Detecting and Reproducing Error-Code Propagation Bugs in MPI Implementations

Daniel DeFreez
University of California, Davis, USA
dcdefreez@ucdavis.edu

Antara Bhowmick
University of California, Davis, USA
abhowmick@ucdavis.edu

Ignacio Laguna
Lawrence Livermore National Laboratory
ilaguna@llnl.gov

Cindy Rubio-González
University of California, Davis, USA
crubio@ucdavis.edu

Abstract

We present an approach to automatically detect and reproduce error code propagation bugs in MPI implementations. Specifically, we combine static analysis and program repair for bug detection, and apply fault injection to reproduce error propagation bugs found in MPI libraries written in C. We demonstrate our approach on the MPICH library, one of the most popular implementations of MPI, and the MPICH-based implementation MVAPICH, uncovering 447 previously unknown bugs. We discovered that 31 of these bugs result in program crashes, and 60% of the MPICH test suite is susceptible to crashing due to failures to propagate error codes. Moreover, 95 bugs produce undesirable behavior that has been confirmed dynamically, causing tests to fail, hanging processes, or simply dropping error codes before reaching user applications.

• **Software and its engineering** → *Automated static analysis*; *Dynamic analysis*; **Error handling and recovery**; • **Computing methodologies** → **Parallel programming languages**.

1 Introduction

Most large-scale parallel computing applications use the Message-Passing Interface (MPI) to perform multi-node communication. Implementations of the MPI Standard[1] are available as open-source libraries such as MPICH [1] (which is used as base for the vast majority of implementations) and Open MPI [3], as well as vendor-provided implementations. Given the wide use of MPI in high-performance computing (HPC) clusters to support large scientific applications, the correctness and reliability of MPI implementations is paramount for HPC centers and vendors.

Since large-scale production jobs can suffer from frequent failures [29], an important correctness aspect of MPI programs is their ability to handle runtime errors. The MPI Standard (as of the most recent version, 3.1) specifies two error handling modes to allow programmers deal with runtime errors. The default mechanism, *errors are fatal*, causes the MPI program to abort automatically when an error is detected. While this mode is practical for small and short-running applications, it imposes limitations to long-running programs that may need to perform a more graceful termination when an error is encountered; for example, the program may need to save a checkpoint before aborting. To support a less abrupt termination, the MPI Standard supports a second mode of operation, *errors return*, which allows the MPI library to return an error code to the application when an error is detected. Depending on the returned error code, the application can take appropriate actions, such as cleaning up its state before the program is terminated.

The MPI Standard version 4.0, which is expected to be released in 2020, will introduce a new call, `MPI_ERR_IS_CATASTROPHIC`[2], which will allow the application to check whether an error in the library was *catastrophic* or not. An error is defined as catastrophic if the library must abort due to the error, and non-catastrophic otherwise. As a result of this new feature, future applications may check more frequently for non-catastrophic errors (which allow the application to continue) by checking the return error codes.

While the *errors return* model for error handling is useful to applications, implementing it correctly in an MPI library is non-trivial. Error codes must propagate all the way to the user application as reliably as possible and may traverse long call stacks before they arrive to users. Current MPI implementations provide more than 400 user calls (as specified by the standard) and must detect and propagate errors through different software layers, including transport and user-interface layers. More importantly, software defects on

[1] https://www.mpi-forum.org/mpi-31/

[2] https://github.com/mpi-forum/mpi-issues/issues/28

PPoPP '20, February 22–26, 2020, San Diego, California, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374515

the error propagation mechanism can lead to unexpected behavior, from program crashes and hangs to silent failures that can lead to data loss.

In this paper, we present an approach to automatically detect and reproduce *error code propagation bugs* in MPI implementations. Our framework detects error propagation bugs found in MPI libraries. The type of error propagation bugs we focus on this paper are unsaved error codes. Since reproducing these bugs is critical for implementors to determine the consequence of failed propagation, we also present a series of novel strategies for error code injection to efficiently reproduce and isolate MPI error propagation bugs.

While our approach is generally applicable to MPI implementations written in C (the common case), we focus on MPICH [1] and its derivations. MPICH is one of the most popular implementations of MPI and is used as the base for the vast majority of other MPI implementations, including IBM MPI (for Blue Gene), Intel MPI, Cray MPI, Microsoft MPI, Myricom MPI, OSU MVAPICH/MVAPICH2, and others. Because of the large influence of MPICH on several other implementations, improving its correctness and isolating error propagation bugs in it will likely have a significant impact. Our evaluation confirms this—59% of the bugs that we found in MVAPICH [2] were carried over from MPICH.

MPICH is written in C, a language that does not provide support for exception handling, and uses the popular return-code idiom to implement the *errors return* model. Macros define integer error codes that are propagated from one function to another through return values and/or pointer parameters. Unfortunately, explicit propagation of integer error codes is error prone; functions can fail to check and propagate errors to the user. Prior work [8, 25] has developed static analyses to track the propagation of error codes in Linux, which differs significantly from MPI implementations (see Section 3). In this paper, we combine static analysis with program repair to detect error propagation bugs in MPI implementations.

Once detected, a crucial step in understanding error propagation bugs is to reproduce them. Reproduction of MPI error propagation bugs presents several challenges: (1) finding MPI programs that exercise the buggy library code, (2) forcing error conditions so that error codes are generated and propagated, and (3) observing the runtime behavior of the buggy library when an error is detected. In this paper, we propose an approach that addresses the above challenges. First, we automatically identify relevant tests in the MPICH regression test suite that cover each detected bug. Second, we "simulate" the occurrence of errors by automatically injecting error codes and memory failures in the library. Third, we run the corresponding tests to observe the side effects of the propagated bug (e.g., the program crashes, or the error code is not returned to the user program).

We found a total of 447 previously unknown unique bugs in MPICH and MVAPICH, all of which we have manually examined. The MPICH bugs have been reported, and MPICH developers have provided positive feedback on the validity of the bug reports. We find that, despite the existence of hundreds of regression tests in MPICH, the test suite is not sufficient to cover enough code to reproduce all of the bugs. Using our fault injection framework we are able to dynamically reproduce 95 bugs.

We make the following research contributions:

- We present a novel hybrid approach (Section 3.3) that, relying on the unique characteristics of MPI implementations, combines static analysis (Section 3.1) and program repair (Section 3.2) to find error propagation bugs in widely used MPI libraries.
- We present a methodology for reproducing error propagation bugs in MPI libraries (Section 3.4 - Section 3.5).
- We present an experimental evaluation of our technique on the popular MPI implementation MPICH and one of its derivations, MVAPICH, revealing 276 previously unknown error propagation bugs in MPICH and an additional 171 bugs in MVAPICH. Our reproduction technique is able to reproduce 95 bugs (Section 4.1 - Section 4.3).
- We present the first characterization of the consequences of error code propagation bugs in MPI (Section 4.2). We find that error code propagation bugs have severe consequences, with 60% of the MPICH regression tests and all three real MPI programs in our experiments being susceptible to a segmentation fault in the event of an error.

2 Background

This section describes error code checking in MPI, and gives an example of an error propagation bug found in MPICH.

2.1 MPI Error Code Checking

Propagating errors via function return codes is a commonly used mechanism followed by many HPC libraries and APIs: upon return from a function call of a library API, the user checks the return value to determine whether the routine executed correctly. If an error code is returned, the user can execute recovery code.

MPI supports this mode of error code checking. Programmers, however, must indicate to the MPI implementation that they desire to receive error codes—this can be configured using the predefined `MPI_ERRORS_RETURN` error handler. When this mode is set, it avoids the default error handler `MPI_ERRORS_ARE_FATAL`, which terminates the application when an error is found. Propagation is relevant to both modes; either the error code must be propagated until it reaches the user application, or it must be propagated until the implementation aborts execution.

Table 1. Some of the return error codes defined in MPICH.

Error Code Name	Value	Description
MPI_ERR_BUFFER	1	Invalid buffer pointer
MPI_ERR_COUNT	2	Invalid count argument
MPI_ERR_TYPE	3	Invalid datatype argument
MPI_ERR_TAG	4	Invalid tag argument
MPI_ERR_COMM	5	Invalid communicator

The MPI Standard specifies that all MPI routines (except `MPI_Wtime` and `MPI_Wtick`) return an error value upon failure. Error codes are implementation-specific values that indicate the reason the call completed unsuccessfully. Similarly, the standard requires MPI operations that complete successfully to return `MPI_SUCCESS` as the returned value.

Table 1 shows some of the error codes defined in MPICH. These error codes can then be mapped into standardized *error classes* to allow portable handling of error scenarios. The current version of the MPI Standard features 59 error classes defined for categorizing error codes, and in many cases this is sufficient to narrow down the reason for an error. However, in more targeted cases, an application may need to interpret and handle MPI implementation-dependent error codes. More formally, error classes are part of the MPI Standard, whereas error codes are implementation specific and not part of the standard. However, for simplicity of exposition, in the rest of the paper, we refer to both error classes and codes simply as error codes.

While the MPI Standard could be implemented in several ways, most (if not all) implementations are provided as a library. We refer to applications that use an MPI library as the *user application* or simply the *application*.

2.2 Example of an Error Propagation Bug

We present an illustrative example of an error propagation bug. A common operation in MPI programs is duplicating a communicator (groups of MPI process ranks in the MPI library that intend to communicate). Application code to duplicate a communicator usually involves the use of `MPI_Comm_dup` as follows:

```
1  error = MPI_Comm_dup(oldComm, &newComm);
2  checkError(error);
3  ...
4  MPI_Comm_free(&newComm);
```

An example of an error propagation bug in MPICH that could be triggered by `MPI_Comm_dup` is observed in function `MPII_Comm_copy` shown in Figure 1a. `MPII_Comm_copy` is executed in MPICH after the application calls `MPI_Comm_dup`. In turn, `MPII_Comm_copy` calls `MPIR_Comm_map_irregular`, which is shown in Figure 1b. `MPIR_Comm_map_irregular`

```
548 int MPII_Comm_copy(MPIR_Comm *ptr, ...) {
611   if(ptr->comm_kind==COMM_KIND__INTRACOMM)
612     MPIR_Comm_map_irregular(newcomm_ptr,..);
613   else
614     ...
616 fn_fail:
617 fn_exit:
618   FUNC_TERSE_EXIT(MPID_STATE_MPIR_COMM_COPY);
619   return mpi_errno;
620 }
```

(a) Dropping site on Line 612 (`src/mpi/comm/commutil.c`).

```
224   int MPIR_Comm_map_irregular(...) {
225     int mpi_errno = MPI_SUCCESS;
247     MPIR_CHKPMEM_MALLOC(mapper, ...);
260   fn_exit:
263     return mpi_errno;
264   fn_fail:
265     MPIR_CHKPMEM_REAP();
266     goto fn_exit;
267   }
```

(b) Origin site on Line 247 (`src/mpi/commutil.c`).

Figure 1. The origin and the dropping sites of a bug.

might return an error code if, for example, `MPIR_CHKPMEM_MALLOC` fails to allocate memory, but the return value of `MPIR_Comm_map_irregular` is not checked in `MPII_Comm_copy`. In this example the error propagation bug is that `MPII_Comm_copy` **fails to check** and further propagate the error to its callers, including the API function `MPII_Comm_dup`.

Consequences of Error Propagation Bugs. The consequences of error propagation bugs can be serious. In this example, the bug can result in communicator structures that are silently corrupted during duplication. The result of destroying such a corrupted communicator using `MPI_Comm_free` is a null dereference leading to a segmentation fault.

Note that we refer to Line 247 in Figure 1b as the origin site of the error. In general, an origin site is the source location where the error code being propagated is first used. In this case, `MPIR_CHKPMEM_MALLOC` is a macro that uses the error-code macro `MPI_ERR_OTHER`, which is assigned to the variable `mpi_errno`. We refer to Line 612 in Figure 1a as the dropping site. This is the source location at which an error code is dropped, and therefore lost.

Exposing the consequences of error propagation bugs requires a user application that exercises both origin and dropping sites, as well as a runtime injection technique to simulate the occurrence of an error in an MPI implementation.

2.3 Error Code Propagation Challenges

The error code propagation bugs discussed in this paper are calls to functions that may return an error code, and where the call site fails to save the returned error code. There are two challenges to determining which call sites need to have their return value saved.

First, not all functions in an MPI implementation are fallible. The MPI Standard asserts that all MPI routines return an error, but MPI routines may contain calls to many intermediate, implementation-specific functions. Error code propagation bugs are often manifested by the failure to propagate error codes returned by one of these implementation-specific functions, even if the MPI routine directly returns an error code on some paths.

The second challenge is that some functions always succeed, and some functions abort on error instead of returning an error code, as is allowed by the MPI Standard. In these cases, calls that do not check the return value are not necessarily defective. Furthermore, if a function is defective, then it does not correctly represent the propagation of error constants in the MPI implementation, and thus fixing the error-propagation bug might lead to the discovery of additional buggy call sites as the propagation of error codes is expanded. As described in Section 3.2, our approach addresses this by applying a propagation fix to the defective function and then re-analyzing the MPI implementation.

An overly simplistic approach to finding error propagation bugs would be to consider every function as fallible, and to label as buggy every call site whose return value is not saved. This approach has the advantage of being simple to implement and fast to execute, but it results in a large number of false positives. There are many infallible functions in MPICH and every call to one of these functions would be a false positive. One such infallible function is `MPL_strncpy`, which copies characters from one string to another.

We implemented the above approach to determine if it would be sufficient. In MPICH 3.3, there are 8,823 calls to non-void functions where the return value is not saved. Of these, 8,731 return an integer, and 92 return a pointer. The vast majority of these call sites are correct, i.e., the function being called cannot return an error code. In contrast, MPIErrorProp (Section 3) reports 321 buggy call sites with a 6% false positive rate (Section 4.1).

To generate the subset of call sites that must have their return value saved, MPIErrorProp tracks the propagation of error codes using dataflow analysis. If an error code can reach the return value of a function, then all calls to that function are labeled as buggy if the return value is not saved. Figure 2 shows an abstract, but representative, example. It illustrates two general patterns and therefore the particular functions represented by the nodes are unimportant. The dashed red edges $(10, 4)$, $(4, 3)$ and $(4, 9)$ represent possible error code propagation paths starting at the origin site in function 10. These flows exist prior to the application of any partial propagation fixes, and they will be detected by MPIErrorProp. The solid black edges represent calls. The solid black edge $(9, 4)$ indicates there exists at least one call in the body of the function 9 to function 4. If any of these calls do not save the return value, then they would be reported as a bug by MPIErrorProp.

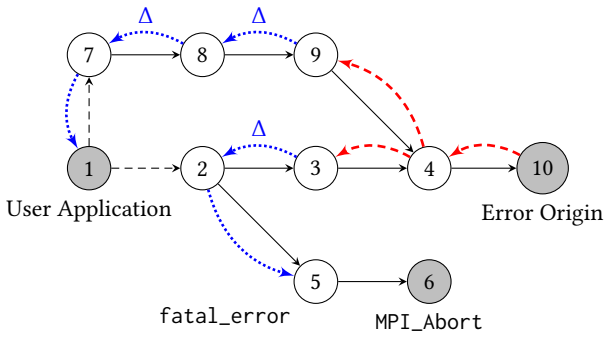

Figure 2. Propagation paths of an error code from an origin site. Each node represents a function. The call graph of the MPI implementation is represented by black solid edges (→). Dashed black edges (⇢) are potential calls from the application. Dashed red edges (⇢) represent existing error code propagation paths, and dotted blue edges (⇢) represent error code propagation paths that only exist after introduction a fix. Edges labeled Δ are partial propagation fixes; unlabeled dotted blue edge propagate the error code, but only after the application of a fix.

Functions 3 and 9 contain defective calls to 4. The call sites are defective because the error code originating in function 10 can reach them, as indicated by the ⇢ edge, and each fails to save the return value of 4 (individual call sites not shown). These would be reported as error code propagation bugs by MPIErrorProp.

A single iteration of the application of partial propagation fixes is represented by the ⇢ edges $(9, 8)$ and $(3, 2)$ labeled with Δ. Only after the fixes are applied do the calls to function 4 within functions 9 and 3 propagate the error code. After these fixes are applied, two distinct behaviors emerge:

1. Applying the $(9, 8)$ fix leads to another buggy call in the function represented by node 8. Subsequently applying the propagation $(8, 7)$ is sufficient to allow the user application to detect the original error.
2. Applying the $(3, 2)$ fix connects the error code propagation flow to an existing error handler that declares the error as fatal and calls abort.

In the second case, the error code does not reach the user application, but this is allowed by the MPI Standard. It is important to track the propagation of error codes up until the point that abort is called.

2.4 Overview of Our Approach

The high-level workflow of our approach is presented in Figure 3. We use a novel combination of static and dynamic analysis. We first identify potential error propagation bug locations via static analysis and program repair. The analysis proceeds in a loop, running the static analysis and fixing

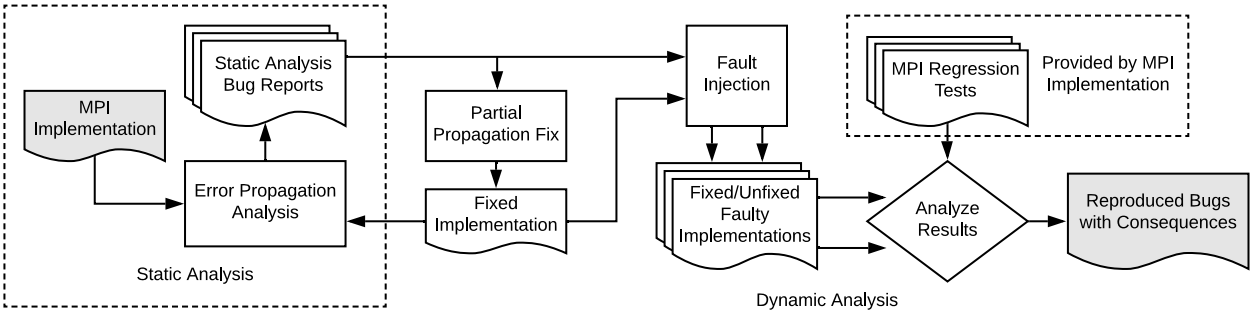

Figure 3. Workflow for detecting and reproducing error code propagation bugs in MPI implementations.

bugs until no new bug reports are produced. Programmers could manually look at these potential bug locations and fix the bugs; however, usually the number of bug reports is large and they may contain false positives. The second phase of the method performs dynamic fault injection to (1) verify the validity of the potential bug reports and (2) determine the consequences of the bugs (if the bug is reproduced).

3 Technical Approach

This section describes each of the components of our approach: a hybrid technique to find error propagation bugs (Section 3.1 - Section 3.3) and a fault injection technique (Section 3.4) to reproduce error propagation bugs (Section 3.5).

3.1 Static Analysis for Error Propagation

This section describes MPIErrorProp, an interprocedural, flow- and context-sensitive static analysis to track the propagation of error codes in MPI implementations. Given a set of integer error constants, MPIErrorProp finds the set of error values that each variable may contain at each program point in an MPI implementation. The analysis is formulated as a forward dataflow problem where error codes are propagated via variable assignments and function return values.

We define a set of constants C that consists of error codes (as defined in each MPI implementation, see Table 1 for examples), the special `MPI_SUCCESS` value used in MPI implementations to signal success, and the special analysis values *OK* to represent non-error values, and *uninitialized* to represent uninitialized variables. The analysis maps each variable v to a set of elements from $\mathcal{V} \cup C \rightarrow 2^{\mathcal{V} \cup C}$, where $\mathcal{V}$ is the set of program variables. In other words, the analysis determines the set of possible values a variable may hold after the execution of a given program statement.

MPIErrorProp dataflow transfer functions define a standard forward dataflow analysis. The transfer functions encode direct assignments and function return values, but do not strictly overapproximate possible error code propagation paths. Deliberate trade-offs have been made to reduce false positives while maintaining scalability and precision, and therefore MPIErrorProp does not guarantee soundness.

Specifically, we define transfer functions for the various statements in the program. Transfer functions denote how program statements affect the values a variable may contain. For example, consider an assignment of the form $x = e$, where $e \in \mathcal{V} \cup C$. The transfer function for such an assignment is $Ident[v \mapsto \{e\}]$, i.e., after the assignment is executed, v must have the value of e while the values for all other variables remain unchanged. For example, the transfer function for the assignment on Line 225 in Figure 1b is $Ident[$`mpi_errno` $\mapsto \{$`MPI_SUCCESS`$\}]$. In the case in which $e \notin \mathcal{V} \cup C$, we assume that the value of such expression is not a valid error code, and thus $Ident[v \mapsto \{OK\}]$.

MPIErrorProp includes a *limited* form of path sensitivity that reasons about simple conditional predicates and prunes error values from variables on normal paths. Error codes are exclusively negative integers, and thus MPIErrorProp can declare that a variable does not hold error codes if the sign of the values is positive. Shown abstractly in Figure 4, this pruning improves both the scalability and precision of MPIErrorProp.

We define the transfer function for function calls in a standard manner: the values passed to the callee as arguments are copied from the caller to the callee's formal parameters. Similarly, the callee's return value is copied back into the caller's receiver variable (if any). In addition to these standard transfer functions, we include MPI-specific transfer functions to suppress the transformation of error codes to error classes. These transfer functions are described in more detail in Section 3.1.1.

As mentioned earlier, the goal of our analysis is to find the set of error codes that each variable may contain at each program point. To compute these sets, the transfer functions are applied until fixpoint, i.e., they are applied iteratively until no changes are observed in the sets of errors codes. The computed sets are used to identify dropped sites for functions that can return errors. The analysis operates on a program abstraction that captures control flow and encodes

```
int err = foo();
if (err < 0) {
  ...
} else {
  // err cannot hold error codes
}
return err;
```

Figure 4. MPIErrorProp performs a limited sign analysis to prune error codes from variables when the sign of the variable is positive.

```
* src/mpi/comm/commutil.c:247:
"MPI_ERR_OTHER" error is passed as argument
* src/mpi/errhan/errutil.c:861:
an unchecked error may be returned
* src/mpi/comm/commutil.c:247:
"mpi_errno" receives error from
function "MPIR_Err_create_code"
* src/mpi/comm/commutil.c:263:
an unchecked error may be returned
* src/mpi/comm/commutil.c:612:
receives an error MPIR_Comm_map_irregular
* src/mpi/comm/commutil.c:612:
error is not saved in function MPII_Comm_copy
```

Figure 5. MPIErrorProp trace for the bug in Figure 1.

transfer functions, which is constructed using LLVM. We use an existing library to compute the fixpoint.

Our LLVM frontend of MPIErrorProp takes as input a bitcode file of an MPI implementation and constructs the transfer functions. These transfer functions capture control flow, model some MPI-specific transformations (Section 3.1.1), and capture variable assignments. Individual error codes are provided as domain knowledge and rewritten to large negative values for the purpose of distinguishing them from other non-error constants used in the library.

After the transfer functions have been created by the LLVM frontend, MPIErrorProp uses the WALi WPDS library [11] as a backend to compute the analysis fixpoint, and produce bug reports. The WPDS dataflow framework is well-suited to the task at hand because it has the capability of producing a witness for a dataflow fact. The two ends of this witness trace (Definition 1) are used for the fault injection phase (Section 3.4) to study the consequences of MPIErrorProp bugs. These witness traces form the basis of MPIErrorProp bug reports. Each bug report trace represents a sample path originating where an MPI error code is first used, and ending where the error code is dropped.

Definition 1 (Trace)**.** An MPIErrorProp trace τ consists of a sequence of source locations $l_1^\tau, l_2^\tau, ..., l_n^\tau$.

An MPIErrorProp trace taken from the analysis of MPICH is shown in Figure 5.

Definition 2 (Origin Site)**.** The origin site of an MPIErrorProp trace τ is $\mathbb{O}_\tau \triangleq l_1^\tau$. This is the location where the error code is first used in the trace. In Figure 1b on Line 247 a macro that wraps `malloc` is the origin site.

Definition 3 (Dropping Site)**.** The dropping site of an MPIErrorProp trace τ of length n is $\mathbb{D}_\tau \triangleq l_n^\tau$. This is the call site of the last function to propagate an error. In Figure 1a, Line 612 is a dropping site for the error returned by `MPIR_Comm_map_irregular` because its returned value is not saved.

While we primarily study MPIErrorProp in this paper, the rest of the steps in the pipeline can be run with any source of bug reports, where each bug report is a pair of source locations with the semantics of origin site and dropping site defined above.

3.1.1 Issues Specific to MPI Implementations

There are three key characteristics that distinguish our work from previous analyses that track error codes in Linux (e.g., [8, 25]). First, our analysis does not require error-handling domain knowledge. *All* non-fatal errors must propagate in MPI implementations (with `MPI_ERRORS_RETURN`), while in Linux only *unhandled* errors are propagated to the user application. In this regard, error-handling in MPI implementations is more straightforward than in the Linux kernel because of the stronger propagation requirements. Second, reporting the first instance in which an error code fails to propagate is often insufficient to expose all error code propagation bugs in MPI applications, as discussed in Section 2.3. Third, the dataflow analysis defines transfer functions that are specific to MPI implementation functions. This section details these MPI-specific transfer functions.

In particular, the MPI Standard distinguishes between error classes and error codes, where error codes are implementation specific and error classes are defined by the MPI Standard for the purpose of making it possible for the error to be interpreted by the application.[3] Because of this, MPICH and MVAPICH provide helper functions for manipulating error codes, error classes, and the messages that go with each. These functions contain elaborate, implementation-specific rules for mapping error codes to error classes. One such function in MPICH is `MPIR_Err_create_code`, a function that creates MPI error codes.

Handling these transformations is a challenge for MPIErrorProp, which uses dataflow analysis to track the propagation of error codes, because the helper functions create a *new*

[3] https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node222.htm

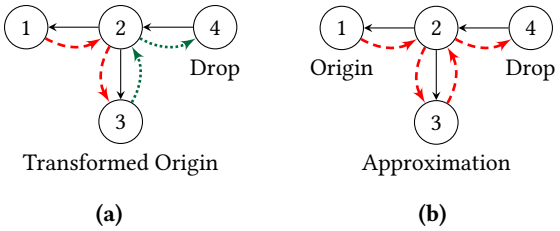

Figure 6. Figure 6a shows the impact of the transformation from error codes to error classes on error propagation witness traces. The red dashed arrows (⇢) show the flow of an error code. Function 2 calls a helper function 3 that transforms the error code into either an error class or another error code. The flow of this transformed value is marked by dashed green arrows (⇢). This transformation causes the origin of the witness trace to start inside function 3, rather than the true origin of the error in function 1. Figure 6b shows the flow of an error code with error code transformations suppressed. The helper function is called, but the error code remains unmodified, allowing MPIErrorProp to identify the true origin of the error.

error code. This causes the origin site of many bug reports to be inside the helper function that created the error code. To deal with this challenge, MPIErrorProp uses a model of the helper function that allows the actual origin site to be connected with the dropping site in the witness trace that defines a bug report. This function model is an approximation that suppresses transformations of error codes. Using the model has the possibility of introducing false positives on paths that check for a specific error code, but in practice we have not encountered any. None of the false positives described in our experimental evaluation (Table 2) are a result of this approximation.

The impact of the error code and error class transformation is illustrated in Figure 6. Figure 6a shows what an error propagation witness trace might look like without suppressing these transformations. The origin site of the actual value that is dropped is reported as a line inside function 3. While this is accurate from the perspective that the value being dropped was manufactured inside function 3, we are interested in the original source of the error, not merely where the transformed value was created. This is important because the origin site is used for fault injection (Section 3.4). Figure 6b shows the error propagation witness that MPIErrorProp reports by assuming that function 3 does not alter the error code. This allows function 1 to be identified as the true source of the error.

3.2 Synthesis of Partial Propagation Fixes

MPIErrorProp only reports unsaved error codes at the most immediate call site. Unfortunately, this is not sufficient for

```
1501  static int FreeNewVC( MPIDI_VC_t *new_vc ) {
1502    int mpi_errno = MPI_SUCCESS;
1523  -   MPIDI_CH3_VC_Destroy(new_vc);
1523  +   int ret = MPIDI_CH3_VC_Destroy(new_vc);
1524  +   if (ret != MPI_SUCCESS) return ret;
1525    MPL_free(new_vc);
1526  fn_fail:
1527    return mpi_errno;
1528  }
```

Figure 7. `MPIDI_CH3_VC_Destroy` may return an error that is dropped. Diff shows synthesized partial fix.

MPI implementations. Fixing a dropping site may still not result in the error propagating to the application; other dropping sites may surface as the error propagates further within the MPI library. MPIErrorProp automatically synthesizes error propagation fixes to either verify that the error reaches the application after the bug is fixed, or to identify new dropping sites that also require fixing.

The dropping site (Definition 3) of an MPIErrorProp trace is the location of the actual bug in the code. To fix the bug, the error code reaching that source location needs to be propagated to callers of the enclosing function. The fix strategy used in this work (shown as `PartialFix` in Algorithm 1) is to immediately return the error code at the dropping site if it is not `MPI_SUCCESS`. Figure 7 shows an example of an error propagation bug and its corresponding fix. Specifically, we declare a local variable to store the dropped error code, and return it immediately if it is not `MPI_SUCCESS`. These are partial fixes because we simply attempt to propagate the error; we do not synthesize other code that may be needed along propagation such as cleanup.

This is just one implementation of `PartialFix`, and there are cases where it does not succeed. Some functions in MPI implementations pass error codes via out parameters, in addition to or instead of return values. In the future, additional repair strategies could be added to handle different scenarios.

3.3 Hybrid Analysis for Error Propagation

Our approach combines static analysis and program repair to find error propagation bugs in MPI implementations. To the best of our knowledge, this is the first approach to integrate program repair as a central step in bug finding.

Figure 8 shows how multiple dataflow iterations with integrated repair are required to expose additional error code propagation bugs in MVAPICH. The function `iba_get` (Figure 8a) calls `Post_Get_Put_Get_List` on Line 2257 without saving the return value. This is reported as an unsaved error code by MPIErrorProp when run on the unmodified MVAPICH code. A partial fix for this bug is shown, which simply saves the value returned from `Post_Get_Put_Get_List` and

```
2211  int iba_get(...) {
2256    if(...) {
2257    -  Post_Get_Put_Get_List(...);
2257    +  mpi_errno = Post_Get_Put_Get_List(...);
2266    }
2268    return mpi_errno;
2269  }
```

(a) Partial fix for the error code propagation bug on Line 2257 of `rdma_iba_1sc.c` in MVAPICH.

```
 883  int MPIDI_CH3I_RDMA_try_rma(...) {
1017  case MPIDI_CH3_PKT_GET:
1038    iba_get(curr_ptr, win_ptr, size);
1039    MPIDI_CH3I_RMA_Ops_free_elem(win_ptr,...);
1040    curr_ptr = next_ptr;
1051    break;
1088  }
```

(b) The second iteration of MPIErrorProp reports an additional error propagation bug on Line 1038. Function `iba_get` now returns an error code that is not saved.

Figure 8. Example of a bug (Figure 8b) that is only exposed when another bug (Figure 8a) is fixed.

propagates it. However, the call to `iba_get` on Line 1038 (Figure 8b) also neglects to save the return value. This is not reported by MPIErrorProp in the first iteration because, without the fix, `iba_get` cannot return an error code other than `MPI_SUCCESS`. Our hybrid analysis automatically synthesizes and applies the bug fix, and invokes MPIErrorProp to uncover new dropping sites on the modified library.

Algorithm 1 describes our hybrid approach. Each iteration runs MPIErrorProp, then applies the propagation fixes for the new bugs reported. The `PartialFix` function is described in Section 3.2. It takes an MPI implementation and a set of bug reports, applies a fix for each bug, and returns a patched MPI implementation. MPIErrorProp is then run on the implementation returned by `PartialFix` to expand the propagation frontier and discover new bugs. This process continues until there are no new bug reports. The variable `Fixes` collects all of the fixes that led to new bug reports and applies them to the next iteration. Thus L' is a version of the library that has fixes applied from each previous iteration that led to additional bug reports. Within an iteration, MPIErrorProp is run on a version of the library that fixes a single bug (in addition to fixes from previous iterations). This is done so that MPIErrorProp can precisely report which individual bug fixes led to additional bug reports.

3.4 Fault Injection

A fault injection strategy is defined by what faults will be injected, where the faults will be injected, and when the injections will be triggered [24]. We use two strategies to

Algorithm 1: Hybrid Analysis for Error Propagation: MPIErrorProp combined with program repair.

Input: L ▷ An MPI implementation
Output: Bugs ▷ A set of bug reports

```
1  Function MPIErrorPropWithRepair(L):
2      AllBugs ← Bugs[0] ← MPIErrorProp(L)
3      i ← 0, Fixes[0] ← ∅
4      while True do
5          i ← i + 1, Fixes[i] ← Fixes[i − 1]
6          for b ∈ Bugs[i − 1] do
7              L′ ← PartialFix(L, Fixes[i − 1] ∪ b)
8              Bugs_b ← MPIErrorProp(L′)
9              if Bugs_b \ AllBugs ≠ ∅ then
10                 Fixes[i] ← Fixes[i] ∪ b      ▷ New bugs
11             Bugs[i] ← Bugs[i] ∪ Bugs_b
12         Bugs[i] ← Bugs[i] \ AllBugs
13         AllBugs ← AllBugs ∪ Bugs[i]
14         if Bugs[i] = ∅ then
15             return AllBugs                   ▷ No new bugs
```

study the failures of an MPI implementation. The Error Code Injection (ECI) strategy injects error codes directly at the origin site of a bug report, and the Memory Failure Injection (MFI) strategy injects memory allocation faults at the interface between the MPI implementation and `libc` memory allocation routines.

Definition 3.1 (Fault Injection Strategy)**.** A fault injection strategy is a 3-tuple $(v, L, trigger)$, where v represents the literal value that will be returned on error, L is a set of source locations at which the injection will occur, and *trigger* is a boolean trigger function.

Definition 3.2 (Call Stack Trace)**.** A call stack trace $\Psi = c_1, c_2, ..., c_n$ is a sequence of call sites in a program execution where c_1 is the first function call in the execution, and for all $1 \leq i < n$, the function called at c_{i+1} is executed before the function called at c_i returns. A call stack trace captures the nesting of function calls during program execution.

Both injection strategies utilize the same trigger function. Let $\mathbb{D}$ be a dropping site, and Ψ represent the current call stack trace at the time that the trigger function is called.

Definition 3.3 (Trigger Function)**.** $trigger(\mathbb{D}, \Psi)$ is a boolean function that is `true` if and only if $\mathbb{D} \in \Psi$.

Algorithm 2 shows the fault injection process for both the ECI and MFI strategies. The input is a single MPIErrorProp

bug report, and the output is a modified version of the MPI implementation designed to reproduce the bug.

3.4.1 Error Code Injection

Our implementation of the ECI strategy prepends a `return` instruction that is only executed when the trigger is activated. The ECI strategy is designed to exercise the recovery code that would execute after the MPI implementation encounters an error. In terms of the fault injection strategy 3-tuple (Definition 3.1), ECI can be defined as follows.

Definition 3.4 (Error Code Injection (ECI))**.** Given an error-propagation trace τ, the ECI fault injection strategy is

$$(\texttt{MPI\_ERR\_OTHER}, \{\mathbb{O}_\tau\}, \mathit{trigger}(\mathbb{D}_\tau, \Psi))$$

Intuitively, the error code `MPI_ERR_OTHER` is injected at the origin site of an MPIErrorProp trace by inserting a return instruction that is executed when the dropping site is on the call stack. Concretely, the injection is done by applying a patch to the implementation of the MPI library (Algorithm 2). Note that any error code can be injected, however we choose to inject `MPI_ERR_OTHER` because it is generic, and because it is commonly used to signal out of memory errors in the MPI implementations.

3.4.2 Memory Failure Injection

The ECI strategy is a general approach that works regardless of the type of error. It does, however, require the origin site to be covered by the program that is being executed. To handle additional bugs, we also inject at the more traditional location of the interface between the application and the library. This strategy is not as general, and is more restrictive in the locations where faults can be injected, but it has a higher level of fidelity as there is no possibility of executing infeasible paths. We target memory failures because we observe these to be a common error source in MPI implementations, and because such failures frequently lead to segmentation faults in C programs.

Definition 3.5 (Allocations)**.** *Allocations* is the set of all call sites to any function in {`malloc`, `calloc`, `realloc`}.

Definition 3.6 (Memory Failure Injection (MFI))**.** Given an error propagation trace τ, the MFI fault injection strategy is

$$(0, \mathit{Allocations}, \mathit{trigger}(\mathbb{D}_\tau, \Psi))$$

Intuitively, the MFI strategy injects a null pointer when the dropping site is on the call stack and any of the three memory allocator functions is called. Faults are injected by overriding the weak symbols `malloc`, `calloc`, and `realloc` directly. The same trigger (Definition 3.3) is used for the ECI strategy and the memory failure strategy.

Algorithm 2: Fault Injection

input : An MPIErrorProp bug report τ
output : Modified MPI implementation

```
1  Define variable t = 0;
2  Insert instruction t = 1; immediately before 𝔻_τ
3  Insert instruction t = 0; immediately after 𝔻_τ
4  if ECI Strategy then
5      R ← if (t) return MPI_ERR_OTHER;
6      Insert instruction R immediately before 𝕆_τ
7  else if MFI Strategy then
8      for f ∈ {malloc, calloc, realloc} do
9          Let f' be the original functionality of f
10         Redefine f as if (t) return 0; else f'();
```

3.5 Reproducing Error Propagation Bugs

Our motivation for reproducing error propagation bugs is twofold: (1) verify the validity of bug reports, and (2) determine the consequences of such bugs. This effort is driven using the regression tests that accompany the targeted MPI implementations. This is a common method [32] of achieving high coverage without resorting to random API testing. MPI regression tests use either the `MPI_ERRORS_RETURN` or the `MPI_ERRORS_ARE_FATAL` modes. When `MPI_ERRORS_RETURN` is used we rely on the test checking the return values of function calls. We found that the regression tests aggressively check function return values, even of infallible functions. Infallible functions never fail, and therefore always return `MPI_SUCCESS`. When `MPI_ERRORS_ARE_FATAL` is set, the MPICH library is guaranteed to notify when an error code is detected by aborting execution.

Given a regression test suite, we calculate its coverage of MPIErrorProp bug reports. That is, we identify bug reports for which at least one test triggers fault injection. In order for fault injection to be triggered, the test must execute the origin site of the bug while the dropping site is on the call stack. We refer to these bugs as candidate bugs.

We run the full regression test suite for each combination of candidate bug, fault injection strategy, and fix strategy, with the aim of exposing unexpected behavior. Unexpected behavior is any behavior other than propagating the error code to the user application or aborting execution. More specifically, unexpected behavior includes program crashes, test hangs, test failure, assertion failure, and silent failure. A bug is *reproduced* if at least one test exposes unexpected behavior. On the other hand, the injected fault is *detected* by the MPI library if at least one test propagates the error code to the application, or produces a fatal error. Note that a single

Table 2. Summary of bug reports. FP: false positives.

Library	Confirmed	Potential	FP	Total
MPICH	276	25	20	321
MVAPICH	416	17	23	456
Total	692	42	43	777
Total Unique	447	28	23	498

bug report may expose different behavior in different tests, and thus can fall into both categories. Finally, we repeat our methodology after applying fixes to identify discrepancies in behavior and verify that the cause of the unexpected behavior is actually the source location of the bug report.

4 Experimental Evaluation

This experimental evaluation is designed to answer the following research questions:

RQ1 Are error code propagation bugs prevalent in MPI implementations? (Section 4.1.)
RQ2 What are the consequences of error code propagation bugs in MPI implementations? (Section 4.2.)
RQ3 How effective is fault injection at reproducing error propagation bugs in MPI implementations? (Section 4.3)

We analyze MPICH 3.3 and MVAPICH 2.3.1. The MPIErrorProp frontend uses LLVM [16] to translate source code into an intermediate XML representation first introduced in [25] to capture control flow and encode transfer functions. We use the WALi WPDS library [11] as a backend to compute the analysis fixpoint, and produce bug reports.

All experiments were run on Amazon Web Services EC2 instances. MPIErrorProp takes on average six minutes of wall clock time to analyze MPICH and 10 minutes for MVAPICH on an AWS c5.9xlarge instance with 12GB of RAM. For each MPIErrorProp bug report, we performed fault injection using both the ECI and the MFI fault models (with and without partial fixes). Each experiment involved running the full regression test suite accompanying the implementation. Fault injection experiments take 20-30 minutes each, as that is how long the regression test suites take to run.

4.1 Error Propagation Bugs Found

Table 2 shows the results of our inspection of the bug reports generated by the MPIErrorProp hybrid approach. A total of 321 bug reports were generated for MPICH, containing 276 previously unknown bugs, all of which we have manually examined and confirmed. To date, 242 bugs have been reported to MPICH developers, who provided positive feedback on the validity of the bug reports. In MVAPICH we confirmed a total of 416 bugs. Because MVAPICH is derived from MPICH, there is some overlap in the bug reports. 171 out of the 416 bugs are unique to MVAPICH, and we are in the process of reporting them. In total 447 unique confirmed bugs were found between the two libraries.

```
129  int MPIR_Comm_split_impl(...) {
368   MPIR_Comm_map_irregular(..., &mapper);
371   for (i = 0; i < new_size; i++) {
372     mapper->src_mapping[i] = keytable[i].color;
373     if (keytable[i].color == comm_ptr->rank)
374       (*newcomm_ptr)->rank = i;
375   }
396  }
```

Figure 9. Failure to check the return value of `MPIR_Comm_map_irregular` in `MPI_Comm_split_impl`, called by the commonly used function `MPI_Comm_split`.

Potential bugs are cases where we could not conclusively determine that the bug is real, and it is not an obvious false positive either. A total of 28 unique bugs fall into this category. The MPICH potential bugs have been reported, but not yet confirmed by developers. In total there were 23 unique false positives between the two libraries. The false positives arise from cases where MPIErrorProp is unable to reason about infeasible paths or functions that propagate error codes both through return values and pointer parameters.

The classes of errors encountered include errors in response to acquiring or releasing locks, sending messages, low-level TCP operations, connection initialization, configuration errors, and input sanitization. Many of the bugs reported by MPIErrorProp are in commonly used MPICH functions, such as `MPI_Comm_split`, the most commonly used function for creating new communicators. For example, `MPI_Comm_split` calls `MPI_Comm_split_impl`, shown in Figure 9. `MPI_Comm_split_impl` calls `MPIR_Comm_map_irregular` on Line 368, but fails to check the return value. `MPIR_Comm_map_irregular` uses `MPIR_CHKPMEM_MALLOC` (see Section 2), returning the error code `MPI_ERR_OTHER` if `malloc` fails to allocate memory, which is dropped on Line 368. The call to `MPIR_Comm_map_irregular` allocates memory for `mapper->src_mapping`, and if it fails, then the subsequent index into `mapper->src_mapping` on Line 372 will cause a segmentation fault. `MPIR_Comm_map_irregular` is a commonly misused function in MPICH. It is directly called in eight locations but the return value is never checked.

Answer to RQ1: Error propagation bugs are prevalent in MPICH-based MPI implementations. We found a total of 447 unique, previously unknown bugs in MPICH and MVAPICH. Since MPICH is used as the base for many MPI implementations, we expect that other implementations are also impacted by these bugs.

Table 3. Consequences of Bugs. Each cell lists the number of unique MPIErrorProp bug reports that triggered the behavior under fault injection strategies ECI and MFI.

			Bug Consequences			
Library	Inj.	Bugs	Crash	Hang	Fail	Silent
MPICH	ECI	83	24	3	35	25
MPICH	MFI	66	26	4	23	21
MVAPICH	ECI	61	24	3	38	17
MVAPICH	MFI	70	23	4	28	2

4.2 Consequences of Propagation Bugs in MPI

To better understand the significance of the bugs discovered in Section 4.1, we used fault injection to study the behavior of the MPICH and MVAPICH implementations (see Section 3.4).

Table 3 shows the behavior of the MPICH and MVAPICH regression tests (1,137 tests for MPICH and 753 for MVAPICH) after fault injection. Column "Bugs" gives the total number of bugs (per library) eligible for each fault injection strategy. We identified four main consequences of these bugs: program crashes, program hangs, test failures, and silent failures. The table counts the number of unique MPIErrorProp bug reports that produced the indicated behavior in any test. A crash is either a segmentation fault (most common) or a floating point exception. A hang indicates that the regression suite timed out after 1 hour. A test failure indicates that the injected fault caused the test to fail in any other way, e.g., computing incorrect results. A silent failure indicates that despite having a fault injected, the test passed. Note that a single bug may manifest distinct behaviors across tests.

For example, in MPICH, 24 out of 83 of the error code injections, and 26 out of 66 memory failure injections, caused crashes. In total 31 unique bugs led to crashes. All 24 of the crashes caused by error code injection were reproduced in MVAPICH as well. The higher ratio of crashes for memory failure injections can be explained by the fact that most crashes are segmentation faults, and error code injections target a wider variety of runtime errors.

An example of a bug that causes a crash is shown in Figure 1a. This bug is triggered by calling the commonly used function `MPI_Comm_dup`. The bug results in a malformed communicator and a segmentation fault upon the use of `MPI_Comm_free`. Another example, a silent failure, is shown in Figure 7. On Line 1523 a call to `MPIDI_CH3_VC_Destroy` is made without saving the return value. When an error code is injected here, there is no observable failure in the regression tests. Other calls to `MPIDI_CH3_VC_Destroy` in the MPICH library do propagate the error code. The partial propagation fix that we synthesized is also shown in Figure 7. After the fix is applied, injected error codes are successfully detected, i.e., the error codes reach the application.

```
337  int MPIR_Comm_commit(MPIR_Comm * comm) {
446    mpi_errno = MPIR_Comm_create(...)
447    if (mpi_errno)
448      MPIR_ERR_POP(mpi_errno);
463    MPIR_Comm_map_irregular(...);
501  }
```

Figure 10. The error propagation bug on line 463 will only be triggered if the call to `MPIR_Comm_create` on line 446 succeeds. Thus fault injection is necessary to reliably reproduce this bug by causing a memory failure only under the calling context of `MPIR_Comm_map_irregular`.

In MPICH, 683 out of 1,137 tests crashed due to a segmentation fault in response to an injected error code. That is, **60.1% of the MPICH 3.3 tests are susceptible to a segmentation fault caused by neglecting to save an error code.** Thus we conclude that these are not merely code smells; the consequences of failing to propagate error codes are severe and impact the core of MPICH. We observe that functions which impact a large number of tests in the regression test suite, such as `MPI_Comm_split` and `MPI_Comm_dup` (Figure 1), are also important to production code.

Consequences for Real MPI Programs. We further investigated the consequences of error propagation bugs in MPI by studying three real-world MPI programs: Kripke [12], miniAMR [19], and miniFE [20]. Kripke is a scalable 3D deterministic particle transport code, miniAMR is an adaptive mesh refinement mini-application, and MiniFE is a proxy application for unstructured implicit finite element codes. These codes are widely used in the procurement of HPC systems and are part of the US DOE Exascale Computing Project (ECP) Proxy Apps Suite[4].

We found that all three programs are susceptible to segmentation faults in response to memory allocation failures when using the MPICH library and default inputs. The two MPICH bugs responsible for these crashes belong to the set of confirmed bugs summarized in Table 2. The first bug is triggered by calls to `MPI_Comm_split`, as described in Figure 9. The second resides in `MPI_Comm_commit` (Figure 10), causing crashes during finalization with `MPI_Finalize` (not shown).

The results of the fault injection experiments are summarized in Table 4. Kripke and miniAMR each call `MPI_Comm_split` during initialization. A null pointer returned by `malloc` caused the applications to crash with a segmentation fault. There is nothing that the applications could do to prevent this, as the segmentation fault occurs inside

[4]https://proxyapps.exascaleproject.org/

Table 4. Behavior of MPI programs under fault injection. *Crash* indicates the application encountered a segmentation fault. *Fatal Err* indicates that MPICH returned a fatal error. *Propagated* means that an error code reached the application.

	Unfixed bug		Fixed bug	
Application	Fatal	Return	Fatal	Return
Kripke	Crash	Crash	Fatal Err	Propagated
miniAMR	Crash	Crash	Fatal Err	Propagated
miniFE	Crash	Crash	Fatal Err	Fatal Err

MPICH before `MPI_Comm_split` returns. The miniFE program generates a segmentation fault in response to a memory allocation failure in `MPI_Finalize`. `MPI_Finalize` calls `MPIR_Comm_commit`, activating the bug in Figure 10. The miniFE crash is potentially more harmful because it occurs after time has been spent computing a result. Again, the bug is inside MPICH and is triggered through normal API use.

In all three programs, applying synthesized error code propagation fixes resulted in more graceful recovery. These programs run in fatal error mode by default. In this mode, for Kripke and miniAMR, MPICH is able to provide the error message "probably out of memory" and a partial stack trace showing the location of the error. When Kripke and miniAMR are modified to run in error returns mode, with fixes applied, an error code successfully propagates to the application where it could be handled. In the case of miniFE, MPICH encounters a fatal error when the partial propagation fix is applied irrespective of the error mode. With the fix, the fatal error not only provides more information than a segmentation fault, but is also detected earlier, before spending time computing the answer to the problem.

Answer to RQ2: Error propagation bugs cause severe consequences. In MPICH, 31 unique bugs caused crashes, 24 of which exist in MVAPICH. 60% of the tests in MPICH are susceptible to crashes, including tests for commonly used functions such as `MPI_Comm_dup` and `MPI_Comm_split`. We also demonstrate that three real-world programs are susceptible to crashes.

4.3 Error Propagation Bugs Reproduced

For each bug report covered by a test, we determined whether injecting a fault into the library exposed unexpected behavior (see Section 3.5). More specifically, each bug report is marked as "reproduced" or "detected". A bug is "reproduced" if at least one test case exposes unexpected behavior, and a report is "detected" if at least one test detects the error. Individual tests may select either the `MPI_ERRORS_RETURN` error-handling mode or the `MPI_ERRORS_ARE_FATAL` mode. In the case of `MPI_ERRORS_RETURN`, tests that detect error codes are expected to fail with an error message after checking the return value of a function call (an assumption which we have validated by inspecting the test suite). In the case of `MPI_ERRORS_ARE_FATAL`, tests that detect the error codes will abort with a distinct error message. Any other undesired behavior means that the bug has been reproduced. Note that a single bug report may expose different behavior in different tests, therefore a bug report can fall into both categories.

Table 5. Bug Reproduction Results. Column *Bugs* lists the number of bug reports considered for each injection strategy. Columns under *Before Fix* show the number of unique bugs reproduced (*Rep.*) and detected (*Det.*). Columns under *After Fix* give the corresponding numbers after the bug was fixed.

			Before Fix		After Fix	
Library	Inj.	Bugs	Rep.	Det.	Rep.	Det.
MPICH	ECI	83	82	20	57	43
MPICH	MFI	66	42	29	29	52
MVAPICH	ECI	61	58	11	32	22
MVAPICH	MFI	70	33	26	26	32

For a bug to be reproduced means that an error code passes through its origin and dropping sites producing undesirable behavior, but it does not confirm that the dropping site is indeed a defective source location. We also perform fault injection after fixing the bug. If the error introduced by fault injection is not detected before the fix, but it is detected after the fix, then there is indeed a bug at the dropping site.

Table 5 shows the number of MPIErrorProp reports that the test suite covered. For ECI at least one test must cover the origin site, and for MFI at least one test must trigger memory failure injection. There is room for improvement in the MPICH and MVAPICH regression tests. The table also shows the number of unique bugs reproduced and detected for each library per injection strategy. For example, before fixing, 20 MPICH reports had at least one test detect an error code injected via ECI, while there were 43 bugs detected after the fix. This shows that for 23 bug reports, dynamic analysis has confirmed that not only is the bug report valid, but that the suggested fix leads to correct behavior. In contrast, there is a decrease in bugs reproduced by ECI from 82 to 57, indicating that for 25 bug reports the fix has caused *all* tests that previously produced unexpected behavior to now detect the presence of an error code.

Answer to RQ3: In total 95 unique MPICH bugs were reproduced, of which 74 were reproduced in MVAPICH. As expected, applying fixes increased the number of detected bugs and decreased the reproduced bugs.

4.4 Threats to Validity

ECI injects one specific error code: `MPI_ERR_OTHER`. The reason why we fail to reproduce some bugs could be that other specific error codes may need to be injected. Our fault injection techniques require user applications. We consider the test suites of two MPICH-based implementations of MPI and three real MPI programs. A threat to validity is that we rely on the results reported by the tests when injecting faults. It is possible that tests may report success without checking whether the return value is indeed `MPI_SUCCESS`. We are aware of only one test in which this occurs. Another threat is that we rely on the MPICH environment variable `MPITEST_THREADLEVEL_DEFAULT` to set the thread level for the regression test suite. This sets the default thread level, but it may be overridden by individual tests. Regression tests that explicitly run in multi-threaded mode by calling `MPITest_Init_thread` with `MPI_THREAD_MULTIPLE` as a parameter may distort our fault model. Finally, we only report results for two MPI implementations, but we believe that error propagation bugs exist in other libraries, and that our approach could be instantiated to analyze them.

5 Related work

Bug Detection for Error Handling in C Programs. Gunawi et al. [8] present an error detection flow insensitive static analysis (EDP) to find dropped errors in Linux file systems and drivers. Later work [25, 26, 35] describe flow- and context-sensitive static analysis to track the propagation of error codes, also in Linux, finding a variety of dropped errors. None of the above analyses have been applied to MPI libraries, which differ from Linux in how error codes are transformed and handled. Other work [5, 6, 10, 27, 36] mines error-related specifications that can be used for bug detection in Linux and C libraries, but none of these have been used to find bugs in MPI libraries.

Bug Detection in MPI Programs. Previous work has proposed tools and techniques to detect bugs in MPI programs [13]. These approaches can be roughly grouped into three categories: *correctness checking tools*, *statistical dynamic approaches*, and *MPI library bug detection*. Correctness checking tools perform static and dynamic checks in an MPI program to detect incorrect use of MPI, which can lead to failures, such as deadlocks [9, 33] and message races [28]. Checks can be performed via online profiling or via the use of formal methods [31, 34]. Statistical methods such as [7, 14, 15, 23] detect bugs by first developing a model of the application's normal behavior. These use dynamic information to look for deviations from the normal behavior model. MPI bug detection approaches detect bugs in MPI implementations [4]. Most of these methods focus on detecting communication bugs, while our approach targets detecting error code propagation bugs within the MPI libraries.

Program Repair for Error Handling. Lawall et al. [17] detect and fix incorrect error checks in OpenSSL libraries. ErrDoc [30] detects and repairs error-handling bugs in C programs, including error code propagation bugs. ErrDoc patches are complementary to our work and could be added to our fix strategies. ErrDoc does not examine the severity of the bugs or make any attempt at dynamic reproduction. MemFix [18] fixes memory deallocation bugs, which could also be categorized as error-handling bugs in some cases. MemFix does not fix error propagation bugs.

Reproduction of Error-Handling Bugs. Library Fault Injection (LFI) [21, 22] tests error recovery code by injecting error codes at the interface between a binary and shared libraries. We inject at the source code level, easing developer comprehension of the injected faults. In addition, we inject error codes inside the target MPI implementation to discover internal error code propagation bugs.

6 Conclusion

Ensuring the correctness and reliability of MPI implementations is crucial for large-scale parallel applications. Error propagation bugs in MPI implementations can lead to errors being ignored and not reported to applications, and even to unexpected behavior, such as crashes. In the course of our work with MPICH and MVAPICH, we found that these bugs can elude even experienced programmers. We present a novel approach to automatically detect and reproduce error code propagation bugs in MPI implementations. Our technique combines static analysis with program repair and dynamic fault injection methods. Our evaluation uncovered 447 previously unknown bugs in MPICH and MVAPICH, many of which have serious consequences. Our study of MVAPICH, which is based on MPICH, shows that error propagation bugs in MPICH have spread to other MPI implementations. Our approach provides a practical method to detect, fix and reproduce such bugs.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1750983, the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-771758), and AWS Cloud Credits for Research.

References

[1] 2019. MPICH - A high performance and widely portable implementation of the Message Passing Interface (MPI) standard. https://www.mpich.org/. (2019).
[2] 2019. MVAPICH - MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. (2019).
[3] 2019. Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/. (2019).

[4] Zhezhe Chen, Qi Gao, Wenbin Zhang, and Feng Qin. 2010. FlowChecker: Detecting bugs in MPI libraries via message flow checking. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[5] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 466–476. https://doi.org/10.1145/3338906.3338960

[6] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 423–433. https://doi.org/10.1145/3236024.3236059

[7] Qi Gao, Feng Qin, and Dhabaleswar K Panda. 2007. DMTracker: finding bugs in large-scale parallel programs by detecting anomaly in data movements. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 15.

[8] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, Mary Baker and Erik Riedel (Eds.). USENIX, 207–222. http://www.usenix.org/events/fast08/tech/gunawi.html

[9] Tobias Hilbrich, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. 2010. MUST: A scalable approach to runtime error detection in MPI programs. In *Tools for high performance computing 2009*. Springer, 53–66.

[10] Yuan Jochen Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 472–482. https://doi.org/10.1145/2970276.2970354

[11] Nicholas Kidd, Thomas Reps, and Akash Lal. 2008. WALi: A C++ Library for Weighted Pushdown Systems. http://www.cs.wisc.edu/wpis/wpds/download.php. (2008).

[12] Adam J. Kunen, Peter N. Brown, Teresa S. Bailey, and Peter G. Maginot. 2018. Kripke. https://github.com/LLNL/Kripke. (2018).

[13] Ignacio Laguna, Dong H Ahn, and R Bronis. 2015. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, Feng Qin, Debugging high-performance computing applications at massive scales. *Commun. ACM* 58, 9 (2015).

[14] Ignacio Laguna, Dong H Ahn, Bronis R De Supinski, Saurabh Bagchi, and Todd Gamblin. 2012. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 213–222.

[15] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. 2011. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 50, 10 pages. https://doi.org/10.1145/2063384.2063451

[16] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. 75–88. https://doi.org/10.1109/CGO.2004.1281665

[17] Julia L. Lawall, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. 2010. Finding Error Handling Bugs in OpenSSL Using Coccinelle. In *EDCC*. IEEE Computer Society, 191–196.

[18] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *ESEC/SIGSOFT FSE*. ACM, 95–106.

[19] Mantevo. 2015. miniAMR reference proxy application. https://github.com/arm-hpc/miniAMR. (2015).

[20] Mantevo. 2017. MiniFE Finite Element Mini-Application. https://github.com/arm-hpc/miniFE. (2017).

[21] Paul Dan Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE Computer Society, 379–388. https://doi.org/10.1109/DSN.2009.5270313

[22] Paul Dan Marinescu and George Candea. 2011. Efficient Testing of Recovery Code Using Fault Injection. *ACM Trans. Comput. Syst.* 29, 4 (2011), 11:1–11:38. https://doi.org/10.1145/2063509.2063511

[23] Alexander V Mirgorodskiy, Naoya Maruyama, and Barton P Miller. 2006. Problem diagnosis in large-scale computing environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 88.

[24] Roberto Natella, Domenico Cotroneo, and Henrique Madeira. 2016. Assessing Dependability with Software Fault Injection: A Survey. *ACM Comput. Surv.* 48, 3 (2016), 44:1–44:55. https://doi.org/10.1145/2841425

[25] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 270–280. https://doi.org/10.1145/1542476.1542506

[26] Cindy Rubio-González and Ben Liblit. 2011. Defective error/pointer interactions in the Linux kernel. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 111–121. https://doi.org/10.1145/2001420.2001434

[27] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *DSN*. IEEE Computer Society, 1–12.

[28] Kento Sato, Dong H Ahn, Ignacio Laguna, Gregory L Lee, Martin Schulz, and Christopher M Chambreau. 2017. Noise injection techniques to expose subtle and unintended message races. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 89–101.

[29] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, et al. 2014. Addressing failures in exascale computing. *The International Journal of High Performance Computing Applications* 28, 2 (2014), 129–173.

[30] Yuchi Tian and Baishakhi Ray. 2017. Automatically diagnosing and repairing error handling bugs in c. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 752–762.

[31] Sarvani S Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M Kirby. 2008. ISP: a tool for model checking MPI programs.. In *PPoPP*. 285–286.

[32] Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. 2014. Evaluating Distortion in Fault Injection Experiments. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*. IEEE Computer Society, 25–32. https://doi.org/10.1109/HASE.2014.13

[33] Jeffrey S Vetter and Bronis R De Supinski. 2000. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 51.

[34] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A scalable and distributed dynamic formal verifier for MPI programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–10.

[35] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-Backed Program Analysis for Scalable Error Propagation. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 586–597. https://doi.org/10.1109/ICSE.2015.75

[36] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating precise error specifications for C: a zero shot learning approach. *PACMPL* 3, OOPSLA (2019), 160:1–160:30. https://doi.org/10.1145/3360586