpret the received payload as `Video`, which could lead to undefined behavior.

It is important to note that this vulnerability would not be mitigated simply by using Rust instead of C++. Since the sending and receiving process are separately compiled, they do not share a type definition for the payload. If the receiving process were written in Rust, it would have had to make use of an unsafe `transmute` call to interpret the received bytes. Given the wrong value for the type of the payload, the Rust version of this program would still misinterpret the received payload.

3 Threat model

For this project, we consider two or more separately-compiled applications communicating over IPC, and focus exclusively on IPC-related vulnerabilities that can lead to type confusion.

Hardware is trusted and assumed to be free of exploitable errors; vulnerabilities such as Spectre [19] are out of scope. The kernel, and any other privileged components of the OS, are considered to be benign but potentially buggy. The communicating applications are written in a safe language such as Rust but are compiled separately. Any of the communicating applications may be buggy or authored by a malicious user. All applications must be instrumented by safeIPC , however, and it must not be possible for an attacker to remove this instrumentation.

We also make certain assumptions about the implementation of the IPC mechanism in question. These assumptions allow us to ignore the out of scope implementation errors indicated in Table 1.

First, we assume that all applications know who they are talking to (in other words, all actors have been properly authenticated). Furthermore, we assume that all messages are strongly encrypted, and that there is no opportunity for message tampering or interception. Impersonation and man-in-the-middle attacks are therefore out of scope.

3.1 In-scope vulnerabilities

Consider two processes *p1* and *p2* communicating over IPC on a single machine. We assume that the IPC channel has been correctly initialized, both actors have been authenticated, and the channel is readable and writeable only by the two actors.

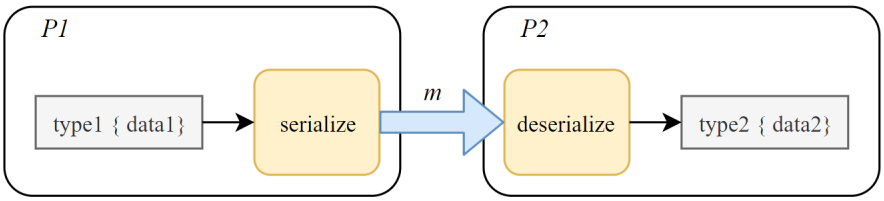

Figure 3: A simplified view of an IPC send

Figure 3 illustrates a single IPC send and receive. Process *p1* is sending data `data1` of type `type1`. It first serializes `data1` into message `m`. `m` is sent over IPC to the receiving process *p2*. *p2* deserializes the message into type `type2`. `type2` is the type that process *p2* expects the data to have.

There are three classes of vulnerabilities that can occur in this simplified model.

1. **Type incoherence:** Occurs when (`type1` != `type2`). May lead to data corruption (`data1` != `data2`) or misinterpretation.

2. **Incorrect serialization:** Occurs when $\texttt{deserialize}_{p2}\texttt{(serialize}_{p1}\texttt{(m))}$!= `m`. May lead to data corruption (`data1` != `data2`) or misinterpretation.

3. **Value-dependent type incoherence:** Occurs when `data2` = `data1` but is outside of the range expected by *p2*. May lead to undefined behavior, such as out-of-bounds reads and writes.

As indicated in Table 1, all of these vulnerability classes have been the cause of real-world exploitable vulnerabilities.

4 Design

safeIPC is an automated tool for the detection and prevention of IPC-facilitated type confusion errors in Rust. Specifically, it (1) detects potentially incoherent types at compile-time, and (2) raises runtime errors when appropriate to prevent them from occurring.

In this section we first define exactly what it means for two separately defined types to be coherent or incoherent. We then lay out our design goals, and give an overview of our solution, safeIPC .

4.1 Defining type coherence

Within a single compiled program, the question of whether or not two objects have the same type is simple, since both objects share the same type definitions. However, between separately compiled programs there is no such guarantee. For instance, two separate Rust programs could both have a user-defined type `foo`, but there is no guarantee that `foo` shares the same definition in both programs. In order to enforce type coherence between separately compiled programs, we must first define exactly what it means for two separately-defined types to be coherent.

For Rust's built-in types, the question of type coherence is straightforward; two types are coherent only if they are the same. For variably sized types such as arrays, size is included in this type definition; `[u8;25]` is not the same type as `[u8;50]`; `i32` is not the same type as `i64`.

For user-defined types (structs and enums), we have to be more specific about what we consider to be type information. Table 4 summarizes the information that we choose (and don't choose) to include in our definition of type.

	Information type	Type or semantic
Included	1. Name of the struct/enum	Type
	2. Names of the struct/enum's fields	Semantic
	3. Types of the struct/enum's fields	Type
	4. Ordering of the fields	Semantic
	5. For enums, 1-4 for all variants	Type
	6. Module name	Semantic
Not included	7. Implemented traits	Semantic
	8. Implemented functionalities	Semantic
	9. Visibility of each field	Semantic
	10. Range of acceptable values for each field	Semantic

Table 4: We include both type and semantic information in our definition of "type", but do not consider struct visibility, implemented functionalities, or value-dependent types

Our definition of type includes information traditionally classified as semantics. For instance, we include the names of all fields as well as their ordering within the struct. We make this decision because many type confusion vulnerabilities arise from these semantic mismatches. For instance, the vulnerability described in Section 2.4.1 and summarized in Listing 1 occurs due to the mismatch in field orderings between the two definitions of `Pwr`.

Sender's definition (T1)

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2)

```
struct Pwrequest {
        master_pw: u64,
        timestamp: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 1: A mismatch in field ordering led to the vulnerability described in Section 2.4.1

Our definition of type also does not include the methods or traits that are implemented for the struct or enum. An example of this sort of mismatch is shown in Listing 2. Although both definitions of `Pwr` are the same in terms of data, they are implemented differently; the sender implements an `inc_time` function, while the receiver implements a `clear_time` function.

Sender's definition (T1)

```
struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
impl Pwrequest {
   fn inc_time(&mut self) {
        self.timestamp += 1;
   }
};
```

Receiver's definition (T2)

```
struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
impl Pwrequest {
   fn clear_time(&mut self) {
        self.timestamp = 0;
   }
};
```

Listing 2: We do not include mismatches in implemented functionalities in our definition of type incoherence

We chose to exclude this information for two main reasons: first, encoding and transmitting all implemented functionalities would become untenable for objects with complex implementations, and would be a non-trivial engineering task; second, a mismatch in implemented functionalities alone should not lead to type confusion. We also do not consider whether the struct and its fields are public or private. We assume that private data will never be sent over IPC. We choose furthermore to ignore the question of value-dependent types for our implementation. We make this decision due to the potential complexity of specifying value-dependent types, which can be Turing-complete. Enforcing value-dependent type coherence is left as a future extension.

Type incoherence therefore occurs when two types do not have the exact same definition of type as described by 1-6 in Table 4. There are many ways that two separately-defined types can be incoherent, summarized in Table 5.

Mismatch in:	Covered by C++	Covered by Rust	Example
Size of object	✗	✓	Appendix A. 6
Struct/enum name	✗	✗	Appendix A. 7
Module name	✗	✗	Appendix A. 9
Field names	✗	✗	Appendix A. 8
Field types	✗	✗	Appendix A. 11
Field ordering	✗	✗	Appendix A. 10

Table 5: C++ and Rust for the most part cannot detect type incoherence between separately-defined types

IPC-facilitated type confusion occurs when two communicating processes hold incoherent types for shared data. As shown in Table 5, regular Rust does not protect against this category of vulnerabilities (with the exception of incoherent types of different sizes).

4.2 Design goals

Our overarching goal is to identify whenever two communicating processes hold incoherent types for shared data, and prevent any such error from causing IPC-facilitated type confusion. Our solution should detect, for any communication over IPC, whether the type of the data being sent is incoherent with the type expected by the receiver. If code that would trigger such a mismatch is invoked, a runtime error should be raised.

More specifically, our solution should meet the following goals:

1. Detect and prevent classes 1-6 of type incoherence identified in Table 4
2. Have an acceptable runtime overhead that is maintained across all common workloads.

3. Require little to no effort or security knowledge from the programmer.

While goal 1 follows from our problem statement, goals 2 and 3 are important for ensuring the practicality of our solution. It must not incur an overly large runtime overhead, or programmers would not be willing to adopt it. Similarly, it must not require any security knowledge from the programmer, especially given the observation that many developers are relatively unaware of IPC-related security considerations [9].

4.3 Proposed solution: safeIPC

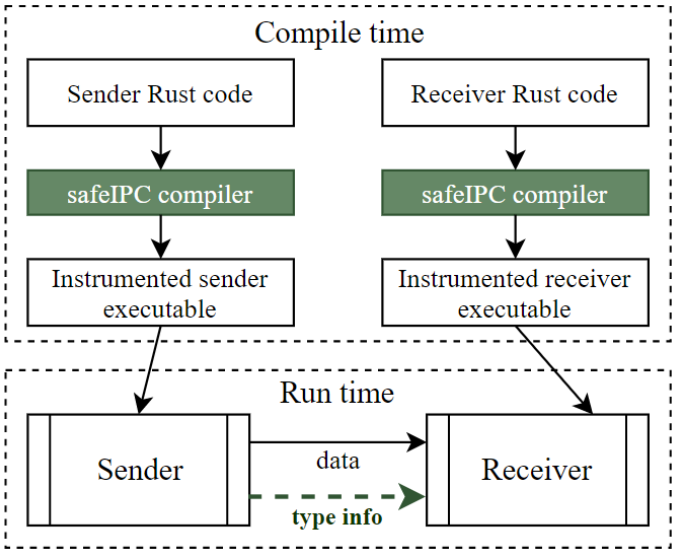

Figure 4: At compile time, safeIPC instruments the sending and receiving processes with automatic checks for type coherence; at run time, these checks ensure that both processes have a shared concept of type for the data being shared.

safeIPC is a Rust compiler extension that prevents exploitable type incoherence vulnerabilities, by automatically communicating type information across IPC boundaries. It does so by performing static analysis in order to identify any sends or receives over IPC, and instrumenting the code to automatically send type information along with the data.

As shown in Figure 4, both the sending and the receiving processes are compiled with the safeIPC compiler. At compile-time, safeIPC collects and preserves the type information for any data sent or received over untyped IPC. It outputs an instrumented version of the input application that automatically shares this information with any process it communicates with.

At runtime, each time data is transmitted over IPC, a second, secure channel is opened in order to also communicate the type of the data. If there is type incoherence, an error is raised.

This process involves three steps: first, collecting accurate type information for the transmitted data, as well as the type information that is expected by the receiver; second, transmitting that type information from sender to receiver; and third, on the receiving side, comparing the expected type information with what was actually received.

For simplicity, the following sections assume only two processes communicating over IPC, with one acting as the sender and the other as the receiver. However, safeIPC generalizes to more complicated systems.

4.4 Collecting type information

For any given communication, there are two pieces of type information needed: (1) the original type of the information sent, and (2) the type of data that is expected by the receiver. These can be determined in a similar manner through static analyses performed at compile-time.

To facilitate this analysis, safeIPC assumes the following properties of the IPC mechanism in consideration: (1) the API includes a send and receive function, and these are the only method for sending or receiving data over IPC, respectively; (2) information must be serialized before being sent, and deseri-

alized upon receipt; (3) this serialization and deserialization is accomplished via Rust's `transmute`.

4.4.1 Sender

On the sender side, the first step is to identify a call to the IPC send function. The object being sent must then be inspected, and its original type determined. This is accomplished by tracing backward through the use-def chain of the object. Use-def chains are compiler-internal data structures that contain, for a given use of a variable, all of the possible definitions it could have [20]. For the purpose of this project, we assume there is only one possible definition for the value being sent; in other words, we assume no branching or conditionals in its definition.

Once we have the definition, we check whether or not its right-hand side is a transmute. If it is, the input to the transmute is then inspected, and its type determined. The type of this input object is the original type of the data being sent over IPC. For instance, in the code shown below, the input type to the transmute (and thus the data's original type) is `foo`.

```
m = transmute<foo, [u8; 32]>(d);
```
[4]

If no transmute is found, then the original type of the information must be equal to its current type.

4.4.2 Receiver

On the receiver side, a call to the IPC receive function is identified. The received message is identified, and the analysis traces forward through its def-use chain until it encounters a call to transmute. Def-use chains are the inverse

of use-defs; they contain, for a given definition, all of its potential future uses [20].

This transmute call will typically convert the received information into a richer target type. For instance, in the code shown below, the received byte array is transmuted into an object of type `bar`.

```
d = transmute<[u8; 32], bar>(m);
```
[5]

The target type is the expected type of the data. If no transmute is found, then the expected type of the information is simply its original type upon receipt.

Once actual and expected type information has been collected for all IPC send and receive calls, the first stage of the analysis is complete. This information is preserved and passed to the second stage: transmission.

4.5 Transmitting the type

Once type information has been collected, it must be transmitted from sender to receiver. safeIPC accomplishes this via instrumentation on the sending application. It inserts a runtime call prior to the IPC send that transmits to the receiver the actual type information for the data being sent.

4.6 Enforcing coherence

On the receiver side, the actual type information for the received data must be compared to the expected. safeIPC inserts a runtime call prior to the IPC receive that reads the actual type information sent by the sender, and compares it to what is expected. If the types are incoherent, a runtime error is thrown.

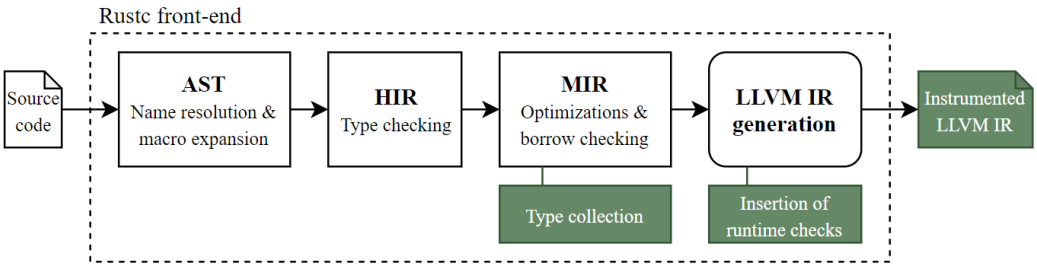

Figure 5: safeIPC was implemented as an extension to the Rust compiler front-end

Since these checks occur at runtime, safeIPC detects only those type confusion errors that are actually invoked.

5 Implementation

We implemented safeIPC as an extension to the Rust compiler, rustc [16] (Rustc version 1.33.0). Major modifications were made at two stages of compilation, as shown in Figure 5: (1) at the mid-level IR (MIR) stage, a pass was added to perform type collection on any objects sent over untyped IPC as described in Section 4.4; and (2) the LLVM IR codegen stage was modified to insert the runtime checks described in Section 4.5 as LLVM function calls. The function calls were implemented as a compiler runtime library. Inlining the function calls is a performance optimization that we leave as future work. We did not modify the LLVM backend.

We chose the MIR stage to perform type collection because it is the earliest compilation stage that is type checked. As a result, we are able to obtain accurate Rust types for all objects while still maintaining a semblance to Rust that facilitates the analysis. The actual insertion of runtime checks is performed at the LLVM IR generation stage.

Our implementation of safeIPC consists of approximately 910 lines of code added to the Rust compiler, distributed as follows: 650 lines added to implement the MIR pass for type collection; 50 lines added to LLVM codegen for the insertion of runtime checks; and 100 lines added to the rustc driver to hook in the added pass. The safeIPC runtime library consists of another 100 lines of code.

At runtime, the instrumented code invokes the safeIPC runtime to, on the sender side, write the actual type information for the data being sent and, on the receiver side, read the actual type information and compare it to what is expected. If there is a mismatch between the type sent and the type expected, the instrumentation will throw a runtime error that halts execution of the receiver before it can actually receive the data.

5.1 IPC detection and type collection

safeIPC performs IPC detection and type collection in a single MIR pass inserted at the end of the Rust compiler's MIR stage.

IPC detection: The programmer specifies, the names of the IPC send and receive functions of interest. For convenience, this specification is performed via environment variables. The type collection pass then checks for calls to these functions. When a call is identified, safeIPC invokes type collection to determine, for IPC sends, the actual type being sent, and for IPC receives, the type that is expected.

Type collection: safeIPC uses the use-def analysis described in Section 4.4 to collect the relevant type information. The type information is hashed to a constant size. This facilitates transmission, and ensures that overhead does not grow with the complexity of the type information. An MIR basic block

is then inserted prior to the IPC call, and annotated with the hashed type information and whether the following call is a send or receive. This block is a do-nothing block that acts as a placeholder for the next stage of the analysis: code instrumentation.

5.2 Code instrumentation

safeIPC performs the actual code instrumentation at the LLVM IR generation stage, where MIR is converted into LLVM IR. When the do-nothing block described above is detected, the modified compiler replaces the block with a call to the appropriate library function: a type transmission function, if a send is being instrumented; and a type receipt function, if a receive is being instrumented. The type information encoded in the do-nothing block is automatically passed as input to the library function.

5.3 Runtime library

We define two runtime library functions to perform type transmission and coherency enforcement: sendIPCtype and receiveIPCtype.

sendIPCtype takes as arguments the actual type information for the data being sent, as well as a handler to the type transmission channel. It writes the type information to the channel.

receiveIPCtype takes as arguments the expected type information for the data being received, as well as a handler to the type transmission channel. It reads the actual type information from the channel, and compares it to the expected. If the types are incoherent, a runtime error is thrown.

6 Evaluation

We evaluate safeIPC based on its coverage of the type confusion vulnerabilities identified in Section 4.1, the overhead it incurs, and its usability. Specifically, we address the following questions:

1. Does safeIPC prevent all type coherence errors identified in Section 4.1?

2. What is the runtime overhead of safeIPC , and is it constant across different workloads?

3. How usable is safeIPC , in terms of the level of effort and security knowledge required by the programmer?

Evaluation setup: For our evaluation, we consider two communicating processes communicating over IPC through shared memory. We instrument both processes with safeIPC . We then measure the runtime overhead of sends and receives between the processes with and without safeIPC instrumentation. These tests were performed on Ubuntu 16.04.1 LTS with 16 cores, 64GB RAM, and 64GB of disk space.

6.1 Security

Q1. Does safeIPC prevent all type coherence errors identified in Section 4.1? To answer this question, we created a test suite of type incoherence vulnerabilities (Appendix A) that cover all classes of type incoherence identified in Section 4.1. We test both generic Rust, and Rust instrumented with safeIPC , against these examples. The results are summarized in Table 6 rows 1-6.

safeIPC is effective in preventing all classes of type coherence errors identified. It covers a much wider variety of type confusion vulnerabilities than C++ or even regular Rust. C++ is incapable of detecting or preventing any of the classes of type confusion vulnerabilities we identified. Uninstrumented Rust is only capable of detecting incompatible types of different sizes. safeIPC would also have prevented both cases of type incoherence identified in Section 2.4.

In the Tock case study, the vulnerability was caused by the fact that the sending and receiving process had different definitions of the struct `Pwr`, as shown in Listing 3.

Sender's definition (T1)

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2)

```
struct Pwrequest {
        master_pw: u64,
        timestamp: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 3: These incoherent types led to a type confusion vulnerability in Tock

Although both definitions of `Pwr` have the same number and types of fields (e.g. four fields of type u64), they differ in the ordering if their first and second field. As described in Section 2.4, this discrepancy causes the receiving process to confuse the values of `timestamp` and `master_pw`, leading to undefined behavior. safeIPC identifies this type incoherence, and prevents the vulnerability. While both processes compile successfully, a runtime error is thrown when the receiving process attempts to read the message sent by the sender.

In the Firefox case study, the vulnerability arose because it was possible for the sending process to specify and incorrect type for the payload, as shown in Listing 4.

Sender

```
d = Audio { ... };
payld = transmute::<Audio,[u64]>(d);
header = RtpHeader {
    payloadType: Video
};
message = RtpPacket {
    header: header,
    payload: payload
};
send(message);
```

Receiver

```
m = receive();
dt = m.header.payloadType; //Video
rd = transmute::<[u64],dt>(m.payload);
.
```

Listing 4: Type incoherence in Firefox led to data being parsed incorrectly

The sending process claims that its payload is of type `Audio`, when in fact it is of type `Video`. As a result, the receiving process attempts to interpret the audio data as video. If these programs were instrumented with safeIPC, the actual type of the data payload would be automatically communicated to the receiving process, thus preventing the vulnerability.

Mismatch type	**C++**	**Rust**	**Rust w/safeIPC**	**Example**
1. Size of object	✗	✓	✓	Appendix A. 6
2. Struct/enum name	✗	✗	✓	Appendix A. 7
3. Module name	✗	✗	✓	Appendix A. 9
4. Field names	✗	✗	✓	Appendix A. 8
5. Field types	✗	✗	✓	Appendix A. 11
6. Field ordering	✗	✗	✓	Appendix A. 10
7. Public or private	✗	✗	○	Appendix A. 12
8. Implemented functionalities	✗	✗	○	Appendix A. 13
9. Value-dependent types	✗	✗	○	Appendix A. 14

Table 6: Type confusion errors caught with and without safeIPC (○ = potential extension)

	Uninstrumented			Instrumented			
	Mean	Median	Std dev	Mean	Median	Std dev	Overhead
Send	668	651	310	1380	1336	652	+106%
Receive	346	330	267	781	766	336	+126%

Table 7: Microbenchmark results (in cycles) for safeIPC, averaged over 40,000 iterations

6.2 Performance

Q2. What is the runtime overhead of safeIPC ? We performed microbenchmark testing on the same simple program with and without our instrumentation, in order to determine the runtime overhead incurred by instrumented sends and receives. safeIPC was found to incur a constant overhead of approximately $2x$ for IPC communications. These results are summarized in Table 7.

Instrumented sends take on average 106% longer than uninstrumented sends, while receives take 126% longer. These results make intuitive sense. In order to send the type information of the data being communicated, instrumented code must make two writes rather than one for every send. Every instrumented receive, on the other hand, must perform not only an extra read, but also the enforcement logic needed to ensure that the received and expected types match.

Figures 1 and 2 show the probabilities density functions of 40,000 sends and receives with both uninstrumented (red) and instrumented (blue) code. In both cases, the width of the probability density function (the standard deviation) is relatively constant between instrumented and uninstrumented code. The instrumented reads and writes incur an approximately $2x$ overhead.

Although a 2x overhead may be significant for a single read or write, in

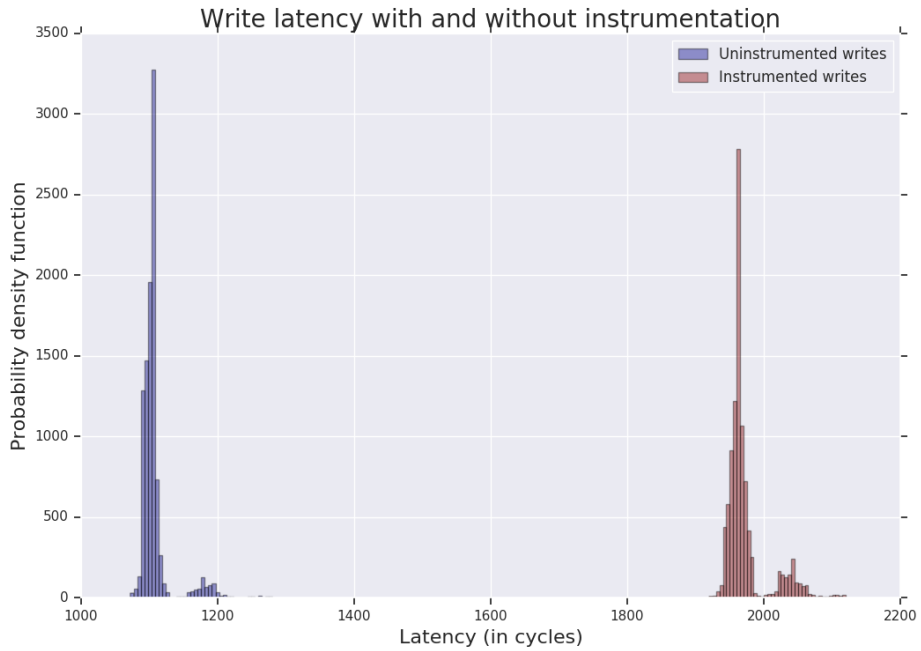

Figure 6: Instrumented writes incurred an approximately 2x overhead (results shown for 40,000 iterations)

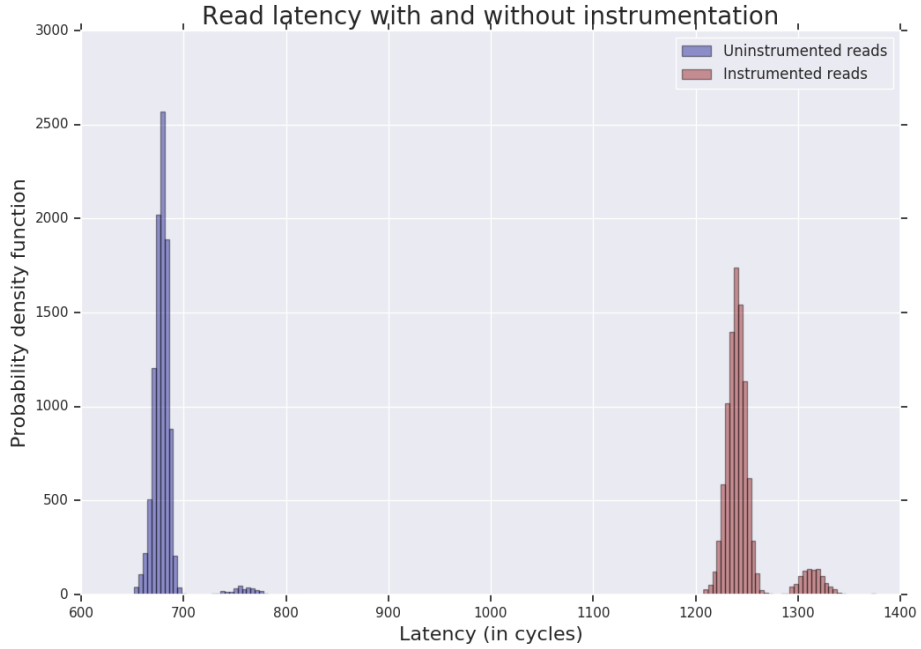

Figure 7: Instrumented reads incurred an approximately 2x overhead (results shown for 40,000 iterations)

practice even for systems that make heavy use of IPC, such as browsers, the overall latency of the system should not increase significantly.

Q3. Is this overhead constant across different workloads? To answer this question, we identified three variables that might have an impact on the overall latency of instrumented reads and writes: (1) the amount of data being sent; (2) the complexity of the type being sent, which corresponds to the amount of type information; and (3) the number of sequential sends and receives.

We find that varying these parameters does not have a significant impact on the overhead of either sends or receives. The complexity of the type information was varied by changing the number of fields in the struct to be sent. As shown in Figure 3, this does not have any significant impact on the overall latency of either reads (in blue) or writes (in yellow). The number of sequential sends and receives were varied by iterating over a single write or read, respectively. As shown in Figure 4, varying the number of sequential operations did not significantly impact the overhead of either reads or writes.

The full results are summarized in Table 8. Each data point represents the average over 10,000 send or receives. The unusual write overhead spike at 1024 bytes (the size of one page in our OS) of data sent is caused by caching behavior related to page table sizing. In general, as the number of data sent in a single write increases, the overhead of the safeIPC instrumentation decreases.

6.3 Usability

Q4. How usable is safeIPC , in terms of the level of effort and security knowledge required by the programmer? Installing safeIPC is a simple matter of downloading our augmented Rust compiler and building

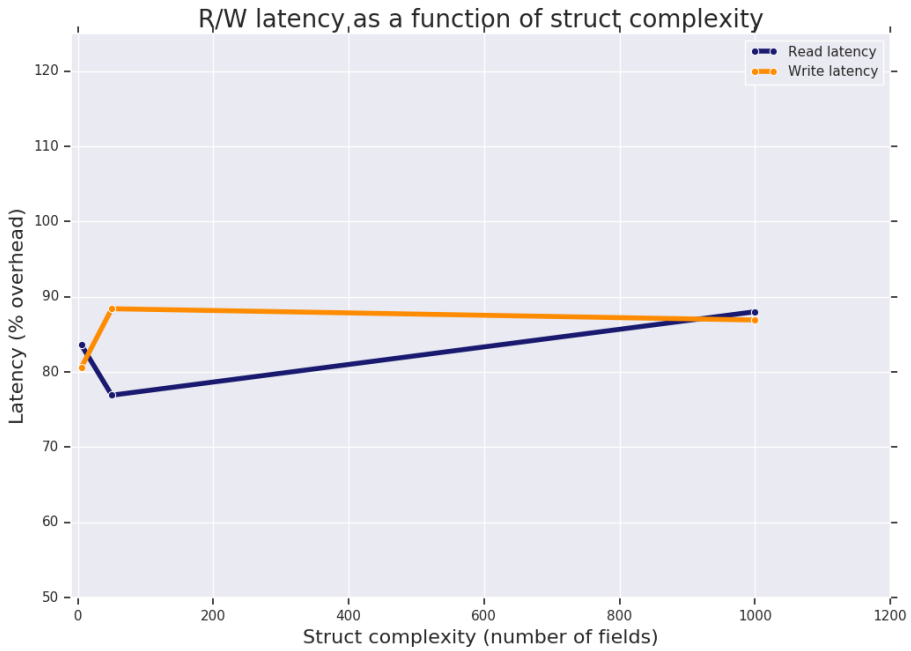

Figure 8: The overhead of reads and writes remain relatively constant as the complexity of the data being sent increases

Figure 9: The normalized overhead of reads and writes remains relatively constant as more are performed sequentially

Variable	Magnitude	Send overhead	Receive overhead
Complexity of type	Baseline (1 field)	+79.8%	+84.1%
	Moderate (50 fields)	+81.2%	+83.7%
	High (1000 fields)	+84.5%	+81.2%
Amount of data sent	Baseline (32 bytes)	+79.8%	+84.1%
	Moderate (256 bytes)	+63.5%	+57.1%
	High (1024 bytes)	+155.7%	+77.8%
Number of iterations	Baseline (1)	+79.8%	+84.1%
	Moderate (100)	+81.0%	+109.3%
	High (1,000)	+89.6%	+97.5%

Table 8: Send and receive latency remain approximately constant for different workloads

it. Then each communicating application in the system to be instrumented must be compiled by the augmented compiler. There is no code annotation or further work required of the programmer. The programmer need not have a deep understanding of computer security, since there is no need for them to manually identify potential vulnerabilities or particularly security-critical pieces of code.

The compile-time overhead of using the augmented compiler is approximately $2x$. In most cases, this should not be prohibitive.

7 Discussion

safeIPC was motivated by the observation that type-safety is not necessarily maintained when multiple separately compiled, internally type-safe programs communicate. As the examples described in Section 2.4, and summarized in Appendix A indicate, type confusion can arise between communicating processes even if they are written in a type safe language. Although safeIPC focuses on IPC-facilitated type confusion, this observation holds for any system

involving communication between multiple compilation units. For instance, system calls and network communications may also be vulnerable to these sort of errors. Given this observation, it may be worth expanding discussions of type-safety to consider system-level safety.

7.1 Implications for secure system design

One major trend in computer systems security is the development of "clean-slate" systems, such as browsers and operating systems, that are designed from the ground-up with security in mind [21, 22]. One such project is the Robust Mission Computer project, which aims to create a secure-by-design system that incorporates everything from hardware to OS to user applications [23]. This project, and many others, leverage Rust to provide type and memory safety [18, 24]. However, they also consist of multiple compilation units and therefore must rely on IPC for communication between these components. As the safeIPC project has shown, communications over IPC are a potential source of vulnerabilities even when all components are written in a safe language. Secure system designers should be aware of these potential vulnerabilities and should take steps to mitigate them. safeIPC could be a powerful tool for ensuring that exploitable IPC-facilitated type-confusion vulnerabilities do not occur in such systems.

7.2 Future work

7.2.1 Expanding coverage

Table 6 rows 7-9 indicate type incoherence errors not covered by safeIPC . These non-covered errors fall into two categories: errors caused by mismatches

in the type definition (7-8); and errors caused by mismatches in the range of acceptable values (9), i.e. value-dependent type incoherence.

safeIPC could be extended to cover any or all of these errors. Covering errors related to mismatches in struct visibility (7) would be a fairly trivial extension; we do not recommend doing so, however, to avoid encouraging the unwise practice of sending private structs over IPC. On the other hand, covering errors related to mismatches in mismatched functionalities (8), while it could be done, would require significant effort. It would involve modifying safeIPC 's type hash to include the definitions of all implemented functionalities and traits. Collecting this information would be a non-trivial engineering task. It is not clear, however, whether mismatches in implemented functionalities can actually lead to type confusion.

7.2.2 Support for value-dependent types

Covering errors related to value-dependent type incoherence could also be done on top of safeIPC . This would involve including range information in the definition of type for each field. While this would be technically possible, full support for value-dependent types would become untenable. Fully-specified value-dependent types have been shown to be Turing complete.

It might be feasible, however, to implement a limited, simplified version of value-dependent types. For instance, value-dependent types that rely only on constants (i.e. $x > 5$), or other fields of the same struct (i.e. self.y $>$ self.x), could be covered without requiring additional program analysis. However, support for value-dependent types that rely on other program variables would require additional program analysis. This version of value-dependent types, though simple, has the potential to cover a wide range of value-dependent type

confusion.

7.2.3 Support for serialization errors

safeIPC could also theoretically be extended to cover serialization errors as described in Section. These errors arise when the sending and receiving process hold incompatible definitions for the serialize/deserialize procedures.

Extending safeIPC to cover these vulnerabilities would involve collecting the definition of serialize/deserialize used by the sender/receiver, and transmitting this information along with the type hash. In terms of engineering effort, this should be fairly straightforward. However, it might be difficult to determine whether or not a serialize and deserialize function are compatible or not.

7.2.4 Extending to other languages

safeIPC could also be extended to other programming languages beyond Rust. Although Rust is popular among the security community, it has not yet seen wide adoption. Furthermore, most real-world systems include some amount of legacy code written in C or C++. In particular, as mentioned in Section 2.1, most browsers are written largely in C++. C++ is therefore a natural target for extending safeIPC .

Extending safeIPC to cover C++ would involve modifying either the C++ front-end compiler or the LLVM back-end to perform the same analysis and instrumentation that safeIPC provides for Rust. Modifying the LLVM back-end may be the best route, since this approach would generalize more easily to other compilers that make use of LLVM.

8 Related work

Previous attempts to mitigate type confusion vulnerabilities falls into two main categories: (1) tools that directly address the problem by identifying and preventing bad typecasts; and (2) more general sanitizing and debugging tools that can be used to mitigate the damage caused by type confusion, but weren't designed with type confusion specifically in mind.

Type-specific tools. safeIPC performs static analysis to identify typecasts resulting from communication over IPC, and insert runtime checks to detect and prevent bad casts. Many tools exist tools that perform a similar function for programs written in C++. Since C++ is not a type-safe language, however, these tools do not consider IPC-facilitated type confusion, and instead focus on type confusion arising from the bad intra-process type casting that C++ allows. HexType [4], CaVer [5], and TypeSan [2] are all examples of such tools. Although they vary in implementation details and coverage, they all perform the same basic function; at compile-time, they detect potential bad casts, and instrument the code to verify these casts at runtime. All are built on top of the LLVM compiler. One important drawback of these projects is that, since they still only consider a single compilation unit at a time, they do not address IPC-facilitated type confusion. As far as we are aware safeIPC is the only tool for addressing such vulnerabilities specifically.

General tools. Other strategies for addressing type confusion vulnerabilities include the use of more general tools to mitigate the impact of bad casts. For instance, Control Flow Integrity (CFI) techniques [25, 26], which detect and prevent invalid control-flow sequences, can help mitigate the damage caused by type confusion vulnerabilities that might otherwise allowed for control-flow hijacking.

Similarly, general tools for detecting memory corruption, such as SoftBound [27] and Baggy Bounds [28] can be employed to indirectly address type confusion vulnerabilities in cases where those vulnerabilities would have led to memory corruption. It's important to note, however, that such tools were created for memory-unsafe languages (usually C), and do not easily generalize to Rust and other such safe languages. They also only address type confusion vulnerabilities that lead to memory corruption.

Debugging tools such as fuzzers [29, 6] and symbolic execution [30] can be used to identify runtime errors that may arise due to type confusion. Once again, although useful, these tools were not created with type confusion specifically in mind. Furthermore, both approaches have their weaknesses: fuzzing does not test every possible control flow sequence, meaning that not all errors are detected; and symbolic execution is typically not feasible in practice due to the huge amount of computing power it requires [31]. While both of these techniques have been used to test Rust programs [32, 33], fuzzing is employed much more ubiquitously. While useful for testing purposes, fuzzing alone cannot prevent type confusion.

9 Conclusion

Type confusion vulnerabilities are a significant source of vulnerabilities in type-unsafe languages such as C++. One strategy for addressing these vulnerabilities is the use of type-safe languages, which provide stronger type safety guarantees. However, these guarantees are valid only for a single compilation unit, and may break down when separately compiled processes communicate.

safeIPC is a tool for eliminating IPC-facilitated type confusion in Rust, a type-safe language. It detects communications over IPC, determines the type

of the information being communicated, and inserts runtime checks to ensure that the sending and receiving process hold coherent type definitions for the information.

Our analysis shows that safeIPC is effective in preventing type confusion vulnerabilities not prevented by Rust alone. Since most of the analysis is performed at compile-time, runtime overhead is minimized. safeIPC does not require any code annotations or security knowledge on behalf of the programmer.

Appendices

Appendices

A Categorized type incoherence

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 5: Coherent types, for reference

Sender's definition (T1):

```
type Pwrequest = [u8; 100];
```

Receiver's definition (T2):

```
type Pwrequest = [u8; 400];
```

Listing 6: Struct size mismatch

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2):

```
struct Pw_request {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 7: Struct name mismatch

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        user_id: u64,
        website_id: u64,
};
```

Listing 8: Field name mismatch

Sender's definition (T1):

```
struct my_module::Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2):

```
struct your_module::Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 9: Module name mismatch

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
```

Receiver's definition (T2):

```
struct Pwrequest {
        master_pw: u64,
        timestamp: u64,
        uid: u64,
        website_id: u64,
};
```

Listing 10: Field order mismatch

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u32,
        metadata: u32
};
```

Receiver's definition (T2):

```
struct Pwrequest {
        timestamp: u32,
        master_pw: u64,
        uid: u64,
        website_id: u64,
        metadata: u32
};
```

Listing 11: Field type mismatch

Sender's definition (T1):

```
struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
```

Receiver's definition (T2):

```
pub struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
```

Listing 12: Visibility mismatch (public or private

Sender's definition (T1):

```
struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
impl Pwrequest {
   fn inc_time(&mut self) {
        self.timestamp += 1;
   }
};
```

Receiver's definition (T2):

```
struct Pwrequest {
    timestamp: u64,
    master_pw: u64,
    uid: u64,
    website_id: u64,
};
impl Pwrequest {
   fn clear_time(&mut self) {
        self.timestamp = 0;
   }
};
```

Listing 13: Mismatch in implemented functionalities

Sender's definition (T1):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
impl Pwrequest {
   fn new(t: u64, mp: u64,
        uid: u64, wid: u64)
        -> Pwrequest {
        assert_eq!(uid, 5);
        Pwrequest {
            timestamp: t,
            master_pw: mp,
            uid: uid,
            website_id: wid,
        }
    }
};
```

Receiver's definition (T2):

```
struct Pwrequest {
        timestamp: u64,
        master_pw: u64,
        uid: u64,
        website_id: u64,
};
impl Pwrequest {
   fn new(t: u64, mp: u64,
        uid: u64, wid: u64)
        -> Pwrequest {
        assert_eq!(uid, 10);
        Pwrequest {
            timestamp: t,
            master_pw: mp,
            uid: uid,
            website_id: wid,
        }
    }
};
```

Listing 14: Value-dependent type incoherence

Bibliography

[1] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[2] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. ACM.

[3] Matthew S. Simpson and Rajeev K. Barua. Memsafe: ensuring the spatial and temporal memory safety of cat runtime. *Software: Practice and Experience*, 43(1):93–128, Feb 2012.

[4] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. Hextype: Efficient detection of type confusion errors for c++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2373–2387, New York, NY, USA, 2017. ACM.

[5] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 81–96, 2015.

[6] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE

AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[7] Rust. The rust programming language.

[8] Ben Niu and Gang Tan. Modular control-flow integrity. In *ACM SIGPLAN Notices*, volume 49, pages 577–587. ACM, 2014.

[9] Thanh Bui, Siddharth Prakash Rao, Markku Antikainen, Viswanathan Manihatty Bojan, and Tuomas Aura. Man-in-the-machine: Exploiting ill-secured communication inside the computer. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1511–1525, Baltimore, MD, August 2018. USENIX Association.

[10] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[11] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 9:1–9:15, New York, NY, USA, 2019. ACM.

[12] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, May 2016.

[14] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017.

[15] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *SIGOPS Oper. Syst. Rev.*, 51(1):94–99, September 2017.

[16] Rust. Guide to rustc development.

[17] Amal Ahmed. Verified Compilers for a Multi-Language World. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15–31, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[18] Tock documentation.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[20] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson India Education Services, 2015.

[21] Silviu Chiricescu, Andre Dehon, Delphine Demange, Suraj Iyer, Aleksey Kliger, Greg Morrisett, Benjamin C. Pierce, Howard Reubenstein, Jonathan M. Smith, Gregory T. Sullivan, and et al. Safe: A clean-slate architecture for secure systems. *2013 IEEE International Conference on Technologies for Homeland Security (HST)*, 2013.

[22] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[23] Hamed Okhravi, Nathan Burow, Richard Skowyra, Bryan Ward, Samuel Jero, Roger Khazan, and Howard Shrobe. One giant leap for computer security.

[24] Brian Anderson, Lars Bergstrom, David Herman, Josh Matthews, Keegan McAllister, Manish Goregaokar, Jack Moffitt, and Simon Sapin. Experience report: Developing the servo web browser engine using rust. *arXiv preprint arXiv:1505.07383*, 2015.

[25] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):16, 2017.

[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307. IEEE, 2014.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[28] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[29] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[30] Corina S. Pundefinedsundefinedreanu and Neha Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 179–180, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[32] Marcus Lindner, Jorge Aparicius, and Per Lindgren. No panic! verification of rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pages 108–114. IEEE, 2018.

[33] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493. IEEE, 2015.