Algorithm Diversity for Resilient Systems

Scott D. Stoller, Yanhong A. Liu

Algorithm Diversity for Resilient Systems

Scott D. Stoller and Yanhong A. Liu

Department of Computer Science, Stony Brook University, USA
{stoller,liu}@cs.stonybrook.edu

Abstract. Diversity can significantly increase the resilience of systems, by reducing the prevalence of shared vulnerabilities and making vulnerabilities harder to exploit. Work on software diversity for security typically creates variants of a program using low-level code transformations. This paper is the first to study *algorithm diversity* for resilience. We first describe how a method based on high-level invariants and systematic incrementalization can be used to create algorithm variants. Executing multiple variants in parallel and comparing their outputs provides greater resilience than executing one variant. To prevent different parallel schedules from causing variants' behaviors to diverge, we present a *synchronized execution* algorithm for DistAlgo, an extension of Python for high-level, precise, executable specifications of distributed algorithms. We propose static and dynamic metrics for measuring diversity. An experimental evaluation of algorithm diversity combined with implementation-level diversity for several sequential algorithms and distributed algorithms shows the benefits of algorithm diversity.

1 Introduction

Diversity can significantly increase the resilience of systems, by reducing the prevalence of shared vulnerabilities and making vulnerabilities harder to exploit. The idea of intentionally introducing software diversity as a defense mechanism has been around for decades, e.g., [5,6]. It is closely related to the well-known *moving target defense* (MTD) strategy: running different variants of a program at different times is MTD. Software diversity is an effective defense against attacks whose success depends on details of the victim software. Without knowing those details for the specific instance (variant) of the software being attacked, attackers can still attempt such attacks (e.g., by making random guesses at those details), but the probability of success is greatly reduced [16].

There is a large corpus of research on techniques for automatically introducing software diversity that increase resilience to various classes of attacks [16]. For example, Address Space Layout Randomization (ASLR), which randomizes the starting addresses of segments in a process's address space, is a classic form of software diversity that increases resilience to some types of memory corruption attacks.

The most common way to use software diversity to increase resilience is to run a randomly selected variant each time the program is executed. With this approach, the use of diversity alters, with high probability, the effect of an attack, so the attack does not have the intended effect (e.g., gaining root privilege and installing a rootkit) [16]. The attack might still have a less malicious and less predictable but nevertheless undesirable effect (e.g., crash or incorrect output).

Another way is to run multiple variants of the application in parallel and compare their outputs. We call this *diversified replication*. Any difference in the outputs of the variants indicates misbehavior of one or more variants due to an attack; this triggers defensive action. This approach provides greater resilience, at the cost of more computational resources. It also provides greater resilience than traditional replication, in which replicas are identical and exhibit the same (incorrect) behavior when their vulnerabilities are exploited. Note that diversity may lead to different behavior (and therefore attack detection) in two ways: (1) it might cause a difference in the direct effect of the attack (e.g., which data structure is overwritten) or, (2) even if the direct effect of the attack is the same (e.g., the same data structure is overwritten), it might cause differences in subsequent behavior, due to differences in the algorithms or implementations used by the variants (e.g., one variant reads the affected data structure earlier in its computation and hence before the attack, and another reads the affected data structure later in its computation and hence after the attack).

This paper focuses on *algorithm diversity* for software resilience, in which different variants run different algorithms, i.e., perform different computations at a high level. In contrast, all of the work surveyed in [16] creates *implementation-level diversity*, changing details of the implementation without changing the algorithm. Algorithm diversity can introduce new and larger differences between variants than implementation-level diversity and hence can provide greater resilience, especially when used together with implementation-level diversity.

Algorithm variants may be obtained in a variety of ways, besides writing them manually. For standard problems (e.g., dictionary ADT), they can be obtained from algorithm libraries. A more general automated approach is to generate them by starting with a high-level algorithm (or specification) and applying different optimizations (algorithm improvements, automated using program analysis and transformation). In particular, we have used a method based on systematic incrementalization [27,22,19,18], which transforms programs to maintain high-level invariants incrementally, and related optimizations to generate multiple variants of many sequential algorithms and distributed algorithms [26,25,24].

Algorithm diversity and implementation-level diversity introduce different kinds of variation and together offer more diversity than either alone. We introduce *diversity metrics* that quantify the difference between—or equivalently, the similarity of—variants. We consider a static metric, *code diversity*, based on the instruction sequences in the compiled program, and two dynamic (behavioral) metrics: *trace diversity*, based on the sequence of instructions executed, and *input access diversity*, based on the sequence of accesses to input data. The latter dynamic metric is motivated by the fact that invalid inputs are the primary attack vector for external attackers. A direction for future work is to augment these broad diversity metrics with more specialized metrics that quantify resilience to specific classes of attacks.

Algorithm diversity can be applied to programs in any language. In this paper, we focus on Python and DistAlgo [25,24], an extension of Python for high-level, precise, executable specifications of distributed algorithms. In contrast, existing work on automated software diversity primarily targets C programs or (disassembled) machine code.

Python is interpreted—more precisely, CPython, the predominant implementation of Python, compiles Python to bytecode and then runs the bytecode in an interpreter. Algorithm diversity applied to Python programs can be used together with implementation-level diversity applied to Python programs and the runtime system. This achieves greater total diversity and increases resilience to vulnerabilities in the runtime system, because vulnerabilities manifest only with specific inputs, and the runtime system's inputs include Python programs as well as network messages, UI events, etc. Diversity at the high-level language level provides additional protection from data-only attacks [4,30], against which many runtime-system-level defenses are less effective. Algorithm diversity applied to Python programs can also provide resilience to functional faults in the runtime system, if the runtime system does not correctly implement the semantics of some built-in constructs or library functions in some (corner) cases.

Diversified replication requires *synchronized execution* (often called *N-version execution* [1]) of the variants; otherwise, their executions might diverge due to scheduling differences. Synchronized execution of distributed programs generally requires synchronization of message delivery order. DistAlgo's asynchronous message handling requires additional synchronization, to ensure that all variants handle corresponding messages at corresponding points in their executions. We developed a synchronized execution framework for DistAlgo that ensures this. Our framework can also suppport variants whose behaviors differ in prescribed ways.

Measuring dynamic diversity for Python and DistAlgo programs required development of new runtime monitoring tools, which are also more broadly useful. We designed and implemented a tool that intercepts accesses to fields of selected objects; we use it to log accesses to objects read as input, including objects received in messages. Handling built-in types such as integers and strings is tricky, because they are sometimes accessed directly by C code in the Python interpreter, but essential, because they are commonly used in program inputs.

We also designed and implemented a tracing tool that reconstructs the exact sequence of bytecode instructions executed by a Python program. It uses the standard Python tracing module to record the sequence of source lines executed, and then analyzes the compiled program to determine the sequence of bytecode instructions corresponding to each source line. Supporting DistAlgo requires some extra work, due to details of DistAlgo's implementation by translation to Python.

In summary, the contributions of this paper include:

- The first study of semi-automated algorithm diversity for software resilience, using a method based on systematic incrementalization to generate algorithm variants.
- A synchronized execution framework for DistAlgo and for high-level executable specifications of distributed algorithms.
- Static and dynamic metrics for measuring diversity.
- A runtime monitoring tool for Python and DistAlgo that logs accesses to fields of selected objects, including instances of built-in types.
- A tracing tool for Python and DistAlgo that reconstructs the exact sequence of executed bytecode instructions.
- Experimental evaluation of algorithm diversity combined with implementation-level diversity for several sequential algorithms and distributed algorithms, demon-

strating that algorithm diversity can achieve more diversity than implementation-level diversity, and the two together can achieve even more.

2 Background on DistAlgo

Liu et al. [25,24] propose DistAlgo, a language for high-level, precise, executable specifications of distributed algorithms, and study its use for specification, implementation, optimization, and simplification of such algorithms. For expressing distributed algorithms at a high level, DistAlgo supports four main concepts by building on an object-oriented programming language, Python: (1) distributed processes that send messages, (2) control flow for handling received messages, (3) high-level queries for synchronization conditions, and (4) configuration for setting up and running. DistAlgo is specified precisely by a formal operational semantics [24].

Processes That Send Messages. A process type P is defined by a class definition for P that inherits from DistAlgo's built-in `process` class. The definition of P may contain, in addition to the usual definitions that may appear in Python classes, definition of a `setup` method for taking in and setting up the values used by the process, definition of a `run` method containing the main control flow of the process, and definitions of `receive` handlers for handling messages, as described below.

To create instances of P, DistAlgo provides a `new` P construct; it can optionally be preceded by the number of processes to create (the default is 1) and followed by "`at` h" where h identifies the host where the process(es) should be created (the default is the local host). After a new process has been created, and its `setup` method called to initialize it, invoking its `start` method causes execution of its `run` method.

Processes send messages using the statement `send` m `to` ps, where ps is a process or set of processes.

Control Flow for Handling Received Messages. Received messages can be handled asynchronously, using `receive` definitions, and synchronously, using `await` statements. A `receive` definition has the form `receive` m `from` p`:` *stmt*. It handles un-handled messages that match m `from` p, where m and p are patterns. If matching succeeds, unbound variables in m (and p) are bound to the corresponding component of the message (and the message's sender, respectively), and then *stmt* is executed.

To synchronize message handling with local computation, `receive` handlers are executed only at *yield points*. The program point before or after any statement can be declared as a yield point. In addition, there is an implicit yield point before each `await` statement, for handling messages while waiting. By default, any number of pending messages can be handled at a yield point.

An `await` statement has the form

`await` $cond_1$`:` $stmt_1$ `or ... or` $cond_k$`:` $stmt_k$ `timeout` t`:` *stmt*

It waits until one of $cond_1$, ..., $cond_k$ is true or time t has elapsed, and then nondeterministically selects one of $stmt_1$, ..., $stmt_k$, *stmt* whose condition is true and executes the selected statement. Each branch is optional.

High-Level Queries for Synchronization Conditions. DistAlgo provides constructs to express synchronization conditions in `await` statements as high-level queries over message histories (or other sets or sequences). A query can be an existential or universal quantification, a comprehension, or an aggregation. An existential quantification has the form `some` v_1 `in` s_1`, ...,` v_k `in` s_k `|` *cond*. It returns true iff *cond* holds for some combination of values of variables that satisfies all v_i `in` s_i clauses. Universal quantification is similar, with keyword `each` instead of `some`.

A comprehension has the form `{`e`:` v_1 `in` s_1`, ...,` v_k `in` s_k`,` *cond*`}`. It returns the set of values of e for all combinations of values of variables that satisfy all v_i `in` s_i clauses and condition *cond*.

DistAlgo automatically maintains histories of messages sent and received by each process in variables `sent` and `received`; they are automatically eliminated if unused.

Configuration. Configuration for requirements such as use of logical clocks and use of reliable and FIFO channels can be specified using DistAlgo's `configure` statement. For example, `configure clock = Lamport` specifies that Lamport's logical clocks are used; it configures sending and receiving of a message to update the clock value, and defines a function `logical_time()` that returns the clock value.

3 Creating Variants using Incrementalization

Algorithm variants differ from each other due to different high-level invariants they maintain and different ways of maintaining them. We describe the ideas of transforming expensive queries into high-level invariants and using systematic incrementalization to generate efficient algorithms that maintain the query results incrementally. Each resulting combination of ways of maintaining the invariants forms an algorithm variant.

Example. We use as an example Lamport's algorithm for distributed mutual exclusion, described in his seminal paper that proposed logical clocks [14]. The problem is for multiple processes to access a shared resource mutually exclusively, in what is called a critical section, i.e., there can be at most one process in a critical section at a time.

Each process can be expressed in DistAlgo as in Figure 1 [24]. The process is set up with sets `s` and `q` (lines 3-4). To run a task mutually exclusively, the process sends a request and adds it to `q` (lines 6-8), waits for (i) own request `(t,self)` to be before each other request `(t2,p2)` in `q` and (ii) having received an ack with a time `t2` later than `t` from each process `p2` in `s` (lines 9-10) before doing the task in critical section (line 11), and then removes the request from `q` and sends a release (lines 12-13). When receiving a request or release, it sends back an ack and adds to or removes from `q` (lines 14-19).

The two conditions in `await` are key to the algorithm to ensure mutual exclusion, while the rest does basic sending and receiving of messages and bookkeeping of `q`.

Incrementalization. Incrementalization transforms queries and updates to maintain high-level invariants, including invariants for intermediate and auxiliary values, incrementally [27,22,26,24]. It can yield diverse algorithms.

```
 1 class P extends process:
 2   def setup(s):
 3     self.s := s       # set of all other processes
 4     self.q := {}      # set of pending requests
 5   def mutex(task):
 6     self.t := logical_time()                            # request
 7     send ('request', t, self) to s                      #
 8     q.add(('request', t, self))                         #
 9     await each ('request', t2, p2) in q
                | (t2,p2) != (t,self) implies (t,self) < (t2,p2)
10           and each p2 in s | some ('ack', t2, =p2) in received | t2 > t
11     task()
12     q.del(('request', t, self))                         # release
13     send ('release', logical_time(), self) to s         #
14   receive ('request', t2, p2):                          # receive request
15     q.add(('request', t2, p2))                          #
16     send ('ack', logical_time(), self) to p2            #
17   receive ('release', _, p2):                           # receive release
18     for ('request', t2, =p2) in q:                      #
19       q.del(('request', t2, p2))                        #
```

Fig. 1. Lamport's algorithm (lines 6-19) plus setup in DistAlgo.

For the example in Figure 1, the two conditions in `await` are queries, consisting of three quantifications including two that are nested; and assignments and bookkeeping for `s` and `q` and implicitly adding to `received` at `receive` handlers are updates.

The most direct algorithm can compute queries using iterations, in `for`-loops, whereas an incremental algorithm can maintain the results of queries at updates and look up the results as needed. An incrementalized algorithm maintains high-level invariants not only for the query results but also for intermediate and auxiliary values needed. Alternative invariants can often be used, yielding even greater diversity.

For example, the condition on line 10 in Figure 1 can be transformed into

```
count {p2: p2 in s, ('ack', t2, p2) in received, t2 > t} = count s
```

and then—with variables `responded`, `number`, and `total` holding the set value and two `count` values, respectively, forming three invariants—transformed into:

```
number = total
```

Variable `total` is computed at set up of `s`, and `responded` and `number` at request, and the following `receive` handler is added:

```
receive ('ack', t2, p2):          # new message handler
  if t2 > t:                      # comparison in the condition
    if p2 in s:                   # membership in the condition
      if p2 not in responded:     # test before adding
        responded.add(p2)         # add to responded
        number +:= 1              # increment number
```

The resulting algorithm differs significantly from direct iteration for the nested quantifications. The condition on line 10 could also be transformed into two nested `count` queries, and the condition on line 9 can be transformed into a `count` query also, or an aggregate query using `min`, yielding different algorithms for incremental maintenance. Details of these transformations are in [24].

In general, incrementalization can also transform nested loops that compute aggregate values such as sum and min [23,7,26]. For recursive functions as queries, the resulting incremental algorithm can still use recursion, forming an optimized recursive algorithm, or use iteration, forming an optimized iterative algorithm [20,26]. Additionally, more refined data structures can be used to implement sets more efficiently [3,10,26], such as using one bit for each process in the set `responded` above. Incrementalization also enables new additional optimizations that are made possible as the results of systematic transformations [26].

4 Synchronized Execution for DistAlgo

A *diversified process* is a process with variants. A system may contain a mixture of diversified and un-diversified processes. A *gateway* process is created for each diversified process. It represents the variants of a diversified process to the rest of the system, making them appear as a single process. The gateway intercepts and forwards all inbound and outbound messages of all variants of the diversified process. We focus on synchronization of DistAlgo constructs; other I/O events, such as file accesses, can be synchronized using standard techniques.

Our synchronized execution framework consists of two parts: (1) an automated program transformation that (1a) ensures all messages are routed via the gateway, and (1b) inserts synchronization with the gateway at yield points, to ensure that all variants have yielded the same number of times before handling their copies of a given inbound message, despite differences in message latency and process execution speed; and (2) an algorithm run by the gateway that determines when to forward messages and when to report divergence (i.e., differences in behavior). When divergence is reported, the system may initiate application-specific defensive action.

We first present the core version of this approach, which assumes all variants of a process have the same communication pattern, i.e., send the same messages to the same destinations in the same order; we discuss later how to relax this assumption.

Handling Outbound Messages. To route outbound messages via the gateway, the transformation replaces all calls to DistAlgo's `send` method with calls to `send_sync`, and it inserts a definition of that method in every process class. `send_sync` sends the original message and its original destination to the gateway. Processes often send their own process id in messages. Since each variant of a diversified process has a unique process id, such messages will differ. To accommodate this as normal behavior, not divergence, `send_sync` replaces all occurrences of the variant's process id in the message with the gateway's process id. This also reflects the principle that the gateway represents the variants to the rest of the system. Pragmatically, it ensures that, if the recipient sends a reply to the process id contained in the message, the reply goes to the gateway, as desired.

The gateway stores un-forwarded outbound messages received from each variant in a separate FIFO queue. When all of the queues are non-empty, it compares the messages (including their destinations) at the heads of the queues. If they are identical, the gateway dequeues the message from all queues and forwards one copy to the destination, otherwise it reports divergence. To ensure liveness if some divergent variant fails to send a message, once one queue becomes non-empty, the gateway waits a limited amount of time for all queues to become non-empty; if this time limit is exceeded, the gateway reports divergence.

Synchronization at Yield Points and `await` *Statements.* The transformation inserts a call to `yield_sync(`*block*`,` *timeout*`)` at every yield point, and it inserts a definition of that method in every process class. The first argument *block* is a boolean that indicates whether the yield point is associated with an `await` statement. The second argument *timeout*, meaningful when the first argument is `True`, is a timeout duration if the `timeout` clause is present in that `await` statement and is `None` otherwise. The transformation also extends the `setup` method to initialize a variable `num_yields` to zero. `yield_sync` increments `num_yields`, sends a yield message containing *block*, *timeout*, and `num_yields` to the gateway and waits for a yield-reply message from the gateway before returning.

The transformation for an `await` statement with timeout ensures the total wait time is preserved, even though the waiting period may be split by interactions with the gateway. It transforms `await` c_1`:` s_1 `or` ... `or` c_k `:` s_k `timeout` t`:` s into

```
1   start_time = time.time()
2   while not (c_1 or ... or c_k):
3      elapsed = time.time() - start_time
4      remaining = t - elapsed
5      if remaining ≤ 0:
6         break
7      yield_sync(True, remaining)
8   if c_1: s_1
9   elif c_2: s_2
10  ...
11  elif c_k: s_k
12  else s
```

If the `await` statement has no timeout, then lines 1, 3–6, and 12 are omitted, and the second argument of `yield_sync` is `None`.

Handling Inbound Messages. When the gateway receives an inbound message m, it stores m in a queue of un-forwarded inbound messages, waits until it has received yield messages with the same `num_yields` from all variants, forwards to all variants and dequeues all un-forwarded inbound messages, and then sends a yield-reply message to all variants. The gateway communicates with the variants over FIFO channels, so all variants handle the forwarded messages before proceeding from the current yield point. In the copy of m to be forwarded to variant p, the gateway replaces all occurrences of its own process id with p's process id.

If the gateway has received a yield message from all variants, and has no inbound message to forward to them, its behavior depends on the values of *block* and *timeout* in the yield messages (if the values of *block* differ, or the values of *timeout* differ by more than a small amount, divergence is reported). If *block*=`False`, the gateway sends a yield-reply message to all variants, allowing them to proceed. If *block*=`True` and *timeout*=`None`, the gateway waits until it has received and forwarded an inbound message before sending a yield-reply message, since the conditions in the `await` statements will remain false until the variants' states are updated by handling of an inbound message. If *block*=`True` and *timeout* is a number, the gateway behaves as in the previous sentence, except it will also send a yield-reply message after time *timeout* has elapsed.

Process Creation. The program transformation reads a configuration file that specifies which process types are diversified and the types of their variants. For each diversified process type P, a gateway type `Gateway`P is generated (basically, this is done by instantiating template code with the type P and the types of its variants), and process creation statements with type P are transformed to create instances of `Gateway`P instead. The `setup` method of `Gateway`P creates an instance of each of the specified variant types, and passes the gateway's process id to the variants as an additional argument to their `setup` methods, which are transformed to accept this additional argument.

Relaxed Synchronization. The above approach effectively introduces a barrier synchronization for a diversified process's variants at each synchronization point. This ensures the most timely detection of divergence. An alternative approach, used in some other synchronized execution frameworks [13,33], is to allow one variant (the "leader") to get ahead, try to make the actions of the other processes (the "followers") consistent with the leader's actions (e.g., by delivering the same number of messages at the corresponding yield event), and reporting divergence when this is not possible. This may provide speedup but allows a divergent leader to perform divergent actions before the leader's divergence is detected; when this is unacceptable, such actions should not be allowed to have externally visible effects until the followers catch up and agree on the actions.

Allowing Differences in Message Pattern. It may be desirable to relax the requirement that corresponding messages sent by all variants of a process are identical, in order to allow greater diversity. For example, Lamport's distributed mutual exclusion algorithm [14] sends in `ack` messages the current value of the sender's logical clock, whereas the variant in [17, Fig. 3] sends in `ack` messages the logical time of the request being acknowledged. To support algorithm variants that have the same communication pattern but different message content, we modify the gateway to omit the equality check on outbound messages when the destination is a diversified process, in which case the gateway sends to the other gateway an array containing the message from each variant, which forwards each message in the array to its corresponding variant. The correspondence is determined by indexing variants in the order that their types are listed in the configuration file.

To support algorithm variants with different communication patterns, the configuration file can specify that certain types of messages are *un-synchronized*. When the gateway receives a message of an un-synchronized type from its i'th variant, it

immediately forwards the message to the destination's gateway, which forwards the message to its i'th variant. For example, for synchronized execution of Lamport's distributed mutual exclusion algorithm and Ricart-Agrawala's distributed mutual exclusion algorithm [29], we specify that `ack` and `release` messages (used only in Lamport's algorithm) and `response` messages (used only in Ricart-Agrawala's algorithm) are un-synchronized; the gateway still synchronizes messages of other types.

5 Diversity Metrics and Runtime Monitoring Tools

5.1 Code Diversity

Since diversity is the complement of similarity, we measure code diversity with a well-established document similarity metric, namely, n-gram similarity with winnowing [31], which is used in the popular software plagiarism detection tool Moss to measure similarity of program source code. We apply it to Python bytecode, specifically, the sequence of bytecode instructions in a compiled program. Bytecode similarity is more relevant than source-level similarity, because diversity at the Python level aims to increase resilience to flaws in the runtime system, and bytecode is the program representation used by the runtime system.

An *n-gram* is a sequence of n consecutive instructions, starting at any position. The algorithm computes the hash of every n-gram in the compiled program, and then (for scalability) selects a subset of those hashes and stores them in a set called the program's *fingerprint*. The number of selected hashes is controlled indirectly by an algorithm parameter w called the *window size*. A window of size w consists of the hashes of w consecutive n-grams in the program. The winnowing algorithm is guaranteed to select at least one hash from each window of size w, although it may select more.

A robust metric should have the property that a slightly modified program has high similarity to the original program. In Python bytecode, local variables and global variables are identified by index. Inserting one new global variable at the beginning of the program causes renumbering of all global variables; this could make the metric non-robust. To ensure robustness, we normalize variable indices within each n-gram: we re-index the first global variable accessed in the n-gram as 0, the second one as 1, etc., and similarly for local variables. For similar reasons, we replace absolute line numbers used as targets in jump instructions with a place holder.

We quantify code diversity (and similarity) of two programs as 1 minus the Jaccard similarity of their fingerprints. Recall that the Jaccard similarity of sets S and T is $|S \cap T|/|S \cup T|$. We use 1 minus Jaccard similarity so larger values indicate greater diversity.

An alternative to n-gram similarity is Levenshtein distance (a.k.a. edit distance, namely, the minimum number of single-element insertions, deletions, and substitutions needed to change one string to another) between the bytecode sequences in the compiled programs. Levenshtein distance is less suitable here, because it is sensitive to bytecode orderings in the compiled program that may be unimportant at runtime. For example, permuting the order in which function definitions appear in the compiled program has no effect on the program's runtime behavior but has a large effect on the Levenshtein distance. Similarly, swapping the branches in a conditional statement and

negating the condition yields an equivalent program with high n-gram similarity to the original but (if the branches are large) a large Levenshtein distance from the original.

5.2 Trace Diversity

Trace diversity measures the similarity of the sequences of bytecode instructions executed by two programs. Our bytecode-level tracing tool uses the standard Python `trace` module to obtain a source-level trace, and then translates it to a bytecode trace. A "blacklist" of modules to be ignored during the conversion can be specified; in experiments, we blacklist some system modules, such as `bootstrap` and `trace`. For each source line mentioned in the trace, identified by filename and line number, the translator compiles that .py file to a .pyc file, loads the .pyc file using the `marshal` module to obtain a `code` object, repeatedly uses the `dis` (disassembler) module to obtain the bytecode for the entire program as a list of `Instruction` objects, and uses the source line number information in the `Instruction` objects to determine the sequence of instructions corresponding to each line of source code in that file. In the traces to be compared, we include only the `opcode` and `argument` attributes of each `Instruction`; other attributes (e.g., `is_jump_target`) are less important. We quantify similarity of two traces as the Levenshtein distance (edit distance) between them divided by their average length, for normalization.

5.3 Input Access Diversity

Input access diversity measures the similarity of sequences of accesses to input data by two programs, quantified as Levenshtein distance between the sequences divided by their average length, for normalization. The core of the implementation is a general tool to intercept accesses to attributes of selected objects, by overriding the `__getattribute__` method of appropriate classes. In our use case, the overriding method logs the access and then calls the original `__getattribute__` method. For user-defined classes, this is easily accomplished by inserting a definition of `__getattribute__` in the class. This approach does not work for built-in types such as `int`, `string`, and `tuple`, which are common types of input data.

For each of these built-in classes, we define a new class, e.g. `tracked_int` for `int`, that inherits from the built-in class and overrides the `__getattribute__` method. In the remainder of the description, we focus on `int`; other built-in types are handled similarly. The problem is that some accesses to attributes of `tracked_int` are not logged, because attributes of built-in types are sometimes accessed directly by C code in the CPython runtime system. For example, even if `x` is a `tracked_int`, the addition operator in `x+y` compiles to the bytecode instruction `BINARY_ADD`, which does not call `__getattribute__` on either argument.

We overcome this problem by augmenting `tracked_int` to override all methods of `int` that access the integer value: `__add__`, `__eq__`, `__le__`, etc. If `x` is a `tracked_int`, an expression like `x+y` now compiles to bytecode that uses the `CALL_FUNCTION` instruction to explicitly invoke `x`'s `__add__` method with argument `y`. The `tracked_int.__add__` method logs the access to the first argument (`self`), calls `__getattribute__` on the second argument (so the access to it will be logged, if it is a `tracked_int`), and then calls

the built-in `__add__` method. Since we need to override these operations anyway, we augment log entries to indicate which operation was performed on the accessed attribute.

If `x` is an `int`, not a `tracked_int`, then `CALL_FUNCTION` invokes the built-in `__add__` method, which is implemented by C code that accesses the second argument without calling `__getattribute__`. Consequently, accesses to `y` are not logged, even if `y` is a `tracked_int`. To overcome this remaining problem, we modify the program to replace the remaining uses of `int` with a new class `my_int`, which inherits from `int` and overrides each two-argument method of `int` with a method that calls `__getattribute__` on the second argument and then calls the original method. To accomplish this replacement, we bind the name `int` to our class `my_int`, using the assignment `int = my_int`. As a result, a constructor call such as `int(1)` returns an instance of `my_int`. The literal `1` still produces an `int`. Therefore, we transform the source program to replace literals with constructor calls, e.g., `1` with `int(1)`.

The remaining aspects of input access tracking differ for Python and DistAlgo. These aspects are (1) determining which objects are tracked, and (2) creating meaningful identifiers for tracked objects. We could easily use the result of Python's built-in `id` function to identify objects, but it would be difficult to compare input access traces from different variants (or even different runs of the same variant), because the object identifiers in them would be unrelated. Instead, we create object identifiers that can be compared meaningfully with object identifiers in other logs, as described below. The identifier is stored in an attribute of each tracked object.

Python. For Python programs, the user specifies which objects should be tracked by modifying the program to make them instances of tracked classes. For convenience, our `tracker` class provides a method that recursively traverses an object or collection (dictionary, list, tuple, or set) and replaces all instances of trackable built-in types (i.e., types for which a corresponding tracked type exists) with instances of tracked types. In our benchmark programs, inserting one or two calls to this method suffices. Tracked objects are identified by a sequence number assigned in the order that the objects are created. When tracked objects are used for data read as input, these identifiers are meaningful across logs from different variants, because the variants are given the same inputs and hence read the inputs in the same order.

DistAlgo. For DistAlgo programs, all messages are automatically considered as inputs; additional inputs, if any, are handled as described above for Python programs. Instances of trackable built-in types in messages are automatically replaced with instances of tracked types. Our `tracker` class, which inherits from DistAlgo's `process` class, is automatically inserted as a parent class of every process class in the given program. It overrides `process.send` with a method that replaces all instances of trackable built-in types in the message with instances of tracked types.

To create meaningful identifiers for tracked objects received in messages, we observe that such an identifier should identify the message in which the object was received. Our identifier for such an object is a tuple (*host*, *procNum*, *msgNum*, *objNum*), where *host* is the host on which the sender is running, *procNum* identifies the sending process relative to the host, *msgNum* identifies the message relative to the sending process, and *objNum* identifies the object within the message.

To avoid dependence on standard process identifiers that cannot be meaningfully compared across executions, we identify processes by a sequence number assigned in the order in which the processes are created. The `tracker` class overrides `process.setup` with a method that assigns the process sequence number; `tracker.setup` stores the next available process sequence number in a local file. *msgNum* is a per-sender sequence number assigned in the order in which messages are sent. The object sequence number *objNum* is assigned to each object in the message in the order that the object is encountered in a depth-first traversal of the message.

Input access logs for DistAlgo programs also contain entries corresponding to `receive` events, so we can determine that a particular data item (possibly received in a previous message and stored in a data structure) was accessed while processing a particular message.

6 Evaluation

We evaluated our approach on several sequential and distributed algorithms, using Python 3.7.2 and DistAlgo 1.1.0b13. Our software is available at `https://www.cs.stonybrook.edu/~stoller/software`.

For each problem and each diversity metric, we measure the diversity achieved (1) by algorithm diversity alone by averaging the diversity metric for each pair of algorithms; (2) by implementation-level diversity (ILD) alone by averaging the diversity metric for each pair of an algorithm and its ILD variant (i.e., the variant obtained by applying ILD transformations to it); (3) by both forms of diversity together by averaging the diversity metric for each pair of an algorithm and the ILD variant of another algorithm. For code diversity, we used $n = 5$ (the value used in [31]), and we disabled winnowing (i.e., included all hashes in the fingerprint), because the bytecode for our examples is not too large. Library code is not included in our code diversity measurements.

Implementation-Level Diversity (ILD). We created ILD by applying these typical ILD transformations: (1) NOP insertion: after each line of code, insert a `pass` statement with probability 0.05; (2) instruction reordering: for each two adjacent independent lines of code, swap them with probability 0.5; (3) branch reordering: for each if-statement, swap the branches and negate the condition (if there is no `else` branch, pretend `else: pass` is present) with probability 0.5; (4) function (including `receive` handler) reordering: for each two adjacent independent `def` statements, swap them with probability 0.5; (5) argument reordering: for each function (excluding `run`, `setup`, and receive handlers), swap the first two arguments, swap the third and fourth arguments, etc.; (6) field reordering: reorder the assignment statements that initialize the fields in each class, by swapping the first two, the third and fourth, etc. Applying more complicated implementation-level diversity techniques is future work; it will require significant effort, because existing implementations of those techniques do not handle Python.

6.1 Sequential Algorithms

Our experiments use these algorithms for these problems: (1) graph reachability: original (iterative) algorithm, incrementalized algorithm, and rule-based algorithm

Metric	Level	reach. 3 variants	Hanoi 4 variants	LCS 3 variants	pat. search. 3 variants	sort 4 variants	tree search 6 variants	Avg.
Code	algo	0.80	0.58	0.65	0.81	0.79	0.83	0.74
Diversity	impl	0.40	0.39	0.66	0.52	0.32	0.63	0.49
	both	0.80	0.65	0.82	0.83	0.80	0.89	0.80
Input Access	algo	1.04	0.54	0.58	0.28	0.77	0.35	0.59
Diversity	impl	0	0.18	0.82	0.21	0	0	0.20
	both	1.04	0.57	1.12	0.28	0.77	0.36	0.69
Trace	algo	1.45	0.42	1.22	0.69	0.81	0.80	0.90
Diversity	impl	0.05	0.30	0.60	0.23	0.11	0.14	0.23
	both	1.45	0.45	1.39	0.70	0.82	0.82	0.94

Table 1. Experimental results for sequential algorithms. In the "Level" column, "algo" and "impl" denote algorithm and implementation-level diversity, respectively. The last column contains averages.

(generated from rules using the method in [21]); (2) Hanoi Tower: original recursive algorithm, optimized recursive algorithm, optimized iterative algorithm, and optimized iterative algorithm with swap; (3) longest common subsequence (LCS): original recursive algorithm, optimized recursive algorithm, and optimized iterative algorithm; (4) pattern searching: naive algorithm, Knuth Morris Pratt (KMP) algorithm, Rabin Karp algorithm; (5) sorting: heap sort, quicksort, insertion sort, and merge sort; (6) tree search: recursive and iterative algorithms for AVL trees, recursive algorithm for B-trees, iterative algorithm for red-black trees, and recursive and iterative algorithms for (unbalanced) binary search trees.

The results are in Table 1. We see from the last column that, for all three metrics, algorithm diversity creates more diversity than ILD, and that the two together create even more.

6.2 Distributed Algorithms

Our experiments use the following algorithms: (1) 2-phase commit (2PC); (2) Hirschberg-Sinclair's leader election (HSleader) [11]; (3) Lamport's distributed mutual exclusion (Lamutex) [14]; (4) Lamport's basic Paxos [15]; (5) Ricart-Agrawala's distributed mutual exclusion (RAmutex) [29]. We used configurations with 3 or 4 processes for each algorithm. There are two variants of each algorithm: one variant that uses high-level queries over message histories, and one that explicitly maintains the result of those queries (and related intermediate results and auxiliary values), updating them in assignment statements, especially in `receive` handlers.

When measuring the dynamic metrics, we avoid spurious differences between the variants due to the platform's scheduling variability by running all variants in parallel using synchronized execution (for programs other than 2PC, due to a bug that we are still resolving in the interaction between our program transformations for synchronized execution and input access tracing, when measuring input access diversity, we instead avoided such spurious differences by running the variants separately but each with the

Metric	Level	2PC	HSleader	Lamutex	Paxos	RAmutex	Average
Code	algo	0.56	0.66	0.50	0.68	0.53	0.59
Diversity	impl	0.19	0.18	0.08	0.30	0.27	0.21
	both	0.59	0.68	0.53	0.68	0.54	0.60
Input Access	algo	1.10	0.47	0.21	0.28	0.61	0.53
Diversity	impl	0.08	0.04	0	0.03	0.17	0.06
	both	1.09	0.52	0.21	0.30	0.61	0.55
Trace	algo	0.20	0.35	0.13	0.54	0.21	0.29
Diversity	impl	0.06	0.03	0.02	0.13	0.04	0.06
	both	0.20	0.36	0.14	0.52	0.21	0.29

Table 2. Experimental results for distributed algorithms, with 2 variants for each algorithm. In the "Level" column, "algo" and "impl" denote algorithm diversity and implementation-level diversity, respectively.

same pattern of injected message delays that are larger than the platform's scheduling variability and designed to avoid races in message delivery order).

The results are in Table 2. We see from the last column that, for all three metrics, algorithm diversity creates significantly more diversity than ILD. The trace diversity produced by ILD is considerably smaller than the input access diversity it creates. These results are not inconsistent, because both are measured as ratios, and input accesses constitute a small fraction of the program's full activity recorded in the bytecode trace. The results for trace diversity for algorithm diversity for distributed algorithms are notably smaller than for sequential algorithms, because the trace includes execution of DistAlgo runtime library for networking, which is not diversified.

7 Related Work

Existing techniques for automated software diversity, including all those surveyed in [16], create implementation-level diversity, changing details of the implementation without changing the algorithm. Typically this is done by applying relatively simple local transformations, like those used in our evaluation. There are also some complex global transformations, such as instruction set randomization. These transformations are fully automated and more easily applied to large programs, but they are limited in that they do not create algorithm diversity. For example, they do not change the pattern in which inputs are used by the program.

Most work on automated software diversity for resilience transforms C programs or (disassembled) machine code, for broader applicability to systems code. There is some work on automated diversity for programs in JIT-compiled high-level languages, which diversifies the machine code generated by the JIT compiler. For example, librando does this for Java and JavaScript [12], and INSeRT does this for JavaScript [32]. This low-level approach is suitable for creating implementation-level diversity. Our methodology diversifies the high-level program directly to create algorithm diversity.

In N-version programming [1], N versions of a system (or component) are created by separate and independent manual design and implementation efforts starting

from the same requirements specification, and the versions are run in parallel with synchronized execution. The goal is resilience in the presence of design faults, since independent teams are less likely to make the same design mistakes. Our work, like other work on software diversity, aims to mitigate software vulnerabilities, not design errors. The two techniques could be used together to address both. N-version programming may introduce algorithm diversity, but not in a controlled way, and at the cost of significant manual effort. In contrast, our approach is to create variants using a program transformation and optimization method based on systematic incrementalization, which guides the process, helps control how much diversity is introduced, and helps ensure correctness compared to ad-hoc development of variants. Our program transformation system InvTS [19,9] provides semi-automated support for the method, significantly reducing manual effort.

Synchronized execution has been widely studied in the fault-tolerance community, where it is often called *N-version execution*. N-version execution frameworks typically work at the system-call level, so they can be applied to software running on a given operating system, regardless of the application programming language. Our synchronized execution framework is applicable only to applications written in DistAlgo, but it is more portable and lighter weight. It can be used on any OS supported by DistAlgo (Windows, macOS, Linux, and Android), while system-call based approaches are highly OS-specific, e.g., Varan [13] and Bunshin [33] are N-version execution frameworks for Ubuntu. It is lighter-weight because a single high-level synchronization event is typically implemented by multiple system calls.

7.1 Evaluation of Diversity Techniques

A few approaches are commonly used to evaluate implementation-level diversity techniques. One is to estimate the probability of a successful memory-related exploit (e.g., buffer overflow or format string attack) based on the information about the diversified program that the attacker would need to guess, more specifically, the type of information (e.g., the address of a specific object, or the difference between the addresses of two specific objects) and the number of possible values of that type of information due to the randomization in the diversity transformation. This approach is used in, e.g., [2,8].

Diversity techniques designed specifically to defend against ROP attacks are typically evaluated using a *coverage metric* that measures the fraction of ROP gadgets relocated by the transformation, and sometimes also an *entropy metric* that measures the number of possible new positions of the ROP gadgets, reflecting the probability of the attacker correctly guessing the new locations. This approach is used in, e.g., [28,12]).

These approaches based on specific vulnerabilities in low-level languages are unsuitable for evaluating diversity for interpreted languages, such as Java and Python.

Acknowledgements. This material is based on work supported in part by ONR Grant N00014-15-1-2208, NSF Grants CCF-1414078 and CNS-1421893, and DARPA Contract FA8650-15-C-7561. We thank Thang Bui, Rahul Gadi, Shikhar Sharma, Shalaka Sidmul, Shubham Singhal, and Swetha Tatavarthy for their contributions to the implementation and experiments.

References

1. Avizienis, A.: The N-version approach to fault-tolerant software. IEEE Trans. Softw. Eng. **11**(12), 1491–1501 (Dec 1985)
2. Bhatkar, S., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: 14th USENIX Security Symposium. USENIX Association (2005)
3. Cai, J., Facon, P., Henglein, F., Paige, R., Schonberg, E.: Type analysis and data structure selection. In: Möller, B. (ed.) Constructing Programs from Specifications, pp. 126–164. North-Holland (1991)
4. Chen, S., , Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: 14th USENIX Security Symposium. USENIX (August 2005)
5. Cohen, F.B.: Operating system protection through program evolution. Computers and Security **12**(6), 565–584 (Oct 1993)
6. Forrest, S., Somayaji, A., Ackley, D.H.: Building diverse computer systems. In: 6th Workshop on Hot Topics in Operating Systems (HotOS). pp. 67–72 (1997)
7. Gautam, Rajopadhye, S.: Simplifying reductions. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on. pp. 30–41 (2006)
8. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: 21st USENIX Security Symposium. pp. 475–490. USENIX Association (2012)
9. Gorbovitski, M., Liu, Y.A., Stoller, S.D., Rothamel, T.: Composing transformations for instrumentation and optimization. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation. pp. 53–62 (2012)
10. Goyal, D.: A Language Theoretic Approach to Algorithms. Ph.D. thesis, Department of Computer Science, New York University (2000)
11. Hirschberg, D.S., Sinclair, J.B.: Decentralized extrema-finding in circular configurations of processors. Communications of the ACM **23**(11), 627–628 (1980)
12. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Librando: transparent code randomization for just-in-time compilers. In: ACM Conference on Computer and Communications Security. pp. 993–1004. ACM (2013)
13. Hosek, P., Cadar, C.: Varan the unbelievable: An efficient n-version execution framework. In: 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15). pp. 339–353 (3 2015)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
15. Lamport, L.: Paxos made simple. SIGACT News (Distributed Computing Column) **32**(4), 51–58 (2001)
16. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 276–291 (2014)
17. Liu, Y.A.: Logical clocks are not fair: What is fair? A case study of high-level language and optimization. In: Proceedings of the Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems. Egham, U.K. (July 2018)
18. Liu, Y.A., Brandvein, J., Stoller, S.D., Lin, B.: Demand-driven incremental object queries. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming. pp. 228–241. ACM Press (2016)
19. Liu, Y.A., Gorbovitski, M., Stoller, S.D.: A language and framework for invariant-driven transformations. In: Proceedings of the 8th International Conference on Generative Programming and Component Engineering. pp. 55–64. ACM Press (2009)

20. Liu, Y.A., Stoller, S.D.: From recursion to iteration: what are the optimizations? In: 2000 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). Boston (Jan 2000), published in *ACM SIGPLAN Notices*, February 2000
21. Liu, Y.A., Stoller, S.D.: From Datalog rules to efficient programs with time and space guarantees. ACM Transactions on Programming Languages and Systems **31**(6), 1–38 (Aug 2009)
22. Liu, Y.A., Stoller, S.D., Gorbovitski, M., Rothamel, T., Liu, Y.E.: Incrementalization across object abstraction. In: Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 473–486 (2005)
23. Liu, Y.A., Stoller, S.D., Li, N., Rothamel, T.: Optimizing aggregate array computations in loops. ACM Transactions on Programming Languages and Systems **27**(1), 91–125 (Jan 2005)
24. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. ACM Transactions on Programming Languages and Systems **39**(3), 12:1–12:41 (May 2017)
25. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. In: Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. pp. 395–410 (2012)
26. Liu, Y.A.: Systematic Program Design: From Clarity To Efficiency. Cambridge University Press (2013)
27. Paige, R., Koenig, S.: Finite differencing of computable expressions. ACM Transactions on Programming Languages and Systems **4**(3), 402–454 (1982)
28. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: 33rd IEEE Symposium on Security and Privacy. pp. 601–615. IEEE Computer Society (2012)
29. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM **24**(1), 9–17 (1981)
30. Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K.Z., Polychronakis, M.: Revisiting browser security in the modern era: New data-only attacks and defenses. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017. pp. 366–381. IEEE (2017)
31. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: Local algorithms for document fingerprinting. In: 2003 ACM SIGMOD International Conference on Management of Data. pp. 76–85. ACM (2003)
32. Wei, T., Wang, T., Duan, L., Lu, J.: INSeRT: Protect dynamic code generation against spraying. In: International Conference on Information Science and Technology. pp. 323–328. IEEE (March 2011)
33. Xu, M., Lu, K., Kim, T., Lee, W.: Bunshin: Compositing security mechanisms through diversification. In: USENIX Annual Technical Conference. pp. 271–283. USENIX Association (2017)