Detecting MPI Usage Anomalies via Partial Program Symbolic Execution

Fangke Ye
Georgia Institute of Technology
yefangke@gatech.edu

Jisheng Zhao
Georgia Institute of Technology
jisheng.zhao@cc.gatech.edu

Vivek Sarkar
Georgia Institute of Technology
vsarkar@gatech.edu

***Abstract*—MPI is a message passing based programming model for distributed-memory parallelism that is widely used for programming supercomputers. However, debugging and verification of MPI programs is generally recognized to be a deep technical challenge. This challenge is further exacerbated by a recent increase in the use of nonblocking MPI operations for improved scalability, since incorrect use of these operations can introduce new classes of bugs related to data races.**

In this paper, we introduce a new debugging approach based on *partial symbolic execution* to identify anomalies in MPI usage. Our approach avoids the false positives inherent in static analysis, while still scaling to large programs. Further, our approach can be applied to incomplete programs and explore multiple execution paths, thereby leading to increased coverage compared with dynamic approaches. An experimental comparison with state-of-the-art tools for debugging MPI applications show that our approach delivers demonstrably better precision than a state-of-the art static tool (MPI-CHECKER) and a state-of-the art dynamic tool (MUST), without incurring excessive overheads.

***Index Terms*—MPI, Symbolic Execution, Anomaly Detection.**

I. INTRODUCTION

In general, a variety of verification methods is needed for identifying different classes of bugs in programs written using the Message Passing Interface (MPI), with different trade-offs relative to performance and precision [1]. Both static and dynamic verification tools have well-known limitations in their effectiveness of detecting bugs in distributed MPI programs. Dynamic approaches are unable to cover all parallel execution paths in practice, because their coverage is limited by the input sets used, and can thereby result in false negatives (missed bugs). Exploring large numbers of input tests is especially challenging for MPI programs because of the efforts needed to configure and execute programs for different inputs on distributed-memory parallel platforms. Many MPI programs are poorly parameterized, which makes it hard for HPC developers to downscale a program to smaller instances that can be used for dynamic debugging. On the other hand, static approaches introduce false positives (false alarms) due to their over-approximation of program execution paths. This is also especially challenging for MPI programs because of the conservative nature of static analysis approaches for parallel programs with distributed states. The fact that MPI is commonly used with weakly-typed imperative languages like C/C++ aggravates this problem because of the need to analyze low-level memory operations that may include pointer arithmetic and aliasing.

This challenge is further exacerbated by a recent increase in the use of nonblocking MPI operations for improved scalability, since incorrect use of these operations can introduce new classes of bugs related to data races. Though nonblocking communication APIs can provide significant performance benefits, overlapping computation with communication can also lead to increased synchronization errors in MPI applications. For example, a common scenario is when an MPI application reuses the message buffer that has been passed to an MPI communication call, immediately after the communication API invocation. In the case of nonblocking MPI calls, it is possible for the MPI call and the user code to simultaneously access the same message buffer without proper synchronization and thereby corrupt the message contents, which could lead to the program crashing or to incorrect results.

In this paper, we introduce a novel static analysis framework based on symbolic execution for detecting *MPI usage anomalies* in C/C++ applications. Following standard practice, we use the word, *anomaly*, "to refer to any abnormality, irregularity, inconsistency, or variance from expectations" [2]. As we will see, the anomalies detected by our approach represent true bugs/defects in many cases; in other cases, the anomalies may represent irregularities that could be viewed as coding style violations. Our implementation is based on extensions to the KLEE [3] symbolic execution engine that works on LLVM IR [4], an instruction set that abstracts away details of the target platform but still preserves pointer operations. By reasoning about the low-level memory operations in LLVM IR with symbolic execution, our approach can detect memory-related anomalies. It also produces fewer false positives than traditional static analysis approaches.

Most MPI-based applications are used in the high performance computing area, which focus on large applications that perform numeric computations and scientific simulations. This poses multiple challenges for symbolic execution, since it may not successfully reach later points in a program execution due to path explosion contributing to excessive time and memory overheads., e.g., when symbolically executing computationally intensive loop nests. Rather than applying symbolic execution to the whole program from the start, we demonstrate that starting symbolic executions from carefully selected intermediate program points, which we refer to as *partial symbolic execution*, can be a key enabler of efficient anomaly detection for MPI programs.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

The specific contributions of this paper are as follows:

- A static verification framework for MPI programs that uses partial symbolic execution;
- A tool built on top of our framework that is capable of detecting several MPI usage anomalies;
- An evaluation of our approach on real-world applications, compared with state-of-the-art static and dynamic MPI verification tools.

As an example of real-world bugs found by our approach, Figure 1 outlines the buggy code identified by our approach in commit 21c4d95 of the 63KLOC ATHENA astrophysical magnetohydrodynamics application [5]. The code in Figure 1 has two anomalies, which were found by our approach: 1) a REQUEST OVERWRITING anomaly due to two calls to `MPI_Isend()` using the same request object, and 2) a BUFFER DATA RACE anomaly due to a race condition between the call to `MPI_Isend()` and the write of `send_buf[0]` via the pointer-based lvalue, `*(pSnd++)`. These two anomalies were confirmed by the developers of this code to be real bugs since they later added fixes to these anomalies in the form of calls to `MPI_Wait()` in commit 4ca1615.

Additional details on this and other buggy MPI examples can be found in Section IV-B. As reported in Table III in that section, the lightweight MPI-CHECKER *static analysis* tool [6] did not report the above two bugs for Athena. While checking of BUFFER DATA RACE bugs is not supported by MPI-CHECKER, REQUEST OVERWRITING bugs are checked by MPI-CHECKER but were not found in this example thereby resulting in a false negative. This is not a surprise since many static analysis tools cut back on soundness as a practical way to reduce false positives, and apparently, this was the case when MPI-CHECKER was analyzing the Athena application. Note that static analysis can be challenging for this example due to the need for interprocedural analysis of the call to `pack_ix2(pGrid)` to identify these bugs.

Further, the MUST dynamic analysis tool [7] did not identify the BUFFER DATA RACE bug in the Athena application because its checking of races is confined to overlaps between buffers that occur in MPI calls. To identify the race condition in Figure 1, MUST would need to check all loads and stores that occur in the program (as is done by a data race detector) rather than just the buffer usage in MPI calls, which would add significant runtime overhead. In summary, the above discussion of the Athena application motivates the use of our approach for increased precision, relative to state-of-the-art static and dynamic analysis tools. Further details can be found in Section IV-B.

II. MOTIVATION

In this paper, we focus on detecting lower level MPI anomalies that relate to communication and buffer operations. Since MPI APIs are not expressed as structured program constructs, it can be easy for users to omit necessary synchronization calls needed to match nonblocking calls. Also, MPI applications are often written in C/C++ programs that manipulate communication buffers with pointer arithmetic;

```
1  void bvals_mhd(DomainS *pD) {
2    ...
3    ierr = MPI_Isend(send_buf[0], cnt, MPI_DOUBLE,
4                     pGrid->lx1_id, RtoL_tag,
5                     pD->Comm_Domain, &(send_rq[0]));
6    ...
7    pack_ix2(pGrid);
8    ierr = MPI_Isend(send_buf[0], cnt, MPI_DOUBLE,
9                     pGrid->lx2_id, RtoL_tag,
10                    pD->Comm_Domain, &(send_rq[0]));
11   ... }
12 static void pack_ix2(GridS *pG) {
13   double *pSnd = send_buf[0];
14   ...
15   *(pSnd++) = pG->U[k][j][i].d;
16   ... }
```

Fig. 1. Example of buggy code in commit 21c4d95 of ATHENA.

thus it is important to precisely analyze the low-level pointer arithmetic to avoid the false positives that are common in static analysis tools. Though FORTRAN programs are more strongly typed than C/C++ programs, many of these challenges apply to FORTRAN programs as well with respect to the need for analyzing index arithmetic for array subscripts.

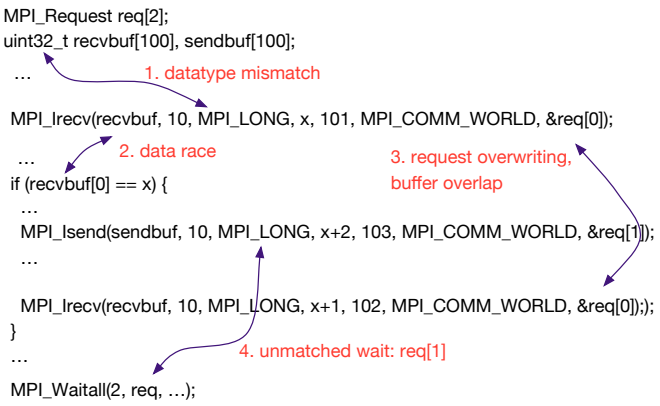

Fig. 2. Examples of anomalies in the usage of nonblocking MPI APIs

Figure 2 illustrates different types of MPI anomalies to motivate our proposed approach for detecting anomalies in MPI usage, especially in the use of nonblocking APIs. We summarize the five anomaly types targeted by our approach below.

1) Buffer Type Mismatch: Case 1 illustrates a buffer type mismatch anomaly, in where there is a mismatch between the MPI type argument and the type of the buffer variable [6]. This kind of anomaly can lead to data corruption or illegal memory accesses on platforms where there are differences in the sizes of the two mismatched types.

2) Buffer Data Race: Case 2 shows a data race due to the use of a nonblocking MPI_IRECV call. The reason for this anomaly is the lack of a corresponding test/wait MPI call to confirm that it is safe to access the buffer operand of the nonblocking MPI call. In the absence of this confirmation, there can be a data race on the buffer, either with a regular

memory access [8] or with another MPI call [7], which can lead to nondeterministic results in different executions of the program.

3) Request Overwriting: Case 3 illustrates a request overwriting anomaly [6]. In this case, a request handle of an ongoing nonblocking MPI operation is reused by a later MPI operation without confirming that the previous one has completed. This anomaly may lead to undesired outcomes, depending on the implementation of the MPI library.

4) Unmatched WAIT *or* TEST*:* Case 4 shows an unmatched wait/test anomaly [6], which occurs when an MPI request handle is used by a wait or test call before it is initialized. A request handle should be initialized by a nonblocking call, or set to MPI_REQUEST_NULL, before it is used by wait or test calls. This anomaly may lead to an invalid request error or other undesired outcomes, depending on the implementation of the MPI library.

5) Unmatched Point-to-point (P2P) Call: Another common kind of anomaly in MPI programs, not shown in Figure 2, is a mismatch between point-to-point (P2P) communication API calls. For example, this can occur if a SEND call does not have a matching RECV call in its target process or a RECV call does not have a matching SEND call in its source process. This anomaly can appear in both blocking and nonblocking MPI calls, and can result in deadlock or other undesired outcomes.

III. Checking for Anomalies

This section presents our approach to using symbolic execution to detect anomalies in MPI programs.

A. Background on Symbolic execution and KLEE

Symbolic execution [3] was introduced in the 1970's. Its use is increasing in popularity for testing and debugging software systems. A key advantage of symbolic execution is that it can simulate program behavior and generate test cases in arbitrary contexts, thereby resulting in increased precision compared with other static analysis approaches. Compared with dynamic approaches, symbolic execution introduces flexibility because it does not require concrete program inputs. However, one of its major limitations is that can incur intractably large memory and time overheads on large programs, making it impractical to use on full-size real-world applications [9]. As we will see, we address this limitation by introducing the use of partial symbolic execution for MPI programs.

KLEE [3] is a state-of-the-art symbolic execution tool built on top of the LLVM [4] compiler infrastructure, that has been widely used in various software verification systems. It is especially effective for verification of small-scale system software components since it works on a low-level intermediate representation that can simulate memory operations precisely and can enable flexible abstractions that helps the user identify different types of program errors.

B. Overall Architecture

Figure 3 summarizes the architecture of our MPI anomaly detection framework based on static pre-analysis and partial symbolic execution. The C/C++ source code is translated to LLVM bitcode [4] via the CLANG compiler front-end [10] . Our anomaly detector is built on the KLEE LLVM-level symbolic execution engine. We add to it MPI anomaly checking capabilities by intercepting a selected set of instructions during symbolic execution and maintaining extra information within the execution states (details in Section III-C). Before invoking our anomaly detector with the KLEE symbolic executor, we perform a static analysis to collect program information (details in Section III-D) which is used by the anomaly detector during symbolic execution.

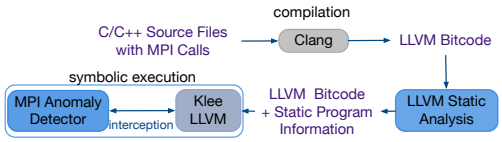

Fig. 3. Anomaly detection framework based on static pre-analysis and symbolic execution.

C. Simulating MPI Programs with Symbolic Execution

In this section, we describe our approach to symbolically executing MPI programs with the goal of anomaly detection. As MPI programs are intended to run with multiple processes and different processes may have different behaviors, it is necessary to take all ranks into consideration. Instead of launching multiple symbolic executions for different ranks of the program, we use one symbolic process to simultaneously represent the state of all processes. This is achieved by using a symbolic rank variable to represent a set of MPI processes in a single combined state. When encountering a branch condition with a value that depends on the process rank, the state may be forked into two states to represent those two sets of processes by conjoining the path conditions with constraints on the value of the rank value. This enables reasoning about the states of multiple processes in a single execution[1]. While not required for the core symbolic execution, the number of processes is also supplied as a concrete value to establish bounds on the rank variables.

During symbolic execution, each instruction is executed symbolically as in sequential programs, except for MPI calls. Instead of symbolically executing an implementation of the MPI library, we intercept the MPI API calls and simulate them separately in a lightweight manner. Actual data communications are not simulated due to the use of a single symbolic state to represent the concrete states of multiple processes. To simulate the case that the received data can impact the control flow, we fill the receive buffers with unbounded symbolic values. This interception mechanism assumes by default that MPI API calls are always executed successfully. For MPI calls like MPI_TEST, MPI_TESTALL, MPI_TESTANY

[1]For simplicity, our implementation currently only supports the MPI_COMM_WORLD communicator. Support for additional communicators is a subject for future work.

and MPI_WAITANY, we fork the execution states to handle all possible cases. For the sake of efficiency, we treat MPI_TESTSOME and MPI_WAITSOME as MPI_TESTALL and MPI_WAITALL, so as to avoid creating forks for every possible subset for the Testsome/Waitsome operations. Doing so may result in some false negatives, but not false positives.

Unlike blocking communication operations, the MPI nonblocking communication operations are initiated and completed by different MPI API calls, e.g. MPI_IRECV and MPI_WAIT. We add the following information to the symbolic execution state to model nonblocking calls. Inside each execution state, we maintain two lists of nonblocking communication operation records. Each record contains the address of the request object identifying the nonblocking operation and the base addresses and sizes of send and receive buffers. The two lists are:

- **Active list ($\mathbf{Act}_L$)**: The list of records for non-blocking operations that have been started but not guaranteed to be completed; and
- **Inactive list ($\mathbf{InAct}_L$)**: The list of records for nonblocking operations that are guaranteed to be completed by an MPI TEST call but not a WAIT call.

Algorithm 1: Simulating MPI Nonblocking Operations in the Symbolic Executor

```
1  function SymbExec ()
   Inputs : entry: input program point, Act_L: active list, InAct_L:
            inactive list
   Output: Updates to Act_L and InAct_L
2  inst := entry;
3  while inst ≠ NULL do
4      StateTransfer (inst);
5      NBStateTransfer (inst, Act_L, InAct_L);
6      inst := NextInst (inst);
7
8  function NBStateTransfer ()
   Inputs : inst: input program point, Act_L: active list, InAct_L:
            inactive list
   Output: Updates to Act_L and InAct_L
9  if IsNonBlocking (inst) then
10     Act_L ∪= inst;
11 else if IsTest (inst) then
12     nbInst := GetMatched (inst, Act_L);
13     if nbInst ≠ NULL then
           //Simulate false case
14         (Act_L', InAct_L') := Fork (Act_L, InAct_L);
15         SymbExec (NextInst (inst), Act_L', InAct_L');
           //Simulate true case
16         Act_L \= nbInst; InAct_L ∪= nbInst;
17 else if IsWait (inst) then
18     nbInstAct := GetMatched (inst, Act_L);
19     Act_L \= nbInstAct;
20     nbInstInAct := GetMatched (inst, InAct_L);
21     InAct_L \= nbInstInAct;
```

Algorithm 1 summarizes how we simulate MPI nonblocking execution in our symbolic executor. Function SYMBEXEC starts the symbolic execution at a given program entry point and keeps interpreting instructions until reaching the end of the current execution path. STATETRANSFER handles the standard symbolic state management. NBSTATETRANSFER processes the state transfer for nonblocking MPI APIs. When a nonblocking MPI API is invoked, a record for the nonblocking operation is added to $\mathbf{Act}_L$ (lines 9,10). If an MPI_TEST call is invoked to examine whether an operation in an issued state has completed, we fork the current state into two states for different results for this call. For the case in which the TEST call is assigned a false result, a new symbolic execution is performed with forked state ($\mathbf{Act}_L\prime$, $\mathbf{InAct}_L\prime$). For the case in which the TEST call is assigned a *true result*, indicating that the operation has completed, we move it from $\mathbf{Act}_L$ to $\mathbf{InAct}_L$ (line 16). If an MPI_WAIT call is invoked for a nonblocking operation, then it will be removed from both lists (lines 18$\sim$21). The GETMATCHED function here matches a nonblocking operation from the given list, if there are multiple matched operations, it only picks up the last one.

D. Static Analysis Pre-pass

As mentioned earlier, the anomaly detector runs as a plugin with the KLEE symbolic executor. Since the symbolic executor interprets the input program in execution order, it can only look "backwards" at the recorded information, but cannot look ahead for information from program points that have not as yet been processed. To address this limitation, we employ a lightweight static analysis as a pre-pass to collect whole program information that can assist the symbolic executor by providing it with look-ahead information. The following sections indicate how this look-ahead information can be used to enable partial symbolic execution, e.g., how the pointer analysis enables the lazy initialization introduced in Section III-E1.

This static analysis prepass includes both intraprocedural and interprocedural analyses. The interprocedural analysis is performed to construct program structure information, in particular, the program's call graph. The interprocedural analysis collects MPI-related control flow information along with supporting analyses:

1) A whole-program flow and context-insensitive pointer analysis;
2) An analysis of MPI derived types based on pointer analysis;
3) For each MPI call C, its unique control dependence predecessor branch's condition expression CDPEXPR(C);
4) For each branch expression EXPR, its control dependence successors [11] that are MPI calls and branch expressions. These successors are divided into two sets: S_{true} for those reached via the *true* branch and S_{false} for those reached via the *false* branch;
5) The loop structure, including the locations of loop preheaders, headers, and exits.

Items 3 and 4 essentially construct a special control dependence tree in which each node is either a function call or a branch expression. By leveraging call graph information, this tree can be built across procedure boundaries, i.e., a copy of

the nodes from callee function *F* can be connected to each branch expression that is the control dependence parent of a call site that has a unique target function *F*.

E. Partial Symbolic Execution

The previous sections introduced the basic approach for using symbolic execution to detect anomalies in MPI usage, especially for nonblocking operations. The cost of running a symbolic executor through an entire program starting from its main function is usually prohibitively large due to the combinatorial explosion in the number of symbolic states. In this section, we present our partial symbolic execution (PSE) approach that supports the start of symbolic execution from a user-specified program point. There are two major techniques related to PSE, lazy initialization and shadow memory allocation.

1) Lazy Initialization: Since a key requirement for PSE is to start symbolic execution from any function, a major challenge is how to initialize global/heap states that are expected to be defined on entry to that function? We address this challenge by employing *lazy initialization* for heap states. This approach is an extension of the technique presented in [12] for heap states. Since a key focus of our work is to handle pointer references that impact anomaly detection, we use on-demand memory allocation to assign values to pointer references that have unbounded values in PSE. When PSE starts from a function **F**, all global variables and arguments of **F** are initialized with unbounded symbolic values. If an unbounded pointer-typed value is dereferenced, there are three steps to be performed[2]: 1. allocate the memory block; 2. mark the new bytes as unbounded symbolic values; and 3. mark the pointer-typed values as bounded and constrain it to be equal to the starting address of the newly allocated memory block.

To handle pointer references reached during PSE, we consider three cases: 1. a newly created object; 2. reuse of an object that has been allocated; and 3. NULL reference. (Details are provided in function **LazyInit** in Algorithm 3 in Appendix B-A.) As discussed in the previous section, our approach includes a pointer analysis in the static pre-analysis. **LazyInit** takes the pointer alias information obtained from the pointer analysis as input. If a pointer reference P is not initialized, **LazyInit** checks if P has an initialized 'Must Alias' pointer reference Q. if so, we constrain P to point to the memory object allocated for Q. Otherwise if there is no 'Must Alias' pointer reference for P, **LazyInit** forks the current state into two states. In one of the states, it allocates a new memory object with each byte set as an unbounded symbolic value. In the other state, it nondeterministically selects P's referent from the set that includes NULL and the memory objects allocated for P's 'May Alias' pointer references.

2) Shadow Memory: To accommodate states for lazy initialization, we employ the idea of a shadow memory to track the boundedness of values. When a pointer gets dereferenced, we need to check if it needs lazy initialization based on the fact that the pointer variable has an unbounded symbolic value. The information regarding whether a pointer value is unbounded or not is maintained by the shadow memory associated with it.

General issues related to shadow memory:

Let N be the size of a pointer in bytes. We create a shadow memory object for every N bytes in the heap, stack and data segment. The shadow memory object stores a pointer to the metadata storage for the N bytes. For the purpose of tracking unbounded pointers, only one bit of metadata is used to indicate whether or not the N bytes store an unbounded pointer value.

The shadow objects are tied to bytes in memory, and the metadata are tied to values. Different bytes in memory can be associated with the same metadata but not the same shadow objects. In this paper, we make two assumptions regarding memory and pointer operations: 1. the pointers stored in memory are aligned, which means it is safe to use one shadow memory object for every N bytes. 2. no negative offset value is involved in pointer arithmetic instructions. We now briefly summarize the constraints for handling shadow memory during symbolic execution:

- The metadata bits of global variables and the entry function's arguments are initialized as *true* (indicating an unbounded value) at the start of PSE;
- For the load instruction case when a value is loaded from a memory address to a register, the address of the corresponding metadata object is also loaded from its shadow memory and saved into the register;
- For the store instruction case when a value is stored from a register to a memory address, the address of the metadata object is also saved from the register to the corresponding shadow memory;
- When a pointer value stored in a register is dereferenced, its corresponding metadata is loaded from the address saved in the register;
- If the metadata shows that an unbounded symbolic value is dereferenced, lazy initialization is applied and the metadata bit is set to *false*;
- For CAST or ASSIGNMENT operations, the destination register receives the metadata address saved by the source register;
- For a pointer arithmetic instruction, lazy initialization is performed when the base pointer address is unbounded, and the result value's metadata address is set to *null* (i.e., the result is not an unbounded pointer);
- For instructions that do not contain pointer-related operations, their result value's metadata address is set as *null*.

Implementation specific issues and optimizations:

In C/C++ languages, a pointer may point to either a concrete object (i.e., primitive type) or a composite object (e.g., an array or struct/class). For a composite object (e.g., an array) whose initial size cannot be identified during lazy initialization, we have to use a default size, which can be set by the user, to allocate the object. For handing a pointer offset value in a

[2]Details are provided in function **NewObject** in Algorithm 3 in Appendix B-A.

pointer arithmetic-related instruction, we choose a size that makes the constraint "OFFSET $<$ LENGTH" satisfiable.

To optimize the overall performance, we reduced the number of loop iterations visited by symbolic execution. As mentioned in Section III-D, the static analysis identifies the preheaders, headers, exits and back edges of loops in the program. During PSE, the symbolic executor maintains a stack of loops and an iteration counter for each loop. A loop **L** is pushed on the stack when the symbolic executor reaches **L**'s preheader. **L**'s iteration counter is incremented at its header. When the symbolic executor reaches **L**'s exit and **L**'s iteration counter exceeds a user-specified threshold, the exit branch is taken and **L** is popped from the stack.

F. MPI Anomaly Detection

TABLE I
MPI ANOMALY DETECTION RULES

(1)		buffer type mismatch
mismatch(x_i)	:-	MPI_CALL(x_i , T_i), ¬MATCH(TYPE(x_i), T_i)
(2)		buffer data race (overlap)
$Act_L \supseteq$ {buf_x,len_x, R}	:-	NONBLOCKING_READ(buf_x, len_x)
$Act_L \supseteq$ {buf_x,len_x, W}	:-	NONBLOCKING_WRITE(buf_x, len_x)
data race	:-	READ(buf_y, len_y), ∃ {buf_x, len_x, W} ∈ Act_L, OVERLAP({buf_x, len_x}, {buf_y, len_y})
data race	:-	WRITE(buf_y, len_y), ∃ {buf_x, len_x, W} ∈ Act_L, OVERLAP({buf_x, len_x}, {buf_y, len_y})
data race	:-	WRITE(buf_y, len_y), ∃ {buf_x, len_x, R} ∈ Act_L, OVERLAP({buf_x, len_x}, {buf_y, len_y})
(3)		request overwriting
$Act_L \supseteq$ r	:-	MPI_REQUEST(r)
$Act_L \setminus$=r, $InAct_L \setminus$=r	:-	MPI_WAIT(r)
$Act_L \setminus$=r, $InAct_L \cup$=r	:-	MPI_TEST(r)
request overwrite	:-	MPI_REQUEST(r_i), $r_i \in Act_L$
(4)		unmatched WAIT or TEST
$Act_L \supseteq$ r	:-	MPI_REQUEST(r)
$Act_L \setminus$=r, $InAct_L \setminus$=r	:-	MPI_WAIT(r)
$Act_L \setminus$=r, $InAct_L \cup$=r	:-	MPI_TEST(r)
unmatched wait	:-	MPI_WAIT(r_i), $r_i \notin Act_L$, $r_i \notin InAct_L$
unmatched test	:-	MPI_TEST(r_i), $r_i \notin Act_L$, $r_i \notin InAct_L$
(5)		unmatched P2P call
$OPC_S \leftarrow$ OPPSITE(E, $call_i$)	:-	E = CDPEXPR($call_i$) HASMPIRANK(E)
MATCHEDPAIRS ← ($call_i$, c)	:-	∃ c ∈ opc_S, RANK(c) = RANK($call_i$), ISCOMMPAIR(c, $call_i$)
$call_i$ has unmatched p2p call	:-	MATCHEDPARIS = ∅

As mentioned in the previous sections, our MPI anomaly detector intercepts the KLEE symbolic executor's instruction handling and maintains extra states for checking MPI-related anomalies. Algorithm 2 presents the anomaly detection mechanism that runs together with nonblocking state transfer in the symbolic executor (lines 11∼15). The auxiliary analysis was applied before the symbolic execution (line 2). The anomaly detection function checks each intercepted instruction and updates the MPI-related states. There are five anomalies handled by the implementation developed for this paper. The exact anomaly detection rules are summarized in a Datalog-like representation in Table I), with further details discussed below.

Algorithm 2: Anomaly Detection via Symbolic Execution

```
1  function MainProc ()
     Inputs : P: input program, F: start function
     Output:
2    InterProcAnalysis (P);
3    startInst := GetFunctionEntry (F);
4    Act_L := ∅; InAct_L := ∅;
5    anomaly_S := SymbExec (startInst, Act_L, InAct_L);
6    if anomaly_S ≠ ∅ then
7      Alarm (anomaly_S);
8
9  function SymbExec ()
     Inputs : entry: entry point, Act_L: active list, InAct_L: inactive
              list
     Output: anomaly_S: the set of anomalies
10   inst := entry;
11   while inst ≠ NULL do
12     StateTransfer (inst);
       //Check anomalies on current instruction
13     anomaly_S ∪= AnomalyDetect (inst, Act_L, InAct_L);
14     NBStateTransfer (inst, Act_L, InAct_L);
15     inst := NextInst (inst);
16   return anomaly_S;
17
18 function AnomalyDetect ()
     Inputs : inst: current instruction, Act_L: active list, InAct_L:
              inactive list
     Output: anomaly_S: the set of anomalies
     /*Apply anomaly detect rules from Table I
       and return collected anomalies          */
```

Buffer Type Mismatch: Rule (1) in Table I checks the type of the buffer arguments in a MPI call against the corresponding MPI_DATATYPE handles. We maintain a map from instances of MPI_DATATYPE to types in LLVM IR used by KLEE, including the MPI primitive types and MPI derived types (a derived type detection algorithm is presented in Algorithm 4 in Appendix B-A). A given buffer's type is checked against the type obtained from the map using the MPI type handle as the key.

Buffer Data Race: Rule (2) in Table I identifies data races in nonblocking MPI calls that may read/write buffers. For each memory read, it checks if it accesses a RECV buffer from an ongoing nonblocking MPI call. Similarly, it checks if each memory write accesses a SEND or RECV buffer from an ongoing nonblocking MPI call.

The ongoing nonblocking calls are stored in the active list (Act_L, defined in Section III-C), which is maintained inside every execution state in KLEE. The code for checking this anomaly is inserted into KLEE's memory operation execution function so that each time there is a read/write operation, the data race check is executed.

Request Overwriting: Rule (3) in Table I describes how to find an overwritten request handle. When a request handle is

passed to a nonblocking call, we check if it is used by an ongoing nonblocking call to decide whether there is a request overwriting anomaly.

Since ongoing nonblocking calls are recorded in Act_L maintained by KLEE, we implement the check by comparing the address of the request handle with that of each nonblocking call in Act_L.

Unmatched WAIT or TEST: When processing WAIT or TEST operations (shown in the rule (4) of Table I), our anomaly detector checks if a request handle in the arguments is used by a MPI nonblocking call that has not been waited on as yet.

The check is performed by looking up the address of the request handle in both Act_L and $InAct_L$ stored for the current KLEE execution state. If there is no matching call found in either list, an unmatched WAIT or TEST anomaly is reported.

Unmatched Point-to-point (P2P) Call: Based on our restriction of simulating multiple processes in a single symbolic execution, the matching of P2P communication API calls cannot be performed precisely. Since MPI applications often use control flow branches with condition expressions that contains rank values to distinguish code running on different processes, if KLEE can check both *true* and *false* cases for these rank value-related control flow branches, then it can also identify some of the matched P2P APIs (e.g., SEND/RECV pairs).

As mentioned in Section III-D, we employ static analysis to establish the connection between P2P calls with branch expressions (i.e. CDPEXPR) that contains rank values. For a given P2P call, we check if there is a corresponding sender or receiver from the code region that is control dependent on the same branch (shown in the rule (5) of Table I).

IV. EVALUATION

A. Experimental Setup

We created a prototype implementation of our approach using LLVM 3.6.2. We evaluated our tool on a Linux workstation with an Intel Core i7-3770K processor running at 3.50GHz with 16GB of RAM. We evaluated our approach with C programs, since there is limited support for C++ in KLEE.

We collected 6 MPI programs as our benchmarks. Two of them are real-world applications: AMG2013 and ATHENA, that are approximately 75KLOC and 63KLOC in size respectively. NPB.IS and NPB.DT are two benchmarks from the NPB benchmark suite [13]. The remaining two benchmarks are OPENFFT and SORT. Many of these benchmarks have been used in past work, e.g., MPI-CHECKER used AMG2013 and OPENFFT, and MUST used the NPB benchmarks for their evaluations respectively.

Table II gives the basic statistics of the benchmarks, including their size in lines of code, their numbers of static MPI call sites, as well as of static non-blocking MPI call sites. Since our approach is based on partial symbolic execution (PSE), we do not have to start the execution from the main function. The last column of Table II gives the entry points used to initiate PSE for each benchmark. Other detailed execution configurations are given by Table VI in Section A-C.

The symbolic execution related configurations used for each benchmark are presented in Appendix A-C Table VI, including the number of processes simulated, interpretation-related parameters (e.g., maximum depth of fork, maximum solver time and the maximum number of loop iterations visited), and the default size assumed for lazy allocation.

To evaluate the effectiveness and performance (details in Section IV-B and IV-C), we compared our PSE approach with static and dynamic verification tools. For the static case, we chose MPI-CHECKER [6] which provides a state-of-the-art lightweight MPI verification mechanism based on the analysis of the abstract syntax tree (AST) level program representation. Limited by its high-level abstraction and lightweight design, MPI-CHECKER cannot achieve the same level of precision as our KLEE based approach that is capable of modeling branch conditions and reasoning about heaps. For lightweight checking, MPI-CHECKER has been integrated into the official release of LLVM CLANG compilation toolchain.

For the dynamic case, we chose MUST [7], which focuses on detecting MPI anomalies related to non-blocking MPI APIs. As a typical dynamic verification tool, MUST instruments the target program and runs the instrumented program to obtain a trace for analysis. It can precisely analyze a concrete execution of the given program, but it is limited by its input dependent nature. For the input of the dynamic test, we used the standard test data provided by benchmark vendors.

B. Effectiveness

Table III shows statistics regarding the effectiveness of PSE compared with MPI-CHECKER and MUST on 6 benchmarks.

For AMG2013, MPI-CHECKER reported 2 REQUEST OVERWRITING and 1 UNMATCHED WAIT anomalies from the same code region that is covered by PSE. Those are all false positives. Figure 4 shows the code extracted from AMG2013 that causes the false positives. The anomaly report of MPI-CHECKER indicates that with the knowledge that `myid` is greater than 0, after assuming the condition at line 7 to be false in the first iteration, MPI-CHECKER makes another assumption that `children_complete` does not equal to zero at line 3 in the second iteration. This is due to the bitwise AND operation inside the branch condition, which is beyond MPI-CHECKER's constraint reasoning capability. PSE and MUST did not report any anomaly for AMG2013.

Both PSE and MUST reported REQUEST OVERWRITING anomalies in 8 locations of ATHENA. MUST identified the anomalies as requests that are not freed, and issued a number of reports for an anomaly in the same location because it was triggered multiple times in different loop iterations. PSE reported at most one anomaly at each program point. PSE also identified BUFFER DATA RACE anomalies in several line ranges, which are not supported by MUST. MPI-CHECKER missed both of those anomalies in ATHENA due to its less precise modeling of heap data.

Figure 1 (in Section I) shows the anomalies introduced in ATHENA at commit 21c4d95 when non-blocking sends were adopted. As discussed in Section I, PSE detects the

TABLE II
CHARACTERISTICS OF MPI BENCHMARKS USED IN OUR EVALUATION

Benchmark	Description	LOC	# of MPI calls	# of nonblocking MPI calls	entry function
AMG2013	A parallel algebraic multigrid solver for linear systems	74,901	402	59	HYPRE_DATAEXCHANGELIST
Athena	A grid-based application for astrophysical magnetohydrodynamics [5]	63,012	86	24	BVALS_MHD
OpenFFT	A high performance FFT library	6,498	196	70	OPENFFT_EXEC_C2C_3D
NPB.IS	Distributed parallel bucket sort	711	20	1	MAIN
NPB.DT	Data traffic simulation	892	16	0	MAIN
Sort	Distributed parallel quick sort	127	12	2	MAIN

TABLE III
EFFECTIVENESS COMPARISON AMONG PARTIAL SYMBOLIC EXECUTION (PSE), MPI-CHECKER(MC) AND MUST. RESULTS ARE ONLY PRESENTED FOR 3 BENCHMARKS, BECAUSE ALL THREE TOOLS REPORTED ZERO ANOMALIES FOR THE REMAINING 3 BENCHMARKS (OPENFFT, IS, DT).

Benchmark	ANOMALY TYPE	POSITION		IS REAL	REPORTED		
		FILE NAME	LINE NUM	ANOMALY	PSE	MC	MUST
AMG2013	REQUEST OVERWRITING	exchange_data.c	453			√	
		exchange_data.c	455			√	
	UNMATCHED WAIT	exchange_data.c	471			√	
Athena	BUFFER DATA RACE	bvals_mhd.c	2368∼2410	√	√		
		bvals_mhd.c	2439∼2481	√	√		
		bvals_mhd.c	2507∼2549	√	√		
		bvals_mhd.c	2575∼2617	√	√		
	REQUEST OVERWRITING	bvals_mhd.c	259	√	√		√
		bvals_mhd.c	263	√	√		√
		bvals_mhd.c	285	√	√		√
		bvals_mhd.c	306	√	√		√
		bvals_mhd.c	362	√	√		√
		bvals_mhd.c	366	√	√		√
		bvals_mhd.c	388	√	√		√
		bvals_mhd.c	409	√	√		√
Sort	BUFFER DATA RACE	sort.c	136∼146	√	√		√

```
1  while (!terminate) {
2    ...
3    if (!children_complete) {
4      ierr = MPI_Testall(tree.num_child,
5                         term_requests,
6                         &children_complete,
7                         term_statuses);
8      if (children_complete & (myid > 0)) {
9        MPI_Isend(NULL,0,MPI_INT,tree.parent_id,
10                 term_tag,comm,&request_parent);
11       ... }
12   } else {
13     ...
14     if (terminate) {
15       if (myid > 0)
16         MPI_Wait(&request_parent,&status_parent);
17       ... } } }
```

Fig. 4. An example that causes MPI-CHECKER to report false positives.

above-mentioned two kinds of anomalies inside it. The non-blocking send operation at line 3 is not checked for completion before a call to `pack_ix2` function, which writes to the send buffer. This is identified as a BUFFER DATA RACE anomaly. After that, another non-blocking send operation is initiated and the request handle of the previous non-blocking operation is reused without confirming the completion of that operation. This is identified as a REQUEST OVERWRITING anomaly.

For SORT, PSE identified BUFFER DATA RACE anomalies caused by a load of a value from the buffer of an ongoing MPI_IRECV operation. The value is used to determine the buffer length of a subsequent MPI_RECV call. MUST caught this anomaly in the form of a mismatch of buffer length between MPI_SEND and MPI_RECV. However, the program halted afterwards without doing any further anomaly detection. We did not test OPENFFT with MUST since MUST can only be applied to a complete program, while OPENFFT is a library.

Based on the above observations, PSE can identify all anomalies reported by the MUST dynamic analyzer, as well as some that are not identified by MUST. Compared with the lightweight static approach, MPI-CHECKER, PSE avoided most of false positives and identified more true positives.

C. Performance

Table IV compares the performance of our approach with that of static/dynamic approaches. (Since the performance of dynamic approaches is highly dependent on the program inputs

used, we just used inputs that could be considered reasonable for debugging purposes.) We collected performance statistics for both execution time and memory usage. Columns 2∼4 show the execution time comparison. Due to the cost of interpreting a program in low-level IR (i.e., both interpretation and symbolic states generation/maintenance are expensive) and shadow memory allocation/management, PSE ran slower than other approaches, especially for ATHENA that is memory intensive and executes pointer operations frequently. In most of the cases, MPI-CHECKER is the cheapest approach, MUST is more efficient in some cases, depending on the input size used for dynamic verification.

Columns 5∼7 show the memory usage comparison. PSE employs shadow memory allocation which is about 10× compared with normal symbolic execution, but it still used less memory than MPI-CHECKER for ATHENA and OPENFFT due to the partial execution. MPI-CHECKER shows lower cost than MUST.

Table V shows the performance of PSE under different configurations. We vary the number of processes simulated and the maximum fork depth while keeping other configurations unchanged as in Table VI. By comparing the data under the same fork depth configuration but different numbers of processes, it can be seen that the increase in the number of processes does not hurt the performance of PSE. This means that our approach is scalable in terms of the number of MPI processes that it can handle. It can also be observed that changing the depth limit of forking largely impacts the time and memory consumption for the three real-world applications, which indicates that there are still a number of states not explored because of the limited fork depth. This is an expected result of PSE, when applied to all three applications, because of a large amount of possible states that could exist due to lazy initialization.

Overall, these results show that debugging based on symbolic execution is a viable approach in terms of both effectiveness and efficiency. The longest execution time was 1960.14 seconds, which is just over 30 minutes. We hope that our work will, in turn, motivate future work on further performance improvements for symbolic execution, e.g., via the use of parallelism within the symbolic executor.

TABLE IV
EXECUTION TIME (SECS) AND MEMORY USAGE (MB) COMPARED WITH MPI-CHECKER(MC) AND MUST.

Benchmark	EXEC TIME(SEC)			MEM USAGE(MB)		
	PSE	MC	MUST	PSE	MC	MUST
AMG2013	28.57	96.70	8.54	220	418	677
Athena	1960.14	27.13	119.19	277	333	700
OpenFFT	75.21	4.76	N/A	74	312	N/A
NPB.IS	206.21	1.00	9.15	1,926	85	654
NPB.DT	9.54	1.60	8.49	1,316	128	714
Sort	0.11	0.39	6.38	27	65	682

V. RELATED WORK

Static and dynamic anomaly detection for MPI program Testing and debugging parallel program are always challenging problems due to the need for both precision and efficiency. There have been both static and dynamic approaches that were developed. The dynamic tools used to profile the program and generate the trace for verification. In this paper, we did experiments on the Marmot Umpire Scalable Tool (MUST [7]) which is a dynamic tool for detecting nonblocking MPI API related anomalies. Based on runtime profiling, MUST employs a graph-based deadlock detection approach for MPI, which also covers future MPI extensions.

ISP [14], [15] tackles the correctness evaluation problem for MPI programs in the form of a "Scalable and Distributed Dynamic Formal Verifier". Based on runtime profiling, ISP replays MPI calls in distinct schedules, to simulate different outcomes of non-deterministic behavior. The range of enforced distinct outcomes derived from non-deterministic behavior is then verified, which leads to a better coverage than replay techniques that do not enforce different schedules. However, both MUST and ISP relies on profiling of MPI commuication APIs, thus it can not identify data race anomalies. Chen et al. introduced SyncChecker [8], which focuses on checking synchronization errors when MPI programs use nonblocking communication. It performs both application level and library level profiling to detect whether the use of the message buffers in nonblocking communication is well synchronized between MPI programs and the underlying MPI library. Pham et al. introduced SimGridMC [16], that uses dynamic partial order reduction and state equality algorithm to to solve state space exploration problem and enhances the efficiency of dynamic MPI verification. Both SyncChecker and SimGridMC instrument low level buffer synchronous operations, they can identify the racy cases for buffer operations, but not for high level anomalies (i.e. type mismatch and unmatched wait).

Static verification approaches typically result in more false positives due to their over-approximations of the underlying abstractions. MPI-Checker [6] is a lightweight static analysis tool that is implemented in LLVM/CLANG. It checks the correct usage of MPI APIs via traversing the C/C++ program presented as abstract syntax tree (AST). Due to the lack of precise control flow information and inter-procedural information, the ratio of false positives is high. Strout et al. built the interprocedural control-flow graphs (ICFGs) [17] for MPI programs, and modeled communication APIs as additional ICFG edges. This technique can help the interprocedural dataflow analysis for MPI programs.

Symbolic execution for anomaly detection Symbolic execution can simulate program behavior and generate test cases in arbitrary contexts, and deliver more precision compared with other static analysis approaches. It also brings more flexibility compared with dynamic approaches that need to be input dependent and reduces overhead for running the whole program. KLEE [3] is a popular state-of-the-art symbolic execution tool built on top of LLVM [4], and it can simulate

TABLE V
EXECUTION TIME (SECS) AND MEMORY USAGE (MB) OF PSE UNDER DIFFERENT CONFIGURATIONS.

Benchmark	AMG2013				Athena				OpenFFT			
# of procs, max fork depth	8, 11	16, 11	8, 12	16, 12	2, 14	4, 14	2, 15	4, 15	2, 11	4, 11	2, 12	4, 12
EXEC TIME(SEC)	11.73	11.74	28.57	27.65	1140.13	1117.67	1960.14	1909.8	42.55	42.3	75.21	75.85
MEM USAGE(MB)	215	215	220	220	220	221	277	278	68	68	74	75
Benchmark	NPB.IS				NPB.DT				Sort			
# of procs, max fork depth	8, 11	16, 11	8, 12	16, 12	11, 11	22, 11	11, 12	22, 12	2, 11	4, 11	2, 12	4, 12
EXEC TIME(SEC)	166.86	148.32	206.21	161.24	9.36	9.52	9.54	9.43	0.11	0.1	0.11	0.1
MEM USAGE(MB)	1940	1490	1926	1539	1316	1316	1316	1316	27	26	27	26

a wide range of program behaviors from the system level to high-level abstractions. Ramos and Engler introduced an under-constrained symbolic execution mechanism [18] with KLEE, that improves scalability by directly checking individual functions, rather than whole programs.

There have been multiple past works that use symbolic execution to verify MPI programs. Fu et al. introduced a KLEE based tool [19] that records the MPI APIs' status and applies a lazy matching mechanism that identifies the communication pairs until the execution touches the block operations. In [20], Bergan et al. introduced a technique to evaluate the symbolic synchronous technique during symbolic executing multithreaded applications. They proposed a context-specific dataflow analysis that can precisely approximate the initial context with low overhead to avoid some infeasible-path explosion. In [21], Li et al. discussed automatically creating parametric flows for symbolic execution. Their technique enhances the verification of CUDA program via checking for data races only across a pair of threads per parametric flow. Siegel et al. introduced concurrency intermediate verification language (CIVL) [22] approach, that translates parallel programs like MPI or libraries into CIVL-C program, which is a dialect of C with concurrent extension, and uses symbolic execution to model check the programs.

Lazy initialization for symbolic execution

In this paper, we employed lazy initialization for both scalar variable and heap fields to enable the partial symbolic execution. The lazy initialization technique was introduced in [12], which can generate the test for those heap address with unclear initialization states. Their approach is to nondeterministically choose if a heap address should be initialized/uninitialized, or alias to other address. There are several works that improve lazy initialization, like delaying aliasing choices [23] and checking initializations against invariants [24]. Hillery et al. introduced generalized symbolic execution (GSE) [25], which gives a method for initializing input references in a symbolic input heap using guarded value sets that exactly preserves GSE semantics. Their initialization technique can be used to ensure that guarded value set based symbolic execution engines operate in a provably correct manner with regards to symbolic references as well as provide the ability to generate concrete heaps that serve as test inputs to the program.

In our work, we employ partial symbolic execution that works on a selected program region, starting from the user-specified function. The initialization of heap fields is based on lazy initialization that is extended to support for weakly typed programming languages and takes pointer alias information to assist the selection of shadow memory objects. By leveraging the advantages of symbolic execution, our approach has the capacity to explore multiple execution paths in a program, thereby avoiding the false negative limitations of dynamic analysis approaches in which each execution with a concrete set of inputs may only explore one execution path in a program. Our approach can also avoid the false alarms that are inherent in past approaches based on static program analysis. Based on these advantages, our approach can detect MPI communication API-related anomalies, especially for racy anomalies that impact various program configurations.

VI. CONCLUSIONS AND FUTURE WORK

This paper introduced a new static analysis framework for MPI-based parallel programs using partial symbolic execution, as well as several techniques to detect anomalies in MPI usage built on top of this framework. We built our prototype of this partial symbolic execution (PSE) based anomaly detection system on top of the KLEE [3] symbolic execution engine. Our evaluation with two real-world application and four MPI benchmarks show that PSE is effective in detecting the MPI communication API-related anomalies (especially for nonblocking communication operations) with high precision and acceptable overhead compared with state-of-art static and dynamic verification tools (i.e., MPI-CHECKER and MUST).

In this paper, we evaluated our anomaly detection approach for MPI applications that use a single MPI_COMM_WORLD communicator. As future work, we plan to extend our symbolic executor to support multiple communicators. There may also be opportunities to relax some of the assumptions made in our approach, e.g., no misaligned pointer accesses, and no negative offsets in pointer arithmetic. Finally, we are exploring approaches to parallelize our partial symbolic execution to further improve its compile-time performance.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive suggestions and comments, which helped improve the presentation of the paper. We are especially thankful for the observation by a reviewer that it suffices to allocate 1 bit in shadow memory for each pointer, rather than 1 bit per byte.

We would also like to thank Ganesh Gopalakrishnan and Alex Orso for their valuable feedback on earlier drafts of this paper, and for pointers to related work.

REFERENCES

[1] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Formal Analysis of MPI-based Parallel Programs," *Commun. ACM*, vol. 54, no. 12, pp. 82–91, Dec. 2011.

[2] *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993): IEEE Standard Classification for Software Anomalies*. IEEE, 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5399061

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[4] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[5] "Athena," available at: https://github.com/PrincetonUniversity/Athena-Cversion/.

[6] A. Droste, M. Kuhn, and T. Ludwig, "MPI-checker: Static Analysis for MPI," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:10.

[7] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Muller, "MPI runtime error detection with MUST: Advances in deadlock detection," in *2012 SC - International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*, vol. 00, 11 2012, pp. 1–10.

[8] Z. Chen, X. Li, J.-Y. Chen, H. Zhong, and F. Qin, "SyncChecker: Detecting Synchronization Errors Between MPI Applications and Libraries," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 342–353.

[9] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, "Report of the HPC Correctness Summit Jan 25 and 26, 2017, Washington, DC," 2017, available at: https://science.energy.gov/ /media/ascr/pdf/program-documents/docs/2017/HPC_Correctness_Report.pdf.

[10] "Clang Compiler," available at: http://clang.llvm.org/.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[12] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 553–568.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991, available at: https://www.nas.nasa.gov/publications/npb.html.

[14] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur, "Formal Verification of Practical MPI Programs," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 261–270.

[15] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[16] A. Pham, T. Jéron, and M. Quinson, "Verifying MPI Applications with SimGridMC," in *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, ser. Correctness'17. New York, NY, USA: ACM, 2017, pp. 28–33. [Online]. Available: http://doi.acm.org/10.1145/3145344.3145345

[17] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for MPI programs," in *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 2006, pp. 175–184.

[18] D. A. Ramos and D. Engler, "Under-constrained Symbolic Execution: Correctness Checking for Real Code," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 49–64.

[19] X. Fu, Z. Chen, H. Yu, C. Huang, W. Dong, and J. Wang, "Symbolic Execution of MPI Programs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 809–810.

[20] T. Bergan, D. Grossman, and L. Ceze, "Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 491–506.

[21] P. Li, G. Li, and G. Gopalakrishnan, "Parametric Flows: Automated Behavior Equivalencing for Symbolic Analysis of Races in CUDA Programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 29:1–29:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389036

[22] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: The Concurrency Intermediate Verification Language," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 61:1–61:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807635

[23] X. Deng, Robby, and J. Hatcliff, "Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs," in *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 273–282.

[24] P. Braione, G. Denaro, and M. Pezzè, "Symbolic Execution of Programs with Heap Inputs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 602–613.

[25] B. Hillery, E. Mercer, N. Rungta, and S. Person, "Exact heap summaries for symbolic execution." in *Verification, model checking, and abstract interpretation. 17th international conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings*. Berlin: Springer, 2016, pp. 206–225.

[26] "AMG2013," available at: https://codesign.llnl.gov/amg2013.php.

[27] "OpenFFT," available at: http://www.openmx-square.org/openfft/.

APPENDIX A
ARTIFACT DESCRIPTION: DETECTING MPI USAGE ANOMALIES VIA PARTIAL PROGRAM SYMBOLIC EXECUTION

A. Implementation and Environment

The prototype of our MPI usage anomaly detector is implemented as an extension to KLEE (commit d19500e) with LLVM 3.6.2. Its source code can be downloaded from `https://github.com/fkye/PSE-MPI`.

We built and tested it using GCC 5.5.0 on a Linux workstation with an Intel Core i7-3770K processor running at 3.50GHz and 16GB of RAM. The constraint solver we used is Z3 version 4.6.0.

B. Benchmarks

We used the following benchmarks to evaluate our tool:

- AMG2013 can be obtained from [26]. For the concrete execution required by the MUST dynamic analysis tool, we used the test input file provided inside the source tree as the input.
- Athena is available at [5]. The data race bugs were introduced in commit 21c4d95 and fixed in commit 4ca1615. For evaluation, we used the code in commit a25e5fa, which happens to be one commit before the fixed version. The input file for the concrete execution of MUST is tst/3D-mhd/athinput.field_loop.
- OpenFFT can be downloaded from [27]. This benchmark is not executed concretely in our evaluation because it is a library instead of a complete program, and thus no input is needed.
- NPB.IS and NPB.DT come from the NPB 3.3.1 benchmark suite, which is available at [13]. We used the "W" problem size for both benchmarks in our evaluation.
- Sort is a benchmark program used by SyncChecker [8] for evaluation. No input is required for this benchmark.

C. Execution

The benchmarks are compiled into single LLVM bitcode files. Our tool takes the bitcode files as well as their entry functions and execution configurations as input. The entry functions are listed in Table II. Table VI gives the execution configurations for each benchmark including the number of processes simulated, interpretation-related parameters (e.g., maximum depth of fork, maximum solver time and the maximum number of loop iterations visited), and the default size assumed for lazy allocation.

APPENDIX B
ADDITIONAL TECHNICAL DETAILS

A. Additional Algorithms for Partial Symbolic Execution

Algorithm 3 presents the lazy initialization of pointer states with pointer alias information (see **LazyInit** function). The **NewObject** function presents the logic for handling allocation of new objects.

Algorithm 4 shows how to identify MPI derived types. It employs a backward program slicing pass from the derived

Algorithm 3: Lazy Initialization

```
1  function NewObject ()
   Inputs : p: pointer reference, T: pointer's type, MemMap: the
            pointer to allocated memory map
   Output: allocated memory object
   /*Allocate a new memory block for the
     pointer-type value.                                    */
2  p := Alloc (T);
   /*Mark new bytes as unbounded symbolic
     values.                                                */
3  p := MarkBytesAsUnbounded (p);
   /*Mark the pointer-typed value as bounded
     and constrain it to be equal to the
     starting address of the newly al located
     memory block.                                          */
4  p := MarkPointerAsBounded (p)
   /*Register the allocated memory object to
     type T.                                                */
5  MemMap[p] ∪= MemBlock (p);
6  return p
7
8  function LazyInit ()
   Inputs : p: pointer reference, T: pointer's type, MemMap: the
            map between pointer references and allocated memory
            obje cts, F: current function, AA: pointer alias
            information
   Output: the memory object for p
9  if p is uninitialized then
       /*Get p's must alias pointer reference.
         */
10     q := MustAlias (p, AA); if q ≠ ∅ then
11         if q is initialized then
               /*Initialize p with q's memory
                 object.                                    */
12             MemMap[p] := MemMap[q];
13         else
               /*Allocate a new memory object.              */
14             MemMap[p] := NewObject (T, MemMap);
15             MemMap[q] := MemMap[p];
16     else
           /*Fork the current state and allocate
             a new memory object for the new
             state.                                         */
17         newState := ForkCurrentState ();
18         newObj := NewObject (T, newState.MemMap);
19         newState.MemMap[p] := newObj;
           /*Get may alias pointer references'
             allocated memory objects.                      */
20         mayAlias := MayAliasObjects (p, AA);
           /*Nondeterministically select one of
             the two options.                               */
21         fld := NonDetSelect (null, mayAlias);
           /*Check if function F's pre-condition
             is violated with p's shadow memory
             allocation.                                    */
22         if ViolatePreCond (fld, F) then
23             Cancel ();
24         MemMap[p] := fld;
```

TABLE VI
CONFIGURATIONS FOR SYMBOLIC EXECUTION.

Configuration	AMG2013	Athena	OpenFFT
Number of processes	8	2	2
Max fork depth	12	15	12
Max solver time (s)	1	1	1
Max loop iterators	1	1	1
Default lazy array size	1	2	1
Configuration	**NPB-IS**	**NPB-DT**	**Sort**
Number of processes	8	11	2
Max fork depth	12	12	12
Max solver time (s)	1	1	1
Max loop iterators	1	8	1
Default lazy array size	1	1	10

type creation operation to identify the MPI primitive types that compose the derived type. For each derived type, **DerivedTypeAnalysis** gathers its primitive type component set and registers it in an internal table that can be queried via **GetCompTys**.

Algorithm 4: MPI Derived Type Analysis

```
1  function DerivedTypeAnalysis ()
   Inputs : P: MPI program, T: derived type handle, AA: alias
            analysis
   Output: CompTys: a set of MPI primitive types
2  CompTys := ∅;
   /*Backward slicing from the derived type
     creation call of T.                              */
3  OpSet := BackwardSlicing (T, P, AA);
   /*Get the type array assignment operations
     used for composing derived type.                 */
4  TyArrOps := GetTypeArrayOps (OpSet);
5  foreach TyAssign in TyArrOps do
6      ty := GetRHS (TyAssign);
       /*Check if RHS is a MPI primitive type.
         */
7      if IsMPIPrimTy (ty) then
           /*Get the corresponding C primitive
             type.                                    */
8          CompTys.Insert(GetPrimTy (ty));
9      else
10         CompTys := ANY_TYPE;
11         break;
12 if CompTys = ∅ then
       /*If the slicing can not identify the
         type array operations, we make the
         derived type as any type.                    */
13     CompTys := ANY_TYPE;
   /*Register the derived type with its
     corresponding C struct type.                     */
14 RegisterDerivedTy (T, CompTys);
15 return CompTys;
```

Algorithm 5 is used to compare a buffer type with a MPI datatype handle. If the input MPI datatype is a derived type, **Match** uses **GetCompTys** to obtain its primitive type component set and compare it with the buffer type. If one of them is a wildcard type, they are considered to be matched. If the buffer type is a composite type and its components are different from that of the derived type, then they can not be matched to each other. If the derived type is composed of a single primitive type, it is treated like that primitive type.

Algorithm 5: MPI Buffer Type Matching

```
1  function Match ()
   Inputs : T_buf: C primitive type, T_mpi: MPI datatype handle
   Output: Boolean result
2  if T_buf = void or T_mpi = MPI_Byte then
3      return true;
4  if IsMPIPrimTy (T_mpi) then
       /*Get the C primitive type.                    */
5      T_c := GetPrimTy (T_mpi); if T_buf = T_c then
6          return true;
7      else
8          return false;
9  else
       /*IsMPIDerivedTy (T_mpi) = true                */
       /*Get C primitive type components.             */
10     S_c := GetCompTys (T_mpi);
11     if T_c is ANY_TYPE then
12         return true;
13     if T_buf is a composite type then
14         S_buf := T_buf's components;
15         return S_buf = S_c;
16     if T_c is composed of one C primitive type T_base then
17         return T_buf = T_base;
18     return false;
```

B. Check Coverage

The details of API coverage for our anomaly checkers is summarized in Figure 5. The BUFFER TYPE MISMATCH checker works for all MPI communication calls. The other checkers do not support persistent communication calls. The BUFFER DATA RACE checker supports all nonblocking calls using the MPI_COMM_WORLD communicator.

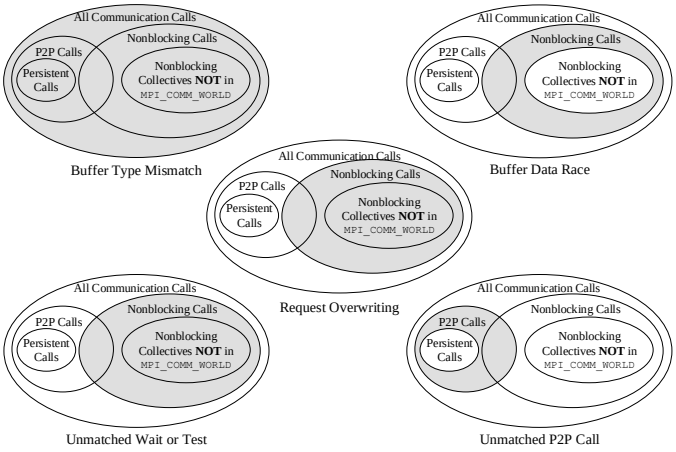

Fig. 5. The coverage of MPI APIs for each anomaly detector related to nonblocking APIs.