Restoring End-to-End Resilience in the Presence of Middleboxes

Eric J. Brown,[†] Mark K. Gardner,[†] Umar Kalim,[*†] and Wu-chun Feng[*]
Office of IT[†], Department of Computer Science[*]
Virginia Tech, Blacksburg, VA 24060
{brownej, mkg, umar, wfeng}@vt.edu

***Abstract*—The philosophy upon which the Internet was built places the intelligence close to the edge. As the Internet has matured, intermediate devices or *middleboxes*, such as firewalls or application gateways, have been introduced, thereby weakening the end-to-end nature of the network. As a result, applications must often modify their behavior to accommodate the middleboxes. This is is especially true in the case of transient failure of stateful devices.**

The failure of a middlebox causes it to lose the state it maintained, causing the failure of the associated TCP connections. Rather than assign the responsibility for recovery to applications, we incorporate a mechanism called an *isolation boundary* into TCP itself to increase resilience. The isolation boundary maintains a small amount of state across TCP connections, thus enabling reconnection. Furthermore, it does so without breaking backward compatibility with existing TCP.

We present an implementation of the isolation boundary in the FreeBSD kernel and demonstrate its backward compatibility with TCP. We quantify the performance impact of the proposed mechanism on the establishment of new and resumed connections for both legacy and extended TCP stacks.

***Index Terms*—TCP/IP, middleboxes, fault tolerance, resilience.**

I. Introduction

A guiding principle in the design of the Internet has been that network communication is end-to-end and that network intelligence should be as close to the resources on the edge as possible [1]. Because of this, the capacity of the Internet has scaled well with the rapid growth in the number of devices. As the Internet has grown and matured, however, it has been necessary to introduce intelligent intermediate devices, such as firewalls or application gateways, hence weakening the end-to-end nature of the network. Although these intermediate devices weaken the notion of end-to-end connections, they are necessary for operational or functional reasons. Nevertheless, intelligent intermediate devices reduce the network transparency for end hosts and their applications, requiring the end hosts to make decisions that accommodate the intermediate devices.

The proliferation of intelligent devices in the network has increased the likelihood that communication will be interrupted. When a middlebox fails, it loses the state it maintained on behalf of ongoing communication. Even if the fault is transient, the applications on the end host have no recourse but to try to re-establish communication or to pass the problem on to the user. Neither approach is entirely satisfactory. In the former case, each application needs to be written to handle middlebox failure. While burdensome, it is preferable to the latter case where responsibility for handling the failure is passed to the user who is not even aware of the existence of middleboxes. However, there is another choice. Rather than require applications or users to handle the failures, we take the approach of incorporating a mechanism for increasing resilience into TCP itself.

Specifically, we introduce a mechanism, which we call an *isolation boundary*, that places a TCP connection in the context of a transport independent flow (TI-flow). The isolation boundary keeps track of where TCP is in the context of the TI-flow so a new TCP connection can be created and communication restored in the event of a transient failure without the application — or more importantly, the user — becoming aware. The isolation boundary separates flow identification from transport addressing. This facilitates extending re-connection to include migration of the flow from one subnet to another. Most important of all, *our approach maintains backward compatibility with existing devices, thus allowing incremental adoption.*

In Section II, we introduce the conceptual basis upon which the isolation boundary rests. Section III discusses the proposed extensions to TCP that provides the isolation boundary mechanism, and Section IV touches upon our implementation and evaluates its compatibility and performance. Section V places our work in the broader context, and Section VI concludes with a discussion of future work.

II. Conceptual Design

A. Logical Construct

It is well established that middleboxes today are an "Internet fact of life" [2], nevertheless it is also accepted that they break the end-to-end semantics assumed by typical applications. This is because the middleboxes maintain state and interact in the conversation, often transparently to the end systems. While we acknowledge that middleboxes provide some benefits, the liability is that when they fail, connections being carried through them are also broken. In addition, a change in network address will also break a connection as the transport protocols rely on these addresses to identify the connection.

Here we propose to address such challenges by developing the notion of an abstract flow, specifically a *transport-independent flow* (TI flow) that represents the abstract com-

munication between applications, independent of the underlying transport protocols. For this purpose, we define the notion of an *isolation boundary*, coupled with TCP, which allows us to maintain end-to-end semantics at an abstract level, without precluding middleboxes. This is possible by introducing two mechanisms: (1) a *transport-independent flow identifier* (TIFID) to uniquely define a flow abstraction and (2) a mechanism to describe its own sequence space in order to maintain the TCP semantics of reliable, in-order delivery.

In order to be backward compatible, the TI-flow capability must be negotiated out-of-band from TCP's data stream. To have the least performance impact, maintenance of the TI-flow capability should also happen out-of-band from TCP's data stream. We employ TCP options to establish and maintain the placement of a TCP connection within the context of the TI flow. To avoid duplicating the mechanics of TCP, we propose that the isolation boundary leverages support from TCP by delegating the tasks of reliable, in-order delivery and the description of the sequence space. However, to implicitly maintain the end-to-end semantics[1] for the flow abstraction, we define an abstract sequence space. The data in each TCP connection is then mapped into this sequence space as a part of placing the TCP connection in the context of the transport-independent flow. All that is required is to describe how each TCP connection fits into the overall context of the TI flow. It is sufficient to establish this mapping during TCP's synchronization phase. Once the mapping is established, progress through TCP's sequence space implies progress through the transport-independent sequence space. Note that the mechanism of defining the TI flow does *not* imply maintaining distributed state. Since we delegate the task of maintaining distributed state to TCP and only synchronize the abstraction with TCP's semantics at the time of setup, the expected overhead will be negligible.

Leveraging TCP options and keeping pace with TCP's sequence space required us to implement the isolation boundary in the kernel, e.g., FreeBSD. With a user-library implementation, we would have needed to develop a mechanism to probe whether the communicating peer had support for such flow abstractions. Any probe protocol would ultimately have to resort to timeouts to infer the lack of support. Unlike the user-library implementation, the presence of custom options in the SYN+ACK message would indicate support, while absence would indicate lack of support, thus eliminating the need for any inferential rules. The discovery of end-to-end support can happen alongside connection setup, thus allowing the kernel implementation to avoid probing and be backward compatible at the same time. Since the isolation boundary plays its role only when a new transport flow is setup and does not interfere with the critical data path, the expected overhead is negligible.

We argue that the isolation boundary does not interfere with the critical path as most of the end-to-end semantics for the flow abstraction are delegated to TCP. As such, the isolation boundary's involvement is only with placing the TCP connection into the context of the TI flow. (Note that with a user-library implementation, we would not be able to delegate the task of maintaining end-to-end semantics to TCP; instead, since these mechanisms would be hidden from the users' context, the same would have to be implemented — as duplicate effort — by the library.)

[1]We assume that a conversation between endpoints may last longer than the life of the TCP flow.

B. Practical Details

With the logical construct defined above, we now study the details of creating an implementation. A critical aspect that must be considered for a practical implementation is the amount of data required to convey the context of the TI flow in the TCP option field. Another important aspect is the question of security. The introduction of the flow option should not compromise TCP's security characteristics.

To realize the flow identifier and the transport-independent sequencing discussed above, we propose employing TCP options. TCP options may be used during the TCP SYN phase to reliably exchange these parameters — a unique flow identifier and the mapping between the transport-independent sequence space (e.g., TIFID) and TCP's sequence space. We discuss the protocol for selecting the flow identifier and populating the transport-independent sequence numbers in Section III.

Before we further discuss the practical details, we need to acknowledge the constraints of using custom TCP options to exchange the transport-independent flow identifier and the transport-independent sequence numbers, e.g., space availability in the TCP header. When the TCP SYN flag is set the following options need to be supported: maximum segment size (RFC793, four octets), window scaling (RFC1323, three octets), selective acknowledgement permitted (RFC2018, two octets), and time stamp (RFC1323, ten octets). This leaves us with 21 octets, although most implementations will only leave 20 octets due to field alignment. Because of this limited TCP option space, not all options can be supported simultaneously. (We discuss the details of these unsupported options in our preliminary work [3].)

The size of the transport-independent sequence space should be at least as big as TCP's sequence space (32 bits). Any smaller would create problems in the mapping between the two spaces during TCP's synchronization phase. Having more space allows for the potential to address the issues of TCP sequence space wrap-around over high-speed links as a future concern. Larger spaces for both the transport-independent sequence space and the TIFID will decrease the vulnerability of the TI flow to session hijacking, which is also covered in more detail in our preliminary work [3]. Because the upper bound on sizes is dictated by the remaining space for TCP options, we chose the upper bound for each field, i.e., TIFID, sequence number, and acknowledgement number, to be 48 bits each for a total of 18 bytes.

III. Extending TCP

The TCP header, shown in Figure 1, consists of 20 octets for fields that must be present in all TCP headers, followed

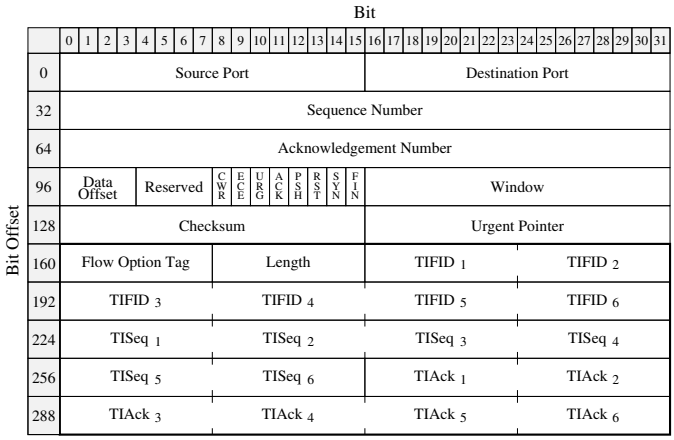

Fig. 1. The Proposed Transport-Independent Flow Option.

by up to 40 octets of options. We extend TCP by creating a *transport-independent flow option*, which is only valid during connection setup. The option consists of three 48-bit fields, as shown, in addition to the option tag and the length.

The first field contains the *transport-independent flow identifier (TIFID).*, which provides the layer of indirection needed for the isolation boundary by labeling a flow independent of the underlying transport. The TIFID is an opaque identifier that is unique within the context of the two end hosts. A straightforward way to specify a TIFID with the correct properties is for the requesting process to specify a locally unique value for the first half of the TIFID in the initial SYN packet (TIFID_1 through TIFID_3) and the responding process to specify a locally unique value for the second half in the SYN-ACK packet (TIFID_4 through TIFID_6). As with TCP initial sequence numbers, both halves of the TIFID should be selected at random to guard against connection hijacking attacks. Finally, the second half of the TIFID is zero during the time that the TIFID is partially specified, i.e., in the SYN packet.

By itself, the TIFID is insufficient to allow resynchronization when the underlying transport fails. The missing information is the position within transport-independent flow. Thus, there are two additional fields in the flow option indicating the next byte to be sent, i.e., the *transport-independent sequence number (TISeq)*, and the last byte received, i.e., the *transport-independent acknowledgement number (TIAck)*. As with traditional TCP, the two end points select an initial TISeq during the three-way handshake and each returns a TIAck to acknowledge the receipt of a SYN packet. Unlike TCP, the SYN bit does not need to be acknowledged because it is TCP's responsibility. When not defined, such as during the first phase of the three-way handshake, a TISeq is zero.

The TISeq are mapped onto the protocol-dependent sequence numbers of the underlying (TCP) transport and remain synchronized with them as long as the transport connection is active. The TISeq progresses through its space in a manner that is consistent with the transport sequence number being incremented. Because of the implicit synchronization, there is no need to explicitly send the TISeq and TIAck numbers after the three-way handshake.

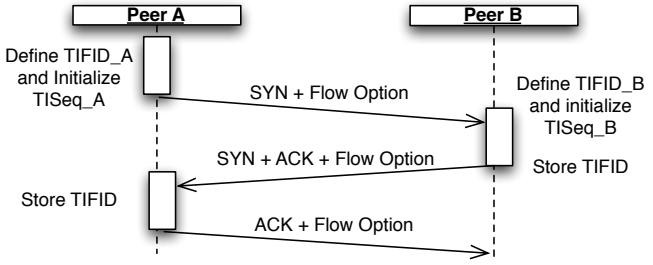

Fig. 2. A Sequence Diagram During Connection Setup.

A. Connection Establishment

Figure 2 shows a sequence diagram of connection establishment. The initiator of communication, Peer_A, defines the first half of the flow identifier, TIFID_A, and initializes the second half to zero. Peer_A also selects a random initial TISeq number, TISeq_A, and establishes a mapping between TISeq_A and the initial TCP sequence number. (TIAck_A is initialized to zero.) It then sends a SYN packet with an flow option containing these values. Peer_B defines the second half of the TIFID, TIFID_B, using a random value and selects a random initial TISeq number, TISeq_B. It acknowledges receipt of the SYN packet by setting $\text{TIAck}_B = \text{TISeq}_A$. It then sends a SYN+ACK packet with a flow option containing these values. Upon receipt of the reply, Peer_A notes the value of the completed TIFID, which uniquely identifies the flow. It then returns an ACK packet containing the completed TIFID, its new TISeq_A, and a $\text{TIAck}_A = \text{TISeq}_B$ acknowledging the SYN+ACK packet as the final phase of the three-way handshake. Finally, Peer_B validates that its SYN packet was received by checking TIAck_A. At this point, transport-independent flows in each direction have been established, along with the associated bidirectional TCP connections.

B. Connection Re-establishment

The failure of a middlebox in the network causes the logical end-to-end connections passing through it to also fail. Even though the connection failed, the isolation boundary maintains the position within the application data streams in each direction, via TISeq and TIAck, so that the transport can resume in the correct place once a new TCP connection is established.

The procedure for resuming operation after disconnection is the same as for creating a new connection except the previously completed TIFID, signified by a second half not equal to zero, is used instead. Since the TIFID is already complete, the receiving stack looks up the isolation boundary information corresponding to the complete TIFID and creates a new TCP connection upon which to resume the sending of application data. The exchange of SYN and SYN+ACK packets in this case allows the stacks to re-synchronize where

they left off at the time of the disconnection by exchanging the TISeq and TIAck numbers. The peers also use the TISeq and TIAck numbers to establish new mappings from the old transport-*independent* sequence space to the new transport-*dependent* sequence space.

C. Backward Compatibility

End hosts advertise that they implement the isolation boundary by specifying the flow option in a TCP header. If both hosts specify the flow option, then the functionality of an isolation boundary is enabled. If either host is unable to support the isolation boundary for any reason, they will not supply the flow option during connection establishment, and hence, both will continue to connect without the isolation boundary, thus maintaining backward compatibility.

Even when some overzealous middleboxes strip off unknown TCP options, compatibility is still maintained because hosts that do not implement the isolation boundary will behave the same as before while hosts that do implement the isolation boundary will be led to believe that the other host does not, and hence, fall back to legacy behavior.

In this way, backward compatibility is maintained, and there is no requirement that all hosts be updated simultaneously.

IV. Implementation and Evaluation

We implemented the isolation boundary in the FreeBSD 8.1 kernel. A summary of the changes follows: 1,156 lines added in 55 locations, 58 lines deleted in 34 locations, and 435 lines modified in 42 locations. A total of 1,649 out of 237,410 (0.7%) lines were touched in the network stack, representing 131 locations in 12 out of 122 files (9.8%). We defer discussing the details due to lack of space but invite the interested reader to peruse the source code, which will be made available under the BSD license, and submitted for inclusion in the kernel.

We now turn our attention to evaluating the backward compatibility, correctness, and performance of the implementation.

A. Backward Compatibility and Correctness

There are two cases to consider in ensuring backward compatibility with legacy TCP: a modified sender connecting to an unmodified receiver and an unmodified sender connecting with a modified receiver. We use SSH as an example and show traces of connection establishment using a `tcpdump`, which has been modified to display the new TCP option.

Trace 1 shows the three-way handshake during connection establishment between a sender that wants to establish an isolation boundary and a receiver that does not. The SYN packet uses the flow option to convey a partial TIFID and an initial TISeq. The TIAck is zero as there is nothing to acknowledge yet. The SYN+ACK packet does not contain a flow option, since the receiver does not implement the isolation boundary or is unable set one up at this time. As a result, the sender does not set the flow option in the ACK packet and both hosts communicate without an isolation boundary.

In the case of an unmodified sender talking to a modified receiver, the receiver is made aware that the sender does not implement (or has decided not to set up) the isolation boundary when it receives the SYN packet without the flow option being set. Therefore, it does not set the flow option in the SYN+ACK packet it sends, and the connection proceeds without the isolation boundary. We omit the trace for this case as it is exactly the same as legacy TCP.

Trace 1 Sender Implements the Isolation Layer

```
Packet 1: IP 192.168.1.2.4874 > 192.168.2.4.ssh:
          Flags[S],seq 100,win 65535,options[mss
          1460,nop,wscale 3,sackOK,TS val 787 ecr 0,
          flow-d tifid 4b2209000000 tiseq 00000000001f
          tiack 000000000000],len 0
Packet 2: IP 192.168.2.4.ssh > 192.168.1.2.4874:
          Flags[S.],seq 200,ack 101,win 65535,
          options[mss 1460, nop, wscale 3, sackOK,
          TS val 197 ecr 787],len 0
Packet 3: IP 192.168.1.2.4874 > 192.168.2.4.ssh:
          Flags[.],ack 1,win 8326, options[nop, nop,
          TS val 788 ecr 197],len 0
```

Trace 2 Both Implement the Isolation Layer

```
Packet 1: IP 192.168.1.2.11305 > 192.168.2.4.ssh:
          Flags[S],seq 110,win 65535,options[mss
          1460,nop,wscale 3,sackOK,TS val 109 ecr 0,
          flow-d tifid 0a4530000000 tiseq 00000000001d
          tiack 000000000000],len 0
Packet 2: IP 192.168.2.4.ssh > 192.168.1.2.11305:
          Flags[S.],seq 456,ack 111,win 65535,
          options[mss 1460,nop,wscale 3,sackOK,TS val
          138 ecr 109, flow-d tifid 0a4530be79bf tiseq
          00000000001f tiack 00000000001d],len 0
Packet 3: IP 192.168.1.2.11305 > 192.168.2.4.ssh:
          Flags[.],ack 1,win 8326,options[nop,nop,TS
          val 110 ecr 138,flow-d tifid 0a4530be79bf
          tiseq 00000000001d tiack 00000000001],len 0
```

Now that backward compatibility has been demonstrated, we show the case where both the sender and the receiver implement the isolation boundary. As Trace 2 shows, the sender sets the flow option in the SYN packet. The receiver replies with a SYN+ACK packet containing a flow option with a complete TIFID, its initial TISeq, and the appropriate TIAck. Upon receipt of the reply, the sender knows that the receiver wants to utilize an isolation boundary so it sends an ACK packet with the flow option filled out to confirm that it received the option correctly. This protects the isolation boundary capability in the presence of a middlebox that strips options in one direction and not the other. The connection proceeds utilizing the isolation boundary. The TCP flow option is no longer needed in any other packets of the connection.

B. Overhead Incurred by the Isolation Boundary

Because the flow option added to TCP to support the isolation boundary is only transmitted on the wire during connection setup, i.e., during the three-way handshake, we expect any additional overhead would be most observable as an increase in setup time during that phase. (During the operation of the connection, there is a small amount of processing needed to keep the TISeq and TIAck in synchronization with their TCP counterparts, but the effect is minimal as we will

show.)

1) Overhead during establishment:: The instructions added to the kernel increase the time it takes to establish a TCP connection. The amount of additional work that is done is small, so the effect should also be small.

There are two challenges in measuring the time it takes to establish a TCP connection so that a comparison can be made. First, precisely measuring the overhead requires kernel instrumentation, which is tedious to set up and has the potential to perturb the normal operation of the kernel, thereby obscuring the values being measured. Second, the overhead occurs on both the initiator of communication and the responder. Measuring elapsed time in a distributed control system, such as TCP, is further complicated by the fact that the clocks on the end hosts are in only loosely synchronized when compared with the magnitude of the values being measured.

The first concern is addressed by measuring times in user-space code under the assumption that (on average) both the extended and legacy TCP stacks should see the same perturbations from unrelated processing on the hosts. This assumption is supported by the low variance seen on repeated measurements.

The second concern is addressed by taking both time stamps on the same host. A simple client and server are used to create a connection. A time stamp is place in the client code just before the call to `connect` whereupon the client immediately blocks on `recv`. The server immediately closes the connection upon returning from `accept` causing the `recv` on the client to return without reading any data. The client then closes its socket. Except for the processing that occurs on the final FIN packet from the client to the server, acquiring time stamps in these locations brackets all the additional processing that is done on both the client and the server in the isolation boundary implementation. Subtracting the elapsed time for establishing a connection with extended TCP stack from the elapsed time for the legacy stack gives the overhead. The average overhead was computed over a large enough number of runs that the half width of the 95% confidence interval is below 5%.

The test environment consists of three Dell PE2650 servers running FreeBSD 8.1. The servers each have dual Intel Xeon SMT processors with a frequency of at least 2.0 GHz and hyperthreading turned on. They also have 4 GB of DDR2 RAM and a bus speed of 533 MHz and are connected by 1 Gbps Ethernet. The average throughput measured with `iperf` is 940 Mbps for the legacy TCP stack. Two of the servers are configured as a client and a server. The third is configured as a WAN emulator using Dummynet.

The average time between time stamps without the isolation boundary mechanism is 1.168 ± 0.054 msec while the average with the mechanism is 1.148 ± 0.045 msec. Based upon the overlapping of the confidence intervals, we conclude that the increase in overhead for connection establishment is negligible.

2) Overhead during data transfers: For each packet received, the TISeq and TIAck values must be updated to advance at the same rate as the corresponding TCP variable to which they are logically bound. This adds a small amount of code that is executed only when the isolation boundary is in use. We quantify the cost of the additional code by comparing the time it takes to transfer an amount of data using `iperf` both with and without the isolation boundary. Dummynet is configured to not delay packets in order to ensure that the network pipe remains as full as possible to make the effect more readily observable.

The time it takes to transfer 262.1 MB of data using `iperf` on a kernel without the isolation boundary is $2{,}230.0 \pm 3.3$ milliseconds (ms) while the time it takes using a kernel with an active isolation boundary is $2{,}229.3 \pm 0.5$ ms with both confidence intervals computed at 95% confidence. This substantiates the claim made earlier that very little processing is needed to keep the transport-independent sequence and acknowledgment numbers in synchronization with their TCP counterparts.

C. Time to Reconnect

Although the isolation boundary enables automatic reconnection when a middlebox loses state, reconnection must be fast enough to remain transparent to the user or application. As above, there is the issue of finding a common time base in a distributed system in which to estimate the reconnect time. Unlike measuring the connection overhead, we cannot use the client/server approach to bound the time since the application is unaware that its TCP connection has failed.[2] Instead, we use the `tcpdump` time stamps on the SYN and ACK packets of the three-way handshake in a trace taken on the originating host as our sources of time. The difference between the two time stamps is an estimate of the reconnection time. We also measure the setup time for a legacy connection as a baseline.

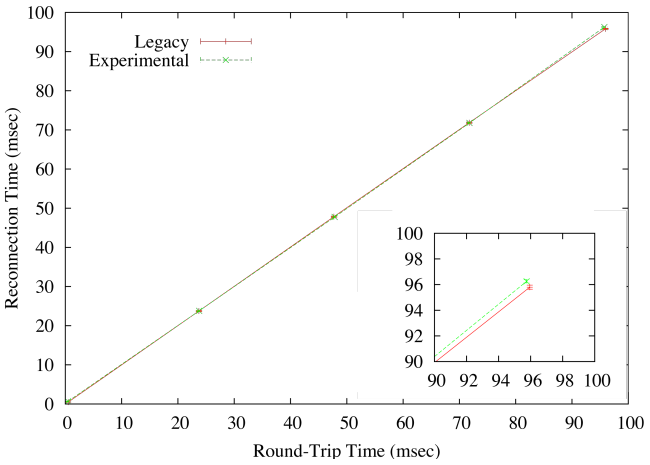

Fig. 3. Time for client to reconnect vs. round-trip time.

As expected, because reconnection causes a packet exchange, the time for a client to reconnect is expected to be a function of the round-trip time (RTT), as evidenced in

[2]We simulate failure by accessing a custom sysctl variable that calls a kernel function to disconnect the "failed" TCP connection and re-synchronize the TI-flow with a new TCP connection.

Figure 3. It takes about one RTT for the sender to get back to the state before disconnection and one and a half RTT for the receiver to do the same. Because we are measuring time from the perspective of the sender, the reconnect time should be less than the $1.5 * \mathrm{RTT}$ needed to establish a TCP connection on the receiver. Except for the highest RTT measured, the reconnection time of the extended TCP stack is indistinguishable from the connection setup time of the legacy stack. For an RTT of 95.75 ± 0.11 ms, as shown in the inset, the reconnection time is 96.26 ± 0.13 ms while the connect time is 95.80 ± 0.16 msec. The difference is significant at the 95% confidence level. For most applications, a difference of 0.46 ms should be acceptable. We have yet to optimize the code and expect that it can be further reduced.

V. Related Work

The realization that middleboxes are no longer harmful and are an "Internet fact of life" has been ever growing [2]. Though it is accepted that when deployed as active entities they typically violate end-to-end semantics, the features that they provide are worth the pain. It is for this reason that we see conscious efforts by the networking community to facilitate the deployment of middleboxes in a manner that retains their benefits while minimizing their drawbacks, e.g., [4], [5].

At present, two approaches exist to mitigate the challenges introduced by the middleboxes: (1) a method of explicit control of the middleboxes (e.g., middlebox communication (MIDCOM) [6]–[9], and IETF Next Steps in Signaling (NSIS) [10]); and (2) the conventional method of traversing the middleboxes (i.e., without any control relationship between the end host and the middlebox as is in the case of IETF Session Traversal Utilities for NAT (STUN) [11], Traversal Using Relays around NAT (TURN) [12], and Interactive Connectivity Establishment (ICE) [13]).

We do not argue that the methods developed to maintain end-to-end semantics without precluding middleboxes are better or lacking. Instead in this paper, we present the case of the Isolation Boundary, which enables the end hosts to establish an abstract concept of the application stream independent of the underlying transport. Such a mechanism allows us to accept interactions with middleboxes all the while strengthening the end-to-end nature of the communication.

Below we discuss select efforts and highlight their approach towards maintaining end-to-end semantics while accepting middleboxes.

Network researchers have been studying interaction of end-hosts with middleboxes [6]–[9]. Here, the authors argue that the middleboxes should be application agnostic (i.e., they should not be required to maintain application intelligence to assist to the fullest). For this reason, they propose an architecture and a framework to allow trusted entities — referred to as MIDCOM agents — to assist middleboxes in meeting their objectives without incorporating application intelligence in the middleboxes. The MIDCOM agents may reside on end-hosts, proxies or application gateways depending upon the circumstances. In contrast to MIDCOM the isolation boundary establishes a higher level concept — a transport independent flow — which allows us to maintain end-to-end semantics despite the presence of middleboxes.

Another noteworthy effort is that of Salz et al. [14] which feature support for end hosts interacting with middleboxes (in this case proxies). Here the authors argue that session layer services are necessary to meet functional requirements such as connection multiplexing, congestion state sharing, application-level routing, mobility management. They present the project TESLA which builds an abstraction of session layer services allowing the applications to interact with network flows instead of calling functions on the sockets API. Also, the TESLA library maps the incoming logical flow to one transport flows. This allows for seamless migration in the case of a disconnection. The implementation is done by writing event-handlers with a callback-oriented interface between handlers. This method also allows programmers to add functionality to the library. The solution is implemented as a shim layer (using dynamic library interposition) which runs in user space and traps network operations. Performance evaluation (in terms of achievable throughput) of the implementation shows that TESLA does incur some overhead, though negligible. However, in case of latency, TESLA performs at par with all application based implementations of session semantics. As with TESLA, by default, the isolation boundary supports resilient transport flows in the face of intermittent connectivity. A prominent difference though is that TESLA is implemented as a user space library while we implement the isolation boundary as part of the TCP/IP stack (kernel) implementation.

As also discussed in Section II opting for kernel implementation allows us to maintain backward compatibility — the isolation boundary avoids using any probes to detect if the peer also supports the same mechanisms. Also, by being part of the kernel, most of the end-to-end semantics can be conveniently delegated to TCP, which would have been hidden from a user-library implementation and therefore would have to be duplicated. Lastly, since the mechanisms proposed by the isolation boundary only interact with the connection setup phase and do not interfere with the critical/data path, the expected overhead is negligible, as we have also observed from empirical observations presented in Section IV.

An approach, a predecessor perhaps to the MIDCOM framework, is presented by Maltz et al. [15]. The authors develop the idea of mobility management with the aid of proxy that implements the required semantics. The idea is to split the communication stack between the mobile node and the proxy to facilitate mobility. This splitting of the stack enables features such as (1) enabling support for use of multiple interfaces; (2) enabling simultaneous use of multiple transport flows over different network interfaces; (3) providing the ability to define preferences for each network interface; (4) implementing presentation primitives, in other words assist middleboxes, by methods such as translation, encoding, compression and, encryption at the proxy; and (5) managing disconnections and migration of flows from one network interface to the other. To do so, they present the

project MSOCKS which features a custom protocol. This protocol allows exchange of control information between the mobile node and the proxy to realize the features listed above. The performance evaluation of the implementation (by the authors) as well as design highlights scalability concerns. Another concern, given the design, is with disconnections and migrations (i.e., how quickly does the implementation react to a disconnection and migrate its flows to alternate network interfaces). Though the work presented accepts middleboxes and makes a conscious attempt to assist them, the mechanics explicitly do away with the end-to-end semantics and therefore adopt an approach which is the antithesis of what the isolation boundary proposes.

VI. Summary and Future Work

In this paper, we have laid out arguments for establishing an isolation boundary in TCP that helps to restore end-to-end resilience in the presence of middleboxes while maintaining full backward compatibility with legacy TCP. Our realization of the isolation boundary introduces little overhead, and in most cases, it is difficult to measure the difference in performance.

One aspect of the isolation boundary which we have not focused on is the second variant of the TCP flow option outlined in our previous work [3]. Instead of providing a stream for application data, it provides a control channel that could be used to allow the end host to negotiate with middleboxes to obtain desired services. In all respects, other than the type of stream, it behaves exactly like the variant discussed in this work. To distinguish between the two, we call the first the *Flow-D* (or data variant) and the second the *Flow-C* (or control variant). A key property of Flow-C is the extensibility it enables in TCP while also maintaining backward compatibility. Because of lack of space, we leave the discussion of this option and its protocol for future work.

Besides releasing the source code under the BSD license for inclusion in the FreeBSD kernel, we are preparing a request for comments for submission to the IETF to stimulate further discussion within the networking community. Also in the spirit of the IETF, we seek collaborators interested in developing independent implementations and joining in interoperability testing.

Acknowledgements:

The authors would like to thank Colin Constable, Barnaby Crahan, Jayabharat Boddu, John Scudder, and Danny Jump from Juniper Networks for sponsoring the research and for providing feedback. They would also like to think Carl Harris and William Dougherty from Virginia Tech for their support and encouragement.

References

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," ACM Transactions on Computer Systems, vol. 2, pp. 277–288, 1984.

[2] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker, "Middleboxes No Longer Considered Harmful," in Proceedings of the 6th Symposium on Operating Systems Design and Implementation. USENIX Association, 2004, pp. 215–230.

[3] U. Kalim, E. Brown, M. Gardner, and W. Feng, "Enabling Renewed Innovation in TCP by Establishing an Isolation Boundary," in 8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT), 2010. [Online]. Available: http://pfld.net/2010/technical.php

[4] D. Joseph and I. Stoica, "Modeling Middleboxes," Network, IEEE, vol. 22, no. 5, pp. 20–25, 2008.

[5] D. A. Joseph, A. Tavakoli, and I. Stoica, "A Policy-Aware Switching Layer for Data Centers," in Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication, ser. SIGCOMM '08. ACM, 2008, pp. 51–62.

[6] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan, "Middlebox communication architecture and framework," RFC 3303 (Informational), Internet Engineering Task Force, Aug. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3303.txt

[7] R. P. Swale, P. A. Mart, P. Sijben, S. Brim, and M. Shore, "Middlebox Communications (midcom) Protocol Requirements," RFC 3304 (Informational), Internet Engineering Task Force, Aug. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3304.txt

[8] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues," RFC 3234 (Informational), Internet Engineering Task Force, Feb. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3234.txt

[9] M. Stiemerling, J. Quittek, and T. Taylor, "Middlebox Communications (MIDCOM) Protocol Semantics," RFC 3989 (Informational), Internet Engineering Task Force, Feb. 2005, obsoleted by RFC 5189. [Online]. Available: http://www.ietf.org/rfc/rfc3989.txt

[10] R. Hancock, G. Karagiannis, J. Loughney, and S. V. den Bosch, "Next Steps in Signaling (NSIS): Framework," RFC 4080 (Informational), Internet Engineering Task Force, Jun. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4080.txt

[11] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)," RFC 3489 (Proposed Standard), Internet Engineering Task Force, Mar. 2003, obsoleted by RFC 5389. [Online]. Available: http://www.ietf.org/rfc/rfc3489.txt

[12] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5766.txt

[13] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[14] J. Salz, A. C. Snoeren, and H. Balakrishnan, "TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-end Network Services," in 4th USENIX Symposium on Internet Technologies and Systems (USITS), 2003, pp. 211–224.

[15] D. A. Maltz and P. Bhagwat, "MSOCKS: An Architecture for Transport Layer Mobility," in IEEE INFOCOM, vol. 3. IEEE, 1998, pp. 1037–1045.