NetCheck: Network Diagnoses from Blackbox Traces

Yanyan Zhuang, *Polytechnic Institute of New York University and University of British Columbia;* Eleni Gessiou, *Polytechnic Institute of New York University;* Steven Portzer, *University of Washington;* Fraida Fund and Monzur Muhammad, *Polytechnic Institute of New York University;* Ivan Beschastnikh, *University of British Columbia;* Justin Cappos, *Polytechnic Institute of New York University*

https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zhuang

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

NetCheck: Network Diagnoses from Blackbox Traces

Yanyan Zhuang[†‡*], Eleni Gessiou[†*], Steven Portzer[◁], Fraida Fund[†],
Monzur Muhammad[†], Ivan Beschastnikh[‡], Justin Cappos[†]
[†]*NYU Poly,* [‡]*University of British Columbia,* [◁]*University of Washington*

Abstract

This paper introduces NetCheck, a tool designed to diagnose network problems in large and complex applications. NetCheck relies on blackbox tracing mechanisms, such as `strace`, to automatically collect sequences of network system call invocations generated by the application hosts. NetCheck performs its diagnosis by (1) totally ordering the distributed set of input traces, and by (2) utilizing a network model to identify points in the totally ordered execution where the traces deviated from expected network semantics.

Our evaluation demonstrates that NetCheck is able to diagnose failures in popular and complex applications without relying on any application- or network-specific information. For instance, NetCheck correctly identified the existence of NAT devices, simultaneous network disconnection/reconnection, and platform portability issues. In a more targeted evaluation, NetCheck correctly detects over 95% of the network problems we found from bug trackers of projects like Python, Apache, and Ruby. When applied to traces of faults reproduced in a live network, NetCheck identified the primary cause of the fault in 90% of the cases. Additionally, NetCheck is efficient and can process a GB-long trace in about 2 minutes.

1 Introduction

Application failures due to network issues are some of the most difficult to diagnose and debug. This is because the failure might be due to in-network state or state maintained by a remote end-host, both of which are invisible to an application host. For instance, data might be dropped due to MTU issues [26], NAT devices and firewalls introduce problems due to address changes and connection blocking [11], default IPv6 options can cause IPv4 applications to fail [8], and default buffer size settings can cause UDP datagrams to be dropped or truncated [49].

Such application failures are challenging for developers and administrators to understand and fix. Hence, numerous fault diagnosis tools have been developed [3, 13, 17, 37, 23, 19]. However, few of these tools are applicable to large applications whose source code is not available. Without source code, administrators often resort to probing tools such as `ping` and `traceroute`, which can help to diagnose reachability, but cannot diagnose application-level issues.

This paper presents NetCheck. In contrast with most prior approaches, NetCheck does not require application- or network-specific knowledge to perform its diagnoses, and no modification to the application or the infrastructure is necessary. NetCheck treats an application as a blackbox and requires only a set of system call (syscall) invocation traces from the relevant end-hosts. These traces can be easily collected at runtime with standard blackbox tracing tools, such as `strace`. To perform its diagnosis, NetCheck derives a global ordering of the input syscalls by simulating the syscalls against a network model. The model is also used to identify those syscalls that deviate from expected network semantics. These deviations are then mapped to a diagnosis using a set of heuristics.

NetCheck diagnosis output is intended for application developers and network administrators. NetCheck outputs high-level diagnosis information, such as “an MTU issue on a flow is the likely cause of loss,” which may be useful to network administrators. NetCheck also outputs detailed low-level information about the sequence of system calls that triggered the high-level diagnosis. This information can help developers locate the underlying issue in the application code.

This work makes the following three contributions:

- **Accurate diagnosis of network issues from plausible global orderings.** Because of complex network semantics, it is not always possible to globally order an input set of host traces without a global

[*] The two authors are co-primary authors.

clock. NetCheck approximates the true ordering by generating a plausible ordering of the input traces. We show that for 46 of the bugs reproduced from public bug-trackers, this strategy correctly detected and diagnosed over 90% of the bugs. Additionally, NetCheck found and diagnosed a new bug in VirtualBox [49].

- **Modeling expected network behavior to identify unexpected behavior.** By using a model of an idealized network environment NetCheck is capable of diagnosing issues even in applications that execute in complex environments. We demonstrate that this approach is effective at detecting many real-world problems, including failures reported in bug trackers of projects like Python and Apache, and problems in everyday applications such as Pidgin, Skype and VirtualBox.

- **Efficient algorithm for finding plausible global orderings.** We present a heuristic trace-ordering algorithm that utilizes valuable information inherent in network API semantics. We prove that our algorithm has a best-case linear running time and demonstrate that NetCheck needs less than 1 second to process most of the traces studied in this paper (Section 6.4). Even on large traces, such as a 1 GB trace collected from Skype, NetCheck completes in less than two minutes.

The following section provides an overview of NetCheck. Section 3 describes the challenges and corresponding contributions of this work. Details of NetCheck's design and implementation are given in Sections 4 and 5, respectively. In Section 6, we evaluate the accuracy, effectiveness, and efficiency of NetCheck. Section 7 outlines limitations of NetCheck and our future work. Related work is discussed in Section 8 and we conclude with Section 9.

2 NetCheck Overview

To use NetCheck, a user needs to first gather a set of *host traces* for an application using a tool like `strace`, `dtrace`, `ktrace`, or `truss`. The user invokes NetCheck with a configuration file that lists the host trace files to analyze and the IP addresses of the hosts. A host trace, as in Figure 1, is a sequence of *syscall invocations* at a single host. A syscall invocation is a 4-tuple that includes (1) a string, such as `socket` denoting the name of the syscall, (2) the arguments passed to the invoked syscall, (3) the returned value, and optionally, (4) an error number (errno) that is returned upon a syscall failure.

For example, the first line of the host A trace in Figure 1 is `socket(...)=4`, which is a `socket` syscall invocation with a return value of 4. For certain syscalls the value is returned through an argument pointer[1]. Figure 1 shows an example of this: `recv` call on host A passes a buffer to a location in memory where the kernel writes a 5-byte string indicated by one of the logged arguments ("Hola!"). For clarity we omit some arguments and errno from syscall invocations in this paper.

```
     Host A trace:                           Host B trace:
A1. socket(...)            = 4    B1. socket(...)            = 3
A2. bind(4, ...)           = 0    B2. connect(3, ...)        = 0
A3. listen(4, 1)           = 0    B3. send(3, "Hello", ...) = 5
A4. accept(4, ...)         = 6
A5. recv(6, "Hola!", ...) = 5
```

Figure 1: An example input trace detailing a TCP connection between two hosts. Many system call arguments are omitted for readability. Returned values (including buffer contents) are underlined. Data sent by host B ("Hello") has been modified in-transit before being received by host A ("Hola!").

```
A1. socket(...)          = 4
B1. socket(...)          = 3
A2. bind(4, ...)         = 0
A3. listen(4, 1)         = 0
B2. connect(3,...)       = 0
A4. accept(4, ...)       = 6
B3. send(3,"Hello",...) = 5
A5. recv(6,"Hola!",...) = 5
```

Figure 2: A valid global ordering of syscall invocations from the two host traces in Figure 1.

The example traces in Figure 1 indicate an error with the network. Host B sends a 5-byte string "Hello" to A, but A receives "Hola!", a different 5-byte string. Used independently, the two host traces are insufficient to identify this issue — the corresponding `send` and `recv` calls both returned successfully. To detect the problem, a developer must manually reason about both the order in which the calls occurred (their serialization) and the underlying behavior of the calls (their semantics). For the traces in Figure 1, the logical serialization of the two host traces reveals a semantic problem: what was received is different from what was sent. NetCheck automatically detects this and other issues by serializing the traces, simulating the calls, and then observing their impact on network and host state.

To detect and diagnose network problems such as the issue in Figure 1, NetCheck uses a **global ordering** that it automatically reconstructs from the input set of black-box host traces. Figure 2 shows one global ordering for the two input traces in Figure 1. A valid global ordering must preserve the local orders of host traces, and conform to the network API semantics. For example, the local ordering at host A in Figure 1 requires that `bind` occur after `socket` has returned successfully. And, `connect` at host B cannot be ordered before `listen` at host A, as such an ordering violates the network API se-

[1] NetCheck expects the host traces to include such return values, which are provided by most common tools.

```
     Host A trace:                   Host B trace:
A1. send("hello") = 5        B1. send("hi") = 2
A2. recv("hi")    = 2        B2. recv()     = -1, EWOULDBLOCK
```

(a) Two input host traces. All operations are performed on a single connected TCP socket.

```
    Valid ordering 1:                  Valid ordering 2:
B1. send("hi")    = 2               A1. send("hello")= 5
B2. recv()       = -1, EWOULDBLOCK  B1. send("hi")    = 2
A1. send("hello")= 5                A2. recv("hi")    = 2
A2. recv("hi")    = 2               B2. recv()     = -1, EWOULDBLOCK
```

(b) Two valid orderings of (a): (left) `recv` returned `EWOULDBLOCK` because the data has not been sent yet. (right) `recv` returned `EWOULDBLOCK` because the content is still in the network.

```
A1. send("hello") = 5
A2. recv("hi")    = 2
B1. send("hi")    = 2
B2. recv()        = -1, EWOULDBLOCK
```

(c) An invalid ordering of (a): data is received before being sent.

Figure 3: An example illustrating the ambiguity of reconstructing a valid order from two host traces considered by NetCheck. For the two host traces in (a), there are two possible valid orderings in (b). An invalid ordering, such as (c), will never be produced by NetCheck.

mantics.

NetCheck reconstructs the global ordering without relying on globally synchronized clocks or logical clocks [18, 31]. Both approaches require modification of the existing systems, and incur performance overhead and complexity[2]. Instead, NetCheck uses a **heuristic algorithm** and a **network model** to simulate and check if a particular ordering of syscall invocations is feasible. As a result, NetCheck has a higher level of transparency and usability.

Next, we overview the challenges that NetCheck faces in diagnosing network issues in complex applications, and then describe the contributions of our work.

3 Challenges and Contributions

Challenge 1. Accuracy: ambiguity in order reconstruction. Reconstructing a global order of traces collected from edge hosts without a globally synchronized clock is sometimes impossible. For example, Figure 3(b) lists two valid orderings of the traces in Figure 3(a). In the ordering shown on the left the `recv` call on B failed because it occurred before the `send("hello")` call on A. In the ordering shown on the right the `send("hello")` call on A occurred before the `recv` call on B, but network delay prevented B from receiving the message when `recv` was invoked. The ordering in Figure 3(c) is invalid since data must be sent before it is received. However, even if invalid orderings are eliminated, from the traces in Figure 3(a) it is impossible to tell if the `send` call on A occurred before or after the `recv` call on B. How can NetCheck diagnose issues without being able to reconstruct what actually happened?

Challenge 2. Network complexity: diagnosing issues in real networks. The host traces that we consider are blackbox traces: they omit information regarding the physical network or the environment in which the traces were collected. How can NetCheck diagnose network issues without this crucial information?

Challenge 3. Efficiency: exploring an exponential space of possible orderings. The space of the potential sequences is exponential in the length of the host traces and the number of hosts. Exhaustive exploration of this space to find an ordering is intractable even at small trace lengths (e.g., 30 – 100 syscalls). Real-world applications, such as a Pidgin client, make over 100K syscall invocations in a single execution. Given this huge space of possible orderings, how can NetCheck efficiently find problems in user applications?

NetCheck handles each of the above three challenges as follows:

Contribution 1. Deriving a plausible global ordering as a proxy for the ground truth. NetCheck approximates the true ordering by generating a plausible ordering of the input traces that preserves the host-local orderings of syscalls (Section 4.1). For this, NetCheck assumes that syscalls are atomic: a syscall runs to completion before the next syscall in the trace. Our evaluation shows that for 46 of the bugs reproduced from public bug-trackers, NetCheck correctly detects and diagnoses more than 90% of the problems (Sections 6.1 and 6.2). Additionally, NetCheck fails to find a plausible ordering in only 5% of the input traces that we studied in our evaluation.

Contribution 2. Modeling expected simple network behavior to identify unexpected behavior. NetCheck tackles the complexity of an application's execution environment by modeling an idealized network (Section 4.2). We rely on the fact that, from the network edge, network behavior can be described with a simple model due to the end-to-end principle. Our network model is based on Deutsch's Fallacies [15, 16, 39], and encodes misconceptions commonly held by developers, such as: the network is reliable, latency is zero, hosts communicate over a direct link, etc. NetCheck detects and diagnoses network problems and application failures by finding deviations from this ideal model of the network (Section 4.3). Our evaluation (Section 6) demonstrates that this approach is effective at detecting many real-world problems, including failures reported in bug trackers of projects like Python and Apache, and problems in everyday applications such as Pidgin, Skype and VirtualBox.

Contribution 3. A best-case linear time algorithm to find a plausible global ordering. We present a

[2] Over 90% of syscall invocations we observed completed in less than 0.1 ms. Widespread and practical clock-synchronization techniques do not provide a sufficiently fine timing granularity to unambiguously order traces from multiple hosts.

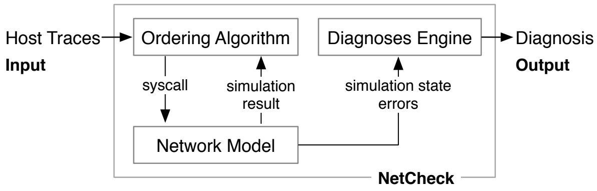


Figure 4: Overview of NetCheck.

Algorithm 1 NetCheck pseudo code.

```
1:  function NETCHECK(trace$_0$, ..., trace$_{n-1}$)
2:      // trace$_i$ is a list of syscall invocations at host i
3:      netModel = new POSIXNetworkModel()
4:      (orderError, permReject) = (False, False)
5:      try:
6:          // Call Algorithm 2 for trace ordering
7:          OrderingAlg(trace$_0$,...,trace$_{n-1}$, netModel)
8:      catch OrderError:
9:          orderError = True
10:     catch PermanentReject:
11:         permReject = True
12:     diagnosis = DiagnosesEngine(netModel.state,
                                    orderError, permReject)
13:     Output diagnosis
```

Algorithm 2 Trace ordering algorithm pseudo code.

```
1:  function ORDERINGALG(trace$_0$, ..., trace$_{n-1}$, netModel)
2:      // trace$_i$ is a list of syscall invocations at host i
3:      while (trace$_0$, ..., trace$_{n-1}$) not empty do
4:          // Record topmost calls
5:          topCalls = top(trace$_0$, ..., trace$_{n-1}$)
6:          // Assign each call a priority (see Table 1)
7:          topCalls.sort()
8:          // Process each call in topCalls or raise OrderError
9:          while True do
10:             if topCalls empty then // No calls can be processed
11:                 raise OrderError
12:             // Highest priority call remaining
13:             call$_i$ = topCalls.dequeue()
14:             outcome = netModel.simulate(call$_i$)
15:             if outcome == ACCEPT then // Completed this call
16:                 ordered_trace.push(trace$_i$.pop())
17:                 break // Break to outer while
18:             else if outcome == REJECT then
19:                 continue // Continue in inner while
20:             else // outcome == PERMANENT_REJECT
21:                 raise PermanentReject
22:         end while
23:     end while
```

heuristic trace-ordering algorithm that utilizes valuable information inherent in network API semantics. The best case running time of our algorithm is linear in the length of the input host traces. In the worst case, our algorithm is asymptotic with the length of the input host traces times the number of traces. Our evaluation demonstrates these bounds and illustrates that NetCheck needs less than 1 second to process most of the traces studied in this paper (Section 6.4). Even on large traces, such as a 1 GB trace collected from Skype, NetCheck completes in less than two minutes.

Next, we explain NetCheck's design in further detail.

4 NetCheck Design

First, NetCheck orders the syscalls with a heuristic algorithm and a network model. Following this, NetCheck uses a diagnoses engine to compile any detected deviations from the network model into a diagnosis. These steps are overviewed in Figure 4 and Algorithm 1. The next three sections describe each of these steps in further detail.

4.1 Ordering host traces

Algorithm 2 lists the pseudocode of the trace-ordering algorithm in NetCheck. The algorithm traverses the set of all input traces in local-host order, and at each iteration considers the calls that are at the top of each of the host traces (maintained in a priority queue `topCalls`, defined on line 5). In each iteration of the outer while loop (line 3), one of the calls from `topCalls` is processed. If this is not possible because no call at the top of the trace can be executed, an `OrderError` is raised. This completes execution of the ordering algorithm (returning to Algorithm 1) and thus no further calls are processed.

To find a call in `topCalls` to process, the algorithm performs two steps. First, the algorithm sorts `topCalls` (line 7) according to syscall priorities in Table 1. These priorities are derived from the dependency graph in Figure 5 (discussed below). Second, the algorithm simulates the highest priority call using the network model (see Section 4.2). This simulation can result in one of three outcomes: (1) accept the call (line 16) and continue with the outer while loop iteration, (2) reject the call (line 19) and then try another call in priority order from `topCalls`, or (3) a `PermanentReject` exception is raised (line 21) — the syscall can never be processed by the model, return back to the NetCheck algorithm (Algorithm 1 line 10).

Prioritizing syscalls. The key to the efficiency of the order reconstruction algorithm is to simulate syscalls in an order that is derived from the POSIX syscalls dependency graph in Figure 5. This graph was created by examining the POSIX specification for each system call and looking at which calls can modify the state used by other calls. This graph can be used to derive a priority value for each syscall (for simplicity we use integer priority values): if syscall `x` may-depend-on `y`, then `x` has a higher priority value and should be simulated before `y` (Table 1). For example, according to Figure 5, `connect` should be simulated before `listen`. This scheme is justified because processing a syscall `y` in the network model could affect `x` and make it impossible to simulate `x` without undoing `y`. This helps NetCheck to avoid significant

Priority value	Syscalls
0	socket, bind, getsockname, getsockopt, setsockopt
1	poll, select, getpeername
2	accept, recv, recvfrom, recvmsg, read
3	connect, send, sendto, sendmsg, write, writev, sendfile
4	close, shutdown, listen

Table 1: Simulation priority of common syscalls: the lower the priority value, the higher the priority.

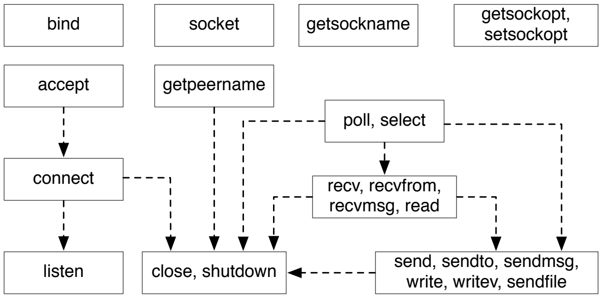


Figure 5: Dependency graph of system calls in the POSIX networking API. Edges represent the *may-depend-on* relation.

backtracking in the case where the return value of x requires that y has not yet occurred.

Syscall prioritization enables the ordering algorithm to permanently process a syscall after trying (in the worst case) the top syscall on each trace. The inner while loop (line 9) iterates through the top-most syscalls on each trace and removes them from a priority queue (thus considering each syscall at most once per inner loop execution). If this syscall is accepted (lines 15–17), then pop removes it from trace$_i$ (line 16) which permanently consumes the syscall (there is no way to later undo this call). If this call cannot currently be processed and is rejected (lines 18–19), then it will be placed again in topCalls (line 5) after a syscall in the current priority queue (and thus on the top of a trace) is consumed. Therefore, in the worst case, Algorithm 2 will never backtrack beyond the current rejected call. This makes the trace-ordering algorithm in Figure 2 *efficient* — its best-case running time is linear in the length of the input host traces if no backtracking occurs, and its worst-case running time is the length of the input host traces multiplied by the number of host traces i.e., the number of hosts (see Section 6.4).

4.2 Model-based syscall simulation

The network model component of NetCheck simulates syscalls (line 14 in Algorithm 2) to determine if a given syscall can be added as the next syscall in the global order. The network model treats the network and the application that generated the traces as a blackbox and requires no application-specific information.

To simulate a syscall, the model uses the current network and host states tracked by the model, and network semantics defined by the POSIX API. The network model state includes information related to the observed connections/protocols (e.g., pending or established TCP connections), buffer lengths and their contents, datagrams sent/lost, etc. Simulating a syscall with a model results in one of three determinations: accept the call, reject the call, or permanently reject the call.

```
A2. bind(3, ...)       = 0
A3. listen(3, 1)       = 0
B2. connect(3, ...)    = 0
```

(a) One valid ordering: all syscalls returned successfully.

```
A2. bind(3, ...)       = 0
B2. connect(3, ...)    = -1, ECONNREFUSED
A3. listen(3, 1)       = 0
```

(b) A second valid ordering: connect returned ECONNREFUSED.

Figure 6: An example that demonstrates how return values of syscalls can guide trace ordering.

A key technique used by the model to determine if a syscall can be accepted is to use the syscall's logged return and errno values. As an example, Figure 6 shows two possible orderings for bind and listen calls at host A, and a connect call at host B. For connect to return 0 (success), it is necessary for listen to have already occurred. So, a return value of 0 by connect indicates that (a) is a valid ordering. However, had connect at host B returned -1 (and a connection refused errno), then (b) would have been the valid ordering.

The current state of the model determines if the model can accept a syscall invocation with a specific return value. In certain cases the model can accept a syscall with a range of possible return values. For example, if the model's receive buffer for a connection has *n* bytes, then the return value of the read syscall (number of bytes read) may be a value between 0 and *n*.

We now explain how NetCheck produces one of the three outcomes through Figure 7, which illustrates how NetCheck processes the log in Figure 1. We use A.x to denote a syscall x at host A. When a call is accepted, the vertex of this syscall and all of its incoming edges are removed in the next step in the figure; the removed call is added to the final output ordering in Figure 7(l). If a call is rejected, the dashed-arrow *may-depend-on* relation edge between two syscalls is converted into a solid-arrow *depends-on* relation edge. The network model produces one of three outcomes:

Accept the syscall: the simulation of the syscall is successful, and the return value and errno match the logged values. In Figure 7(a), the syscalls in priority queue topCalls (defined on line 5 of Algorithm 2) are A.socket and B.socket, shaded in yellow. The priority of the two calls are the same, so either of them could be simulated. In this case, A.socket is simulated, accepted, and then removed from the trace. In Figure 7(a), this corresponds to removing the vertex and all its incoming edges to generate Figure 7(b). Similarly, B.socket and A.bind are accepted and removed from the traces in Figure 7(b) and (c), indicated by the bold circle around each syscall.

Reject the syscall: the network model cannot simulate the syscall because the model is not in a state that can accept the syscall with its logged return value and errno. This indicates that the syscall should be simulated at a later point. In Figure 7(d), the calls A.listen and B.connect are dependent according to Figure 5. In Table 1, connect has a higher priority than listen, so B.connect is simulated first. (This is necessary because of situations like Figure 6(b) where connect fails because listen was not yet called.) However, the network model rejects this syscall (indicated by a bold and red circle) because for B to successfully connect to A, A must be actively listening. In Figure 7(e), the directed edge from B.connect to A.listen indicates that B.connect should be ordered after A.listen. As A.listen is the next call with the highest priority in topCalls, it gets simulated and accepted — its vertex and all incoming edges are removed in Figure 7(f).

We describe all of the cases in which our NetCheck prototype rejects a syscall on our wiki[3].

Permanently reject the syscall: there are errors during the simulation of the syscall and the call can never be correctly simulated at a future point. In Figure 7(j) and (k), the network model first accepts B.send and then attempts to simulate A.recv. This triggers an error since the content in the receiving buffer of A, "Hola!", is different from the content in the send buffer. No additional system calls will allow A.recv to correctly complete in the future.

4.3 Fault diagnoses engine

When NetCheck finished processing the trace (Algorithm 1), either through consuming all actions, finding an order error, or permanently rejecting an action, the state of the model contains valuable information. The diagnoses engine analyzes the model simulation state and any simulation errors to derive a diagnosis. The diagnoses engine makes the simulation results more meaningful to an administrator who might be tasked with resolving the issue.

The diagnoses engine infers a diagnosis based on a set of rules. If the simulation state matches a rule, then the corresponding diagnosis is emitted. Table 2 summarizes the rules; our technical report contains example output for each of the rules [56]. Although the diagnosis rules are heuristics, Section 6 shows that these are effective at detecting problems in a wide range of applications.

Figure 8 lists an example of NetCheck output for the multibyte unit test from Table 4. In this test, the server incorrectly uses byte size to calculate the content-length of an HTTP header. This gives the wrong value for HTTP responses with multi-byte characters and the client fails to get the entire content that it requested. This fault was listed on the Ruby bug tracker (test case 2.16 in [56]).

The output in Figure 8 consists of three parts, each of which can be optionally omitted. Part (1) lists non-fatal errors uncovered through simulation by the network model in the form of a snippet of the valid ordering derived with *OrderingAlg* and any model deviations at the syscall level. This information is useful to application developers to understand the series of actions leading to the fault. In the figure, the model detected that there is still data remaining in the buffer at Browser even though both close on Browser and shutdown on Server returned successfully. Part (2) presents statistics for the observed connections, which may be useful for network administrators to perform performance debugging or see loss / MTU issues. Finally, part (3) present a high-level diagnosis summary generated by the diagnoses engine, which is of interest to all users of NetCheck. Part (3) of Figure 8 shows that the network connection with outstanding data has been shut down by the Browser. This is due to an application-level miscommunication between the Browser and Server.

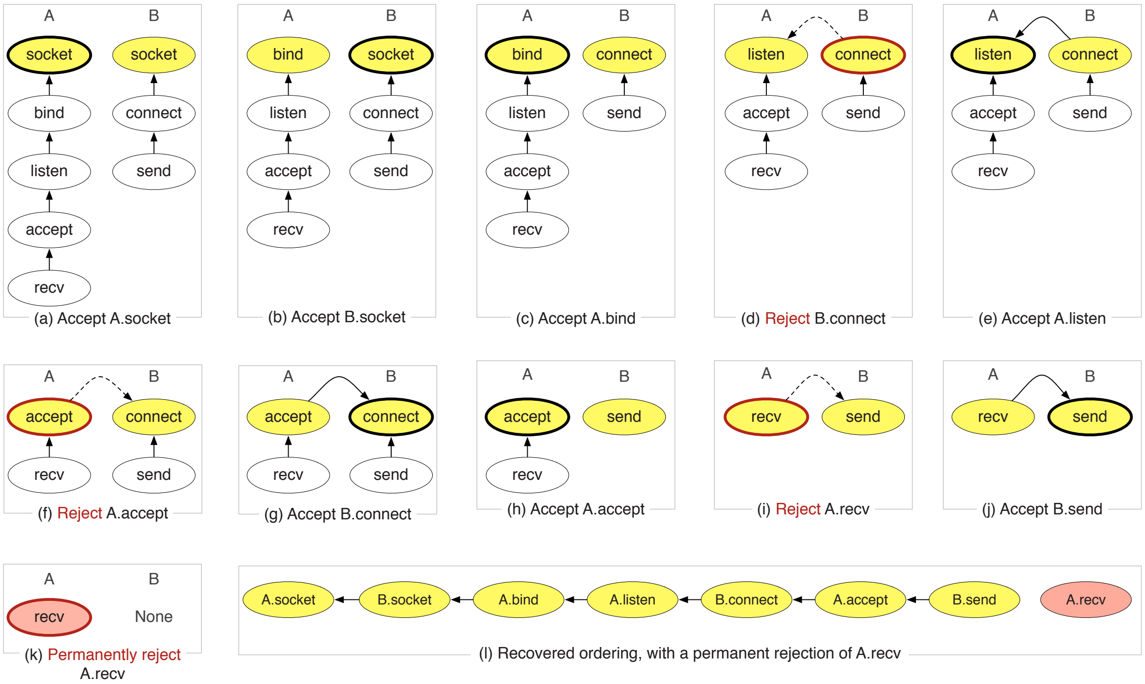
In larger applications, small network errors can accumulate to cause an application failure. For example, an MTU problem that can only be detected after considering a loss pattern across multiple transmission attempts. Packet loss is not unexpected as the network may drop packets, but diagnoses engine correctly diagnoses this as an MTU issue by considering the pattern of loss over the entire trace set (Section 6.2).

5 Implementation

NetCheck consists of 5.1K lines of Python code and is released freely under an MIT license [33]. The implementation supports the widely used POSIX network API, including support for common flags and optional arguments. This includes all of the syscalls that operate on file descriptors, and optional flags observed in traces of popular applications. It took 2 person-months to implement the network model.

NetCheck currently supports traces generated by strace on Linux. We are developing parsers for traces generated by other syscall tracing tools, such as truss on Solaris and dtrace on BSD and Mac OSX, to process these traces in a uniform manner [35].

[3] https://netcheck.poly.edu/projects/project/wiki/network_model



(a) Accept A.socket

(b) Accept B.socket

(c) Accept A.bind

(d) Reject B.connect

(e) Accept A.listen

(f) Reject A.accept

(g) Accept B.connect

(h) Accept A.accept

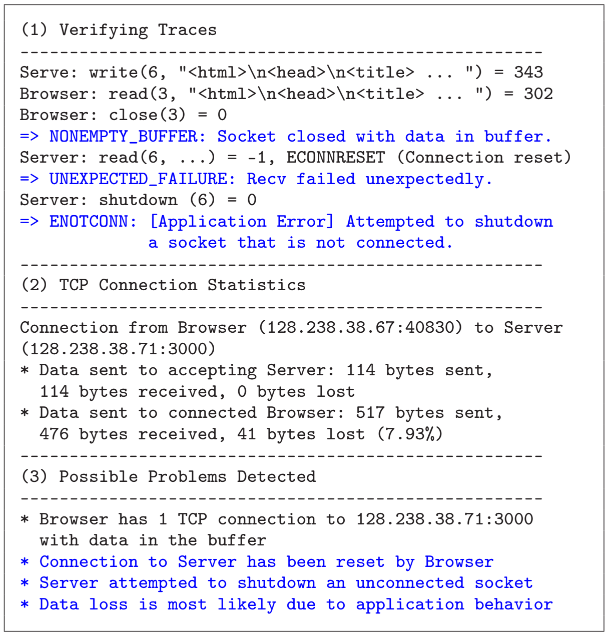
(i) Reject A.recv

(j) Accept B.send

(k) Permanently reject A.recv

(l) Recovered ordering, with a permanent rejection of A.recv

Figure 7: A step-by-step demonstration of how the ordering algorithm (Algorithm 1) processes the traces in Figure 1. Vertices are syscall invocations and solid edges represent dependency — capturing both the local ordering constraint and dependencies between remote syscall invocations. Shaded yellow vertices in each step represent the syscall invocations in the `topCalls` list in Algorithm 2. The syscall invocation simulated by the model at each step of the algorithm is circled in bold. A dashed edge denotes the may-depend-on relation from Figure 5. Each model simulation step either accepts or rejects a syscall. If accepted, the vertex of the syscall and all its incoming edges are removed in the next step, and the call is placed in the final output ordering (l). Steps (d), (f), and (i) show steps in which the syscall invocations were rejected (converting the may-depend-on relation edge into a depends-on relation edge). In step (k), the `A.recv` is permanently rejected because the traces in Figure 1 contain a bug: what was received is different from what was sent. The final output ordering in (l) orders all of the calls, except for the permanently rejected `A.recv`.



```
(1) Verifying Traces
-------------------------------------------------------
Serve: write(6, "<html>\n<head>\n<title> ... ") = 343
Browser: read(3, "<html>\n<head>\n<title> ... ") = 302
Browser: close(3) = 0
=> NONEMPTY_BUFFER: Socket closed with data in buffer.
Server: read(6, ...) = -1, ECONNRESET (Connection reset)
=> UNEXPECTED_FAILURE: Recv failed unexpectedly.
Server: shutdown (6) = 0
=> ENOTCONN: [Application Error] Attempted to shutdown
             a socket that is not connected.
-------------------------------------------------------
(2) TCP Connection Statistics
-------------------------------------------------------
Connection from Browser (128.238.38.67:40830) to Server
(128.238.38.71:3000)
* Data sent to accepting Server: 114 bytes sent,
  114 bytes received, 0 bytes lost
* Data sent to connected Browser: 517 bytes sent,
  476 bytes received, 41 bytes lost (7.93%)
-------------------------------------------------------
(3) Possible Problems Detected
-------------------------------------------------------
* Browser has 1 TCP connection to 128.238.38.71:3000
  with data in the buffer
* Connection to Server has been reset by Browser
* Server attempted to shutdown an unconnected socket
* Data loss is most likely due to application behavior
```

Figure 8: NetCheck's output for the `multibyte` unit test from Table 4.

Rule	Description
1. Unaccepted Connections	If a TCP connection has unaccepted (pending) connections, this is an indicator that the connecting host may be behind a middlebox.
2. Ignored Accepts	No matching connect corresponding for an accept (middlebox indicator).
3. Connect Failure	Connect fails for reasons other than (1) or (2), indicating that a middlebox (e.g., NAT) is filtering network connections.
4. Connection Refused	Connection is refused to an address that is being listened on (middlebox indicator).
5. Nonblocking Connect Failure	A nonblocking connect never connects (middlebox indicator).
6. No Relevant Traffic	A host has outgoing traffic, but not to a relevant address, then the host is likely connecting through a proxy.
7. Datagram Loss	A significant (user-defined) fraction of datagrams are lost.
8. MTU	Datagrams larger than a certain size are dropped by the network.
9. Non-transitive Connectivity	*A* can communicate with *B*, *B* can communicate with *C*, but *A* cannot communicate with *C*.

Table 2: NetCheck post-processing diagnoses. Example of rule (1): when a client is behind a NAT, (i) the client uses a private IP, (ii) the peer socket address in server's accept is not the client's IP.

6 Evaluation

We evaluated NetCheck in four ways. First, we examined network issues in popular applications reported on public bug trackers (Section 6.1). NetCheck diagnosed known bugs across multiple projects with a rate of 95.7%, demonstrating its **accuracy**. Second, we replicated failures on a real network, the WITest testbed [51], and used NetCheck to diagnose the issues (Section 6.2). This result indicates that we can diagnose real network issues as **deviations from a simple model of expected network behavior**. Third, we used NetCheck to diagnose the root cause of faults in widely-used applications, such as FTP, Pidgin, Skype and VirtualBox (Section 6.3). Finally, we evaluated NetCheck's performance across all of the test cases and applications detailed in our evaluation and proved that the algorithm has a best-case linear running time (Section 6.4). This demonstrates that NetCheck is **efficient**. A detailed description of all the bug traces in this section, and the corresponding bug reports are provided on our wiki [33] and in our technical report [56].

6.1 Diagnosing Bugs Reported in Bug Trackers

As noted in Section 3, the first challenge for NetCheck is to reconstruct a global ordering of traces. To evaluate NetCheck's **accuracy**, we collected bugs from public bug trackers of 30 popular projects. We targeted networked-related bugs that are small, reproducible, and have a known cause. We did not intentionally select bugs based on how NetCheck works. For each bug report we reproduced the issue from the report description by writing code to cause the observed behavior. This generated a total of 71 traces. Table 3 lists the results of running NetCheck on these traces. The traces are grouped into 24 categories based on the exhibited bug or behavior. Note that not all traces produce a bug. Some traces generate different behaviors on different OSes and can potentially lead to portability problems. For example, the `loopback_address` category captures the case when a socket exhibited different behaviors bound to the local interface (0.0.0.0 or 127.0.0.1). We briefly review four bugs from this evaluation.

1. MySQL provides a server-side option to only accept TCP connections from the loopback interface. If a user connects to the local IP address of the host that is hosting the MySQL server configured with this option, the connection is refused. This option provides extra security, but it is also difficult to debug. NetCheck correctly diagnoses the root cause as a socket bound to the loopback interface attempting to connect to a non-loopback address — `bind_local_interface` in Table 3.

2. When a Skype call's quality degrades, the user often terminates and restarts the call or restarts Skype. This may cause a known issue: when a TCP/UDP socket is waiting on `recv/recvfrom`, a `close` call made on the socket from a different thread will keep the socket blocking *indefinitely*. This bug has also been reported on GCC and Ruby bug trackers [10, 45]. We reproduced this bug, and NetCheck successfully diagnosed it (`block_tcp/udp_close_socket` in Table 3).

Bug category (number of traces)	**Bugs detected & correctly diagnosed / # Bugs**
bind_local_interface (2)	2 / 2
block_udp_close_socket (2)	1 / 1
block_tcp_close_socket (2)	1 / 1
broadcast_flag (2)	2 / 2
buffer_full (1)	1 / 1
invalid_port (3)	3 / 3
loopback_address (7)	0 / 0
multicast_issue (3)	3 / 3
multiple_bind (1)	1 / 1
nonblock_connect (13)	8 / 9
nonblock_flag_inheritance (2)	1 / 1
oob_data (5)	5 / 5
readline (1)	0 / 0
recvtimeo (1)	1 / 1
setsockopt_misc (3)	2 / 2
shutdown_reset (1)	0 / 1
sigstop_signal (3)	0 / 0
so_linger (5)	2 / 2
so_reuseaddr (2)	2 / 2
tcp_nodelay (3)	0 / 0
tcp_set_buf_size_vm (4)	4 / 4
udp_large_datagram_vm (2)	2 / 2
udp_set_buf_size_vm (2)	2 / 2
vary_udp_datagram (1)	1 / 1
Total Number of Traces: 71	**44 / 46 (95.7%)**

Table 3: Evaluating NetCheck on reported network bugs.

3. Different interpretations of network APIs have resulted in OS portability issues. For example, implementations of `accept` vary in whether file status flags, such as `O_NONBLOCK` and `O_ASYNC`, are inherited from a listening socket [1]. This has caused faults in a variety of applications, including Python's socket implementation [2, 34]. This issue is successfully diagnosed by NetCheck (`nonblock_flag_inheritance` in Table 3).

4. Variations in socket API implementations can also have security implications. On Windows, an application can exploit the `SO_REUSEADDR` socket option to deny access to, or impersonate, services listening on the same local address. Over a dozen major software projects [47, 43, 41, 50, 42, 36] include platform-specific code to mitigate security risks associated with `SO_REUSEADDR`. This issue is successfully diagnosed by NetCheck (`so_reuseaddr` in Table 3).

Overall, NetCheck correctly detected and diagnosed **95.7%** of the 46 reported bugs we considered. This indicates that NetCheck is **accurate**. Our technical report [56] further details the test cases in Table 3.

6.2 Diagnosing Injected Bugs in a Testbed

Deployed applications run in complex networking environments. To evaluate whether NetCheck can diagnose issues in real networks, we conducted an experiment on the WITest testbed [51] — a networking environment

Bug	Total number of diagnoses	Bug detected	Incorrect diagnoses
bind6	1	✓	0
bind6-2	1	✓	0
clientpermission	2	✓	0
closethread	3	✗	0
conn0	1	✓	0
firewall	2	✓	1
gethostbyaddr	1	✗	0
httpprox1	1	✓	0
httpprox2	1	✓	0
keepalive	1	✓	0
local	3	✓	0
max1	3	✓	0
maxconn	4	✓	0
maxconn2	6	✓	0
mtu	2	✓	0
multibyte	2	✓	0
noread	1	✓	0
permission	1	✓	0
portfwd	1	✓	0
special	1	✓	0
Total:	**38**	**18/20 (90%)**	**1/38 (3%)**

Table 4: NetCheck's classification of controlled network bugs. The total number of diagnoses are all the issues detected by NetCheck. The ✓/✗ indicate the success/failure of NetCheck in diagnosing the root cause of the bug. Incorrect diagnoses are false positives.

for studying wireless connectivity. We used a typical setup for this testbed, running a client-side application that issued requests to a WEBrick [48] HTTP server. For the experiment, an administrator replicated and injected network-related bugs from the WEBrick tracker into the testbed. We then gathered trace data from both hosts with `strace`. Table 4 lists the results from this evaluation. We now review two categories of bugs that NetCheck successfully diagnosed: IPv6 compatibility, and issues related to middleboxes.

IPv6 compatibility. With pervasive IPv6 deployment, applications are beginning to set IPv6-only socket options by default. However, IPv6 lacks backward compatibility and can break many legacy IPv4 applications [8]. For example, the IPv6-only option breaks the networking functionality in Java and triggers a "network unreachable" exception [21]. Other applications including Eclipse, VNC, Google Go, etc., will generate similar exceptions that are too generic to diagnose the root cause. NetCheck's model expects that IPv6 does not preempt IPv4 in an idealized environment, making it possible to detect this incompatibility by tracking inconsistent addressing schemes (`bind6`/`bind6-2` bugs in Table 4). As a result, NetCheck generates a much more informative diagnosis of this issue [56].

Middleboxes. Middleboxes, such as firewalls and NATs, introduce features that impact most network applications. NetCheck can be used to detect and diagnose the effects of middleboxes on an application. For example, in an FTP session, a `PASV` command on the client-side allows the server to define an IP/port that the client can use to connect to the server and receive data. The negotiated port is usually a high numbered port on the server that is typically blocked by firewalls on the server-side. NetCheck can detect and diagnose this kind of failure (`firewall` bug in Table 4).

Note that for many of the injected bugs, NetCheck also provided accurate supplementary information (included in the total diagnoses count in Table 4), such as packet loss information and buffer state. For example, if a connection is closed with data in the receive buffer, then NetCheck warns that the connection might have been closed prematurely by the receiver.

Our evaluation of injected bugs in a controlled network environment (Table 4) shows that despite the complexity of a real network, NetCheck's idealized network model is **effective**, diagnosing **90%** of the injected bugs with a false positive rate of 3%. The injected bugs used in this study are detailed in [56].

6.3 Bugs in Popular Applications

We also evaluated NetCheck on traces generated by large, widely used applications — an FTP client, Pidgin, Skype and VirtualBox. The issues detailed in this section were uncovered through normal, practical use of the applications on university and home networks. NetCheck produced useful diagnosis for problems across all of these applications (Table 5).

While using an **FTP client** to interact with an FTP server, we noticed that certain commands, such as `cd` and `pwd`, executed successfully, while others, like `ls` or `get`, would not be processed. When one of the latter command was issued, the FTP client was locked up until the connection timed-out. We used NetCheck to diagnose the issue by applying it to traces from the server and the client. NetCheck's diagnoses are summarized in Table 5. The problem is that commands like `ls` and `get` are followed by a `PORT` command from the client to inform the server about the IP/port that the client is listening on to receive the data (default behavior in most FTP clients). Since the client was behind a NAT, the destination IP address in the `PORT` command was a local address. Therefore, all connection attempts from the server failed to reach the client. NetCheck correctly identified that the problem is that the client was behind a NAT. The diagnoses engine in NetCheck did this by detecting a difference between the IP of the client's original connection to the server and the IP specified in the `PORT` command.

Pidgin is a commonly used chat client application. Pidgin clients communicate with each other via an XMPP server. During a group meeting, one of the users (user A) was repeatedly dropped from the group conversation; another user (user B) could not log in with the Pidgin client. Traces were gathered from all the Pidgin clients and the XMPP server (4 hosts in total), and NetCheck was used to diagnose the network issues.

Application: Issue	NetCheck Diagnoses	Trace Size
FTP: Could issue only some of the commands.	• Client is behind NAT. • 42% data loss.	Client: 245 KB Server: 497 KB
Pidgin: Loss of connection; file transfer and login failure.	• The IP address that getsockname returns is different from the one the socket is bound to. • Message being received that has not been sent.	Client1: 49 MB Client2: 67 MB Client3: 81 MB Server: 93 MB
Skype: Poor call quality and messages lost.	• Data loss due to delay. • A different thread attempts to close sockets. • Client is behind NAT.	Client1: 831 MB Client2: 1.6 GB
VirtualBox: Silent drop of large UDP datagrams.	• Virtualization misbehavior. • Guest OS: MTU. • Host OS: UDP buffer size mismatch.	2.5 MB

Table 5: NetCheck diagnosis of faults in popular applications.

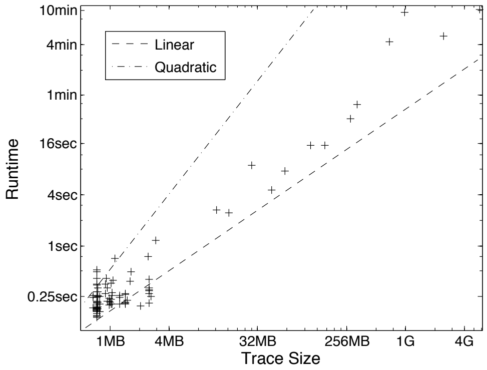


Figure 9: Runtime performance overhead of NetCheck. Data includes all traces in Sections 6.1–6.3.

NetCheck detected two important issues:

(1) Invocations of the `getsockname` syscall by user A's Pidgin client repeatedly returned an IP address that was different from the address that the socket was bound to. User A was multi-homed and was connected to both an ethernet and a wireless network. The wireless connection was poor, causing the default IP address of user A to change as she disconnected and reconnected to the wireless network. In this case, the network model simulated a `connect` syscall call on a socket bound to one IP to an endpoint that expected a different IP address. The `connect` generated a permanent rejection (line 21 in Algorithm 2). NetCheck therefore diagnosed the problem as a mismatch between the intended IP address and the IP address actually used.

(2) User B's Pidgin client could send data to the XMPP server, but all of the server's responses were lost. User B was behind a firewall that filtered packets with high-numbered source port values, including port 5222 on the XMPP server. By examining the model state towards the end of the trace, specifically considering the pattern of packet loss, diagnoses engine observed the selective filtering and diagnosed the problem as a middlebox issue.

Skype is a widely used VoIP service and instant messaging client. During a Skype session we noticed poor call quality and that messages were frequently dropped. Traces from two instances of Skype were gathered and evaluated with NetCheck. NetCheck detected that both clients were behind a NAT and that network delay caused severe data loss. NetCheck also revealed that when call quality degrades, Skype attempts to close its sockets from a separate thread. However, this does not terminate the blocked socket operation, but instead hangs the thread that is blocked on this operation. This is a known issue for some operating systems — a `close` call on a blocking socket from a different thread will keep the socket blocking indefinitely on `recv` or `recvfrom` [10, 45] (also `block_tcp/udp_close_socket` in Table 3). NetCheck diagnoses this issue correctly. The diagnoses engine also outputs a potential solution for Skype developers as part of the diagnoses: invoking `shutdown` on the blocking socket immediately unblocks `recv/recvfrom`.

VirtualBox is a popular tool for running virtualized operating system instances. We found and diagnosed a new bug in VirtualBox using Netcheck – applications running in a Linux VirtualBox instance on a Windows host OS would discard UDP datagrams of size over 8 KB when sent over an interface with VirtualBox's NAT virtual adapter [49]. When run on application traces gathered from the Linux instance (the *guest OS*), NetCheck correctly diagnoses the UDP datagram loss as a MTU issue. This is because from the standpoint of the guest VM, UDP datagrams over a certain size are discarded.

However, the root cause for this bug is as follows. The default receive buffer size on Windows is 8 KB. When the receive buffer is not full, Windows sockets can hold at least one more datagram even if the total datagram size exceeds the buffer size. When VirtualBox queries the socket for the amount of received data, Windows returns either the total size of datagrams in the buffer, or the buffer size, whichever is smaller. When a datagram larger than 8 KB is placed in the receive buffer, VirtualBox believes that the available datagram is only 8 KB and allocates an 8 KB application buffer. VirtualBox then silently drops the large datagram. To understand the usefulness of NetCheck for diagnosing this bug in VirtualBox, we collected traces of syscalls made by VirtualBox to the Windows *host OS* (`udp_set_buf_size_vm` in Table 3) to reproduce this issue. The network model of NetCheck correctly indicates the issue as being a UDP buffer size mismatch interfering with datagram delivery. Therefore, on the host OS, NetCheck also produces an accurate diagnosis of the root cause.

Our evaluation shows that NetCheck is effective at diagnosing faults in large applications in practical use.

6.4 Performance

Dynamically recording syscall invocations of complex applications can produce huge traces. Therefore, efficiency is a key challenge to designing a diagnosis tool that relies on logged syscall information. Figure 9 shows NetCheck's running time for traces of varying lengths. The figure plots the data for all the traces mentioned in this paper. Note that both the x and the y axes of Figure 9 have a logarithmic scale (i.e., a quadratic function is a straight line). The figure illustrates that NetCheck's performance across the various input traces lies between a quadratic and a linear function. NetCheck completes in less than 1 second on most traces, and even on 1 GB long traces, NetCheck completes in less that two minutes[4]. These measurements demonstrate that NetCheck is **efficient** for practical use.

Algorithm complexity. We now consider the complexity of the trace-ordering algorithm (Algorithm 2) — the key algorithm in NetCheck. Let n be the number of hosts, and l be the sum of the lengths of all input host traces. The inner while loop in Algorithm 2 must accept one syscall (if it rejects all syscalls, then the algorithm terminates). In the best case, this loop accepts a syscall on the first iteration, and in the worst case it must run n times to accept a syscall. The outer while loop iterates until all syscalls are accepted, or a total of l times. Therefore, Algorithm 2 makes l simulation calls in the best case, and $n * l$ simulation calls in the worst case. Supposing that the model simulates a syscall in constant time, the worst-case running time of Algorithm 2 is $\mathrm{O}(nl)$ and its best-case running time is $\mathrm{O}(l)$. Typically, the number of logged syscalls is much larger than the number of hosts, so the runtime will trend towards $\mathrm{O}(l)$.

Tracing overhead. To evaluate the overhead of `strace`, we micro-benchmarked the unit tests in Section 6.1, both with and without `strace`, 1K times. The overhead of `strace` across these runs, measured in elapsed time, had a median of 41 ms (0.79%), which is negligible. The standard deviation was 60 sec, due to the varying I/O behavior of the programs. Furthermore, in our experience, the overhead of `strace` on larger applications, such as Pidgin and Skype, was not perceptible.

6.5 Ordering Heuristic Efficacy

Algorithm 2 in Section 4.1 is a heuristic. One potential problem is if this algorithm terminates early: line 11 terminates the trace-ordering algorithm when no calls in `topCalls` can be accepted. Such a termination can occur before NetCheck detects an issue in the application traces. To evaluate the frequency of this early termination one of the authors manually inspected all of the traces collected in Sections 6.1 and 6.2, i.e., bug trackers evaluation (71 traces) and the testbed evaluation (20 traces). On just 2 of these traces (of 91), or 2.2%, did NetCheck not find any bugs and terminated without fully processing all the syscalls. This indicates that the ordering heuristic in NetCheck is effective at reconstructing plausible orderings that can then be used for diagnosis.

7 Limitations

IPC blindspots. NetCheck cannot detect faults that do not impact an application's syscall trace. This limitations impacts two situations. First, if an application uses non-socket IPC mechanism, then NetCheck will not see the resulting network traffic. For example, the `gethostbyaddr` error in Section 6.2 is due to an issue in how DNS requests are handled. Since DNS requests are handled in part by a non-native program `avahi`, the application's `strace` information does not include the relevant calls. However, NetCheck can be extended to parse `strace` call data and arguments to handle application-specific situations, for instance, to better understand DNS resolution errors reported by `avahi`.

Network blindspots. NetCheck observation of and reasoning about the network is limited to what is captured in system call traces. For example, NetCheck does not know the state of the OS network buffers, the network topology, etc. However, NetCheck's reliance on traces allows it to process previously generated traces, which is useful for reproducing and diagnosing bug reports.

Dynamic analysis. By relying on observed behavior, NetCheck can be considered as a dynamic analysis technique. As such, it cannot diagnose latent application behaviors that are possible, but have not been observed. However, dynamic analysis allows NetCheck to diagnose application issues that arise in deployment, such as those due to in-network state.

Improving the diagnoses engine. The diagnoses engine in NetCheck may be further improved with machine learning [3, 52]. For example, a supervised machine learning approach can be used to derive a signature from application traces or network packet traces, which can then be labeled according to previously observed patterns of correct and incorrect behavior [13].

Network model completeness. The network model in NetCheck simulates the behavior of network syscalls in an idealized network. To correctly perform this simulation, the model must be faithful and complete. Currently, the network model implements all syscalls and optional flags observed in traces of popular applications (Section 6). We continue to refine and improve this model as we encounter important new behaviors.

Multithreading. Currently, NetCheck cannot model systems with multiple threads that access shared resources (e.g., use the same socket descriptor). Improving multithreading support is part of our future work.

[4] This suggests that NetCheck can derive multiple plausible orderings without a significant performance penalty and use these to diagnose the issue. However, this would make NetCheck unnecessarily more complex and it already achieves high accuracy (Section 6.1).

8 Related Work

Blackbox diagnosis. Aguilera et al. introduced an important blackbox approach to debugging distributed systems [4]. In this approach, observations of a distributed executions are used to infer causality, dependencies, and other characteristics of the system. This approach was relaxed in later work to produce more informative and application-specific results in Magpie [5] and X-Trace [19]. This prior work focuses on tracking request flows through a distributed system. BorderPatrol [27] is another approach that traces requests among binary modules. In contrast, NetCheck is a blackbox approach to diagnosing network issues in applications.

Khadke et al. [24] introduced a performance debugging approach that relies on system call tracing. Unlike this prior work, NetCheck does not assume synchronized clocks and reconstructs a plausible global ordering.

Ordering events in a distributed setting. The happens-before relation logically orders events in a distributed system [28]. This relation can be realized with vector time, which produces a partial ordering of events in the system [18, 31]. Vector time requires non-trivial instrumentation of the application. NetCheck reconstructs a plausible order of the captured syscalls through heuristics, without modifying the application.

Globally synchronized clocks can order events across hosts. However, over 90% of syscall invocations we observed completed in less than 0.1 ms. Achieving synchronization at a granularity that is sufficient to order syscalls at hosts on a LAN is expensive and difficult.

Log mining. Prior work that uses dynamically captured logs of a program's execution is extensive and includes work on detecting anomalies [12, 22, 52, 30], linking logs and source code [55], identifying performance bugs [40, 44], and generating models to support system understanding [7, 6]. In contrast to this work, NetCheck's focus is on diagnosing network issues from logs of syscalls, though prior work on log mining can be used to expand the scope of NetCheck.

Debugging distributed systems. Techniques for debugging distributed systems are relevant to NetCheck's context of diagnosing network issues in applications. Many tools exist for run-time checking of distributed systems. These tools monitor a system's execution and check for specific property violations [37, 20, 29, 54, 14]. NetCheck is a more light-weight approach to diagnose issue observed through the syscall interface. This makes NetCheck broadly applicable, but it also limits the kinds of issues that NetCheck can uncover (see Section 7).

Specification and runtime verification. Substantial work has been done in validating API and protocol behaviors, e.g., finding faults in Linux TCP implementation [32], SSH2 and RCP [46], BGP configuration [17], and identifying network vulnerabilities [38]. Rigorously specifying protocols and APIs for testing and trace validation has also been described in [9]. These techniques are effective at finding bugs in an API or a protocol, but are not effective when the environment and networking semantic are also contributing factors. NetCheck can diagnose issues even if the input traces are valid API actions. Further, the simplicity of the NetCheck approach is one of its key advantages over prior work.

Application-specific fault detection. Pip [37] and Coctail [53] are distributed frameworks that enable developers to construct application-specific models, which have proven effective at finding detailed application flaws. However, to utilize these methods, a knowledge of the nature of the failures needs to be acquired, and the specific system properties must be specified. NetCheck diagnoses application failures without application-specific models. Khanna [25] identifies the source of failures using a rule base of allowed state transition paths. However, it requires specialized human-generated rules for each application.

9 Conclusion

This work proposes NetCheck, a tool for fault detection and diagnosis in networked applications. NetCheck is a blackbox technique that performs its diagnosis on an input set of traces of syscall invocations from multiple application hosts. NetCheck derives a plausible global ordering as a proxy for the ground truth, and uses a model of expected and simple network behavior to identify and diagnose unexpected behavior.

Our evaluation demonstrates that NetCheck is accurate and efficient. It correctly diagnosed over 95% of faults from traces that reproduce faults reported on bug trackers of 30 popular open-source projects. When applied to injected faults in a testbed, NetCheck identified the main cause in 90% of the cases. Furthermore, we used NetCheck to diagnose issues in large applications, such as Skype and VirtualBox, and in VirtualBox NetCheck found a new bug. We proved that NetCheck's algorithm derives a plausible global ordering in best-case linear running time and that it is efficient in practice.

Our experience with NetCheck demonstrates that it is possible to have an application-agnostic tool that provides practical and accurate fault diagnosis. NetCheck is freely available for download [33].

Acknowledgements

We thank our shepherd Dejan Kostic, Ulrike Stege for discussing ordering algorithms with us, and our reviewers for their invaluable feedback. This work was supported in part by the National Science Foundation through Awards 1223588 and 1205415, NSF Graduate Research Fellowship Award 1104522, the NYU WIRELESS research center and the Center for Advanced Technology in Telecommunications (CATT).

References

[1] accept. Accessed 2/27/2014, http://pubs.opengroup.org/onlinepubs/009695399/functions/accept.html.

[2] add SOCK_NONBLOCK and SOCK_CLOEXEC to socket module. Accessed 2/27/2014, http://bugs.python.org/issue7523.

[3] Aggarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V. N., and Voelker, G. M. Netprints: diagnosing home network misconfigurations using shared knowledge. In *NSDI* (2009).

[4] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[5] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004).

[6] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring Models of Networked Systems from Logs of their Behavior with CSight. In *ICSE* (2014).

[7] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE* (2011).

[8] Biggest mistake for IPv6: It's not backwards compatible, developers admit. Accessed 2/27/2014, http://www.networkworld.com/news/2009/032509-ipv6-mistake.html.

[9] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *SIGCOMM* (2005).

[10] Cannot interrupt blocking I/O calls with close(). Accessed 2/27/2014, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15430.

[11] Chen, K.-T., Huang, C.-Y., Huang, P., and Lei, C.-L. Quantifying skype user satisfaction. In *ACM SIGCOMM Computer Communication Review* (2006), vol. 36, ACM, pp. 399–410.

[12] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic internet services. In *DSN* (2002).

[13] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI* (2004).

[14] Dao, D., Albrecht, J., Killian, C., and Vahdat, A. Live debugging of distributed systems. In *Compiler Construction* (2009), Springer, pp. 94–108.

[15] Fallacies of distributed computing. Accessed 2/27/2014, http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.

[16] Deutsch's fallacies, 10 years after. Accessed 2/27/2014, http://java.sys-con.com/node/38665.

[17] Feamster, N., and Balakrishnan, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[18] Fidge, C. J. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference* (1988).

[19] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-Trace: A pervasive network tracing framework. In *NSDI* (2007).

[20] Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[21] net.ipv6.bindv6only=1 breaks some buggy programs. Accessed 2/27/2014, http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=560238#54.

[22] Jiang, G., Chen, H., Ungureanu, C., and Yoshihira, K. Multi-resolution abnormal trace detection using varied-length N-grams and automata. In *International Conference on Automatic Computing* (2005).

[23] Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., and Bahl, P. Detailed diagnosis in enterprise networks. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 243–254.

[24] Khadke, N., Kasick, M. P., Kavulya, S. P., Tan, J., and Narasimhan, P. Transparent system call based performance debugging for cloud computing. In *Workshop on Managing Systems Automatically and Dynamically (MAD)* (2012).

[25] Khanna, G., Cheng, M., Varadharajan, P., Bagchi, S., Correia, M., and Veríssimo, P. Automated rule-based diagnosis through a distributed monitor system. *IEEE Transactions on Dependable and Secure Computing 4*, 4 (2007).

[26] Knowles, S. IESG advice from experience with path MTU discovery. RFC, Internet Engineering Task Force, 1993.

[27] Koskinen, E., and Jannotti, J. Borderpatrol: isolating events for black-box tracing. In *Eurosys* (2008).

[28] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[29] Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. D3S: Debugging deployed distributed systems. In *NSDI* (2008).

[30] Lou, J.-G., Fu, Q., Yang, S., Xu, Y., and Li, J. Mining invariants from console logs for system problem detection. *ATC* (2010).

[31] Mattern, F. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms 1*, 23 (1989), 215–226.

[32] Musuvathi, M., and Engler, D. R. Model checking large network protocol implementations. In *NSDI* (2004).

[33] NetCheck. Accessed 2/27/2014, https://netcheck.poly.edu/.

[34] On Mac / BSD sockets returned by accept inherit the parent's FD flags. Accessed 2/27/2014, http://bugs.python.org/issue7995.

[35] Posix-omni-parser. Accessed 2/27/2014, https://github.com/ssavvides/posix-omni-parser.

[36] Prevent socket hijacking on OSes that don't prevent it by default (Windows). Accessed 2/27/2014, https://tahoe-lafs.org/trac/tahoe-lafs/ticket/870.

[37] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[38] Ritchey, R., and Ammann, P. Using model checking to analyze network vulnerabilities. In *Security and Privacy* (2000).

[39] Rotem-Gal-Oz, A. Fallacies of distributed computing explained. *Accessed 2/27/2014, URL http://www. rgoarchitects. com/Files/fallacies. pdf* (2006).

[40] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011).

[41] Security: SO_EXCLUSIVEADDRUSE should be enabled when binding to ports on Windows. Accessed 2/27/2014, http://twistedmatrix.com/trac/ticket/4195.

[42] SO_REUSEADDR broken on Windows. Accessed 2/27/2014, http://bugs.sun.com/bugdatabase/viewc_bug.do?bug_id=4476378.

[43] SO_REUSEADDR doesn't have the same semantics on Windows as on Unix. Accessed 2/27/2014, http://bugs.python.org/issue2550.

[44] SUBHLOK, J., AND XU, Q. Automatic construction of coordinated performance skeletons. In *IPDPS* (2008).

[45] TCPSocket readline doesn't raise if the socket is close'd in another thread. Accessed 2/27/2014, http://bugs.ruby-lang.org/issues/4390.

[46] UDREA, O., LUMEZANU, C., AND FOSTER, J. Rule-based static analysis of network protocol implementations. *Information and Computation 206*, 2 (2008), 130–157.

[47] Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE. Accessed 2/27/2014, http://msdn.microsoft.com/en-us/library/ms740621%28VS.85%29.aspx.

[48] webrick: Ruby Standard Library Documentation. Accessed 2/27/2014, http://www.ruby-doc.org/stdlib-1.9.3/libdoc/webrick/rdoc/index.html.

[49] When using NAT interface on Windows host, guest can't receive UDP datagrams larger than 8 KB. Accessed 2/27/2014, https://www.virtualbox.org/ticket/12136.

[50] Windows ntpd should secure UDP 123 with SO_EXCLUSIVEADDRUSE. Accessed 2/27/2014, https://support.ntp.org/bugs/show_bug.cgi?id=1149.

[51] Wireless Implementation Testbed Laboratory (WITest) at NYU-Poly. Accessed 2/27/2014, http://witestlab.poly.edu.

[52] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).

[53] XUE, H., DAUTENHAHN, N., AND KING, S. Using replicated execution for a more secure and reliable web browser. In *NDSS* (2012).

[54] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI* (2009).

[55] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[56] ZHUANG, Y., BESCHASTNIKH, I., AND CAPPOS, J. NetCheck Test Cases: Input Traces and NetCheck Output. Tech. Rep. TR–CSE–2013–03, NYU Poly, 2013.