Paving the Way for NFV: Simplifying Middlebox Modifications using StateAlyzr

Junaid Khalid, Aaron Gember-Jacobson, Roney Michael,
Anubhavnidhi Abhashkumar, Aditya Akella
University of Wisconsin-Madison

Abstract

Important Network Functions Virtualization (NFV) scenarios such as ensuring middlebox fault tolerance or elasticity require redistribution of internal middlebox state. While many useful frameworks exist today for migrating/cloning internal state, they require modifications to middlebox code to identify needed state. This process is tedious and manual, hindering the adoption of such frameworks. We present a framework-independent system, StateAlyzr, that embodies novel algorithms adapted from program analysis to provably and automatically identify all state that must be migrated/cloned to ensure consistent middlebox output in the face of redistribution. We find that StateAlyzr reduces man-hours required for code modification by nearly 20×. We apply StateAlyzr to four open source middleboxes and find its algorithms to be highly precise. We find that a large amount of, but not all, live state matters toward packet processing in these middleboxes. StateAlyzr's algorithms can reduce the amount of state that needs redistribution by 600-8000× compared to naive schemes.

1 Introduction

Network functions virtualization (NFV) promises to offer networks great flexibility in handling middlebox load spikes and failures by helping spin up new virtual instances and dynamically redistributing traffic among instances. Central to realizing the benefits of such elasticity and fault tolerance is the ability to handle *internal middlebox state* during traffic redistribution. Because middlebox state is dynamic (it can be updated for each incoming packet) and critical (its current value determines middlebox actions), the relevant internal state must be made available when traffic is rerouted to a different middlebox instance [16, 26, 30].

Recognizing this, and given the high-overhead and poor efficiency of existing approaches for replicating and sharing application state [16, 24, 26], researchers have developed several exciting frameworks for transferring, cloning, or sharing live middlebox state across instances, e.g., OpenNF [16], FTMB [30], Split/Merge [26], Pico Replication [24], and StatelessNF [20].

However, for middleboxes to work with these frameworks, middlebox developers must modify, or at least annotate, their code to perform custom state allocation, track updates to state, and (de)serialize state objects. The central contribution of this paper is a novel, framework-independent system that greatly reduces the effort involved in making such modifications.

Three factors make such modifications difficult today: (i) middlebox software is extremely complex, and the logic to update/create different pieces of state can be intricate; (ii) there may be 10s-100s of object types that correspond to state that needs explicit handling; and (iii) middleboxes are extremely diverse. Factors i and ii make it difficult to reason about the completeness or correctness of manual modifications. And, iii means manual techniques that apply to one middlebox may not extend to another. Our own experience in modifying middleboxes to work with OpenNF [16] underscores these problems. Making even a simple monitoring appliance (PRADS [6], with 10K LOC) OpenNF-compliant took over 120 man-hours. We had to iterate over multiple code changes and corresponding unit tests to ascertain completeness of our modifications; moreover, the process we used for modifying this middlebox could not be easily adapted to other more complex ones!

These difficulties significantly raise the bar for the adoption of these otherwise immensely useful state handling frameworks. To reduce manual effort and ease adoption, we develop StateAlyzr, a system that relies on *data and control-flow analysis* to automate identification of state objects that need explicit handling. Using StateAlyzr's output, developers can easily make framework-compliant changes to arbitrary middleboxes, e.g., identify which state to allocate using custom libraries for [20, 24, 26], determine where to track updates to state [16, 26, 30], (de)serialize relevant state objects for transfer/-cloning [16], and merge externally provided state with internal structures [16, 24]. In practice we find StateAlyzr to be highly effective. For example, leveraging StateAlyzr to make PRADS OpenNF-compliant took under 6 man-hours of work.

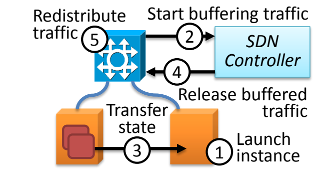
Importantly, transferring/cloning state objects identified with StateAlyzr is provably *sound* and *precise*. The former means that the aggregate output of a collection of instances following redistribution is equivalent to the output that would have been produced had redistribution not occurred. The latter means that StateAlyzr identifies minimal state to transfer so as to ensure that redistribution offers good performance and incurs low overhead.

However, achieving high precision without compromising soundness is challenging. Key attributes of middlebox code contribute to this: e.g., numerous data structures and procedures, large callgraphs, heavy use of (multi-level) pointers, and indirect calls to packet processing routines that modify state (See Table 2).

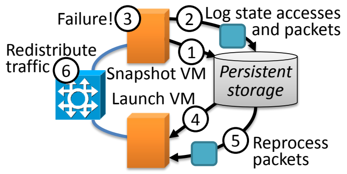
To overcome these challenges, StateAlyzr cleverly adapts program analysis techniques, such as slicing [18, 33] and pointer analysis [9, 31], to typical middlebox code structure and design patterns, contributing new algorithms for detailed classification of middlebox state. These algorithms can automatically identify: (*i*) variables corresponding to state objects that pertain to individual or groups of flows, (*ii*) the subset of these that correspond to state objects that can be updated by an arbitrary incoming packet at runtime, (*iii*) the flow space corresponding to a state object, (*iv*) middlebox I/O actions that are impacted by each state object, and (*v*) objects updated at runtime by an incoming packet.

To evaluate StateAlyzr, we both prove that our algorithms are sound (Appendix B) and use experiments to demonstrate precision and the resultant impact on the efficiency of state transfer/cloning. We run StateAlyzr on four open source middleboxes—Passive Real-time Asset Detection System (PRADS) [6], HAProxy load balancer [2], Snort Intrusion Detection System [7], and OpenVPN gateway [5]—and find:

- StateAlyzr's algorithms improve precision significantly: whereas the middleboxes have 1500-18k variables, only 29-131 correspond to state that needs explicit handling, and 10-148 are updateable at run time. By automatically identifying updateable state, StateAlyzr allows developers to focus on the necessary subset of variables among the many present. StateAlyzr can be imprecise: 18% of the updateable variables are mis-labeled (they are in fact read-only), but the information StateAlyzr provides allows developers to ignore processing these variables.
- Using StateAlyzr output, we modified PRADS and Snort to support fault tolerance using OpenNF [16]. We find that StateAlyzr reduces the manual effort needed. We could modify Snort (our most complex middlebox) and PRADS in 90 and 6 man-hours, respectively. Further, by helping track which flowspace an incoming packet belongs to, and which state objects it had updated, StateAlyzr reduces unneeded runtime state transfers between the primary and backup instances of PRADS and Snort by 600× and 8000× respectively compared to naive approaches.
- StateAlyzr can process middlebox code in a reasonable amount of time. Finally, it helped us identify important variables that we missed in our earlier modifications to PRADS, underscoring its usefulness.



(a) Scaling with Split/Merge [26]



(b) Failure recovery with FTMB [30]

Figure 1: Scaling and failure recovery process with recently state management frameworks

2 Motivation

A central goal of NFV is to create more scalable and fault tolerant middlebox deployments, where middleboxes *automatically scale* themselves in accordance with network load and *automatically heal* themselves when software, hardware, or link failures occur [4]. Scaling, and possibly fault tolerance, requires launching middlebox instances on demand. Both require redistributing network traffic among instances, as shown in Figure 1.

2.1 Need for Handling State

Middlebox scaling and failure recovery should be transparent to end-users and applications. Key to ensuring this is maintaining *output equivalence*: for any input traffic stream, the aggregate output of a dynamic set of middlebox instances should be equivalent to the output produced by a single, monolithic, always-available instance that processes the entire input [26]. The output may include network traffic and middlebox logs.

As shown in prior works [16, 26, 30], achieving output equivalence is hard because middleboxes are *stateful*. Every packet the middlebox receives may trigger updates to multiple pieces of internal state, and middlebox output is highly dependent on the current state. Thus, malfunctions can occur when traffic is rerouted to a middlebox instance without *the relevant internal state being made available at the instance*. Approaches like naively rerouting newly arriving flows or forcibly rerouting flows with pertinent state can violate output equivalence. The reader is referred to [16, 24] for a more formal treatment of the need to handle internal state.

2.2 Approaches for Handling State

Traditional approaches for replicating and sharing application state are resource intensive and slow [16, 24, 26]. Thus, researchers have introduced fast and efficient frameworks that transfer, clone, or share live internal middlebox state across instances. Examples include:

		Required Modifications			
Framework	Provides	State Alloc.	State Access	Serial-ization	Merge State
Split/Merge [26]	Elasticity	✓	✓		✓
Pico Rep. [24]	Fault tol.	✓	✓		
OpenNF [16]	Both			✓	✓
FTMB [30]	Fault tol.		✓		
StatelessNF [20]	Both	✓	✓		

Table 1: Middlebox modifications in different frameworks

Split/Merge [26] and *StatelessNF* [20] that focus on elasticity; *Pico replication* [24] and *FTMB* [30] that focus on fault tolerance; and *OpenNF* [16] that applies to both. Unfortunately, these frameworks require detailed modifications to middlebox code to handle state (see Table 1):

- Split/Merge [26] and Pico Replication [24] require middleboxes to allocate and access all per- and cross-flow state—i.e., state that supports the processing of multiple packets within and across flows, respectively—through a specialized shared library, instead of using system-provided functions (e.g., malloc). This allows the frameworks to transfer and replicate middlebox state without serializing or updating middlebox-internal structures.
- OpenNF [16] requires middleboxes to identify and serialize per- and cross-flow state objects pertaining to a particular flowspace, as well as deserialize and integrate objects received from other middlebox instances. This allows OpenNF to transfer and copy flow-related state between middlebox instances.
- FTMB [30] requires middleboxes to log: (*i*) accesses to cross-flow state, and (*ii*) invocations of non-deterministic functions (e.g., `gettimeofday`). The logs allow FTMB to deterministically reprocess packets on a different middlebox instance in case the current instance fails before an up-to-date snapshot of its state can be captured.
- StatelessNF [20] requires middleboxes to create, read, and update all state values from a central, RDMA (remote direct memory access) based key/-value store. This enables any middlebox instance to have access to any state, and hence any instance can safely process any packet.

Making the above modifications to middleboxes is difficult because middlebox code is complex. As shown in Table 2, several popular middleboxes have between 60K and 275K lines of code (LOC), dozens of different structures and classes, and, in some cases, complex event-based control flow. If a developer misses a change to some structure, class, or function, then output equivalence may be violated under certain input patterns, and a middlebox may fail in unexpected ways at run time. FTMB is the only system that aims to avoid such problems. It automatically modifies middleboxes using LLVM [3]. However, there are two problems: (i) developers must still manually specify which variables may contain/point-to cross-flow state; (ii) the tool is limited to Click-based middleboxes [21].

2.3 Simplifying Modification and its Requirements

Making the aforementioned changes to even simple middleboxes can take numerous man-hours as our own experience with OpenNF suggests. This is a serious barrier to adopting any of the previously mentioned systems.

A system that can automatically identify what state a middlebox creates, where the state is created, and how the state is used could be immensely helpful in reducing the man-hours. It can provide developers guidance on writing custom state allocation routines, and on adding appropriate state filtering, serialization, and merging functions. Thus, it would greatly lower the barriers to adopting the above frameworks.

Building such a system is challenging because of *soundness* and *precision* requirements. Soundness means that the system must not miss any types, storage locations, allocations, or uses of state required for output equivalence. A precise system identifies the minimal set of state that requires special handling to ensure state handling at runtime is fast and low-overhead.

2.4 Options

Well-known *program analysis* approaches can be applied to identify middlebox state and its characteristics.

Dynamic analysis. We could use dynamic taint analysis [29] to monitor which pieces of state are used and modified while a middlebox processes some sample input. Unfortunately, the sample inputs may not exercise all code paths, causing the analysis to miss some state. We also find that such monitoring can significantly slow middleboxes down (e.g., PRADS [6] and Snort IDS [7] are slowed down > 10×).

Static analysis. Alternatively, we could use symbolic execution [10] or data-/control-flow analysis [15, 18].[1]

Symbolic execution can be employed to explore all possible code paths by representing input and runtime state as a series of symbols rather than concrete values. We can then track the state used in each path. While this is sound, the complexity of most middleboxes (Table 2) makes it impossible to explore all execution paths in a tractable amount of time. For example, we symbolically executed PRADS—which has just 10K LOC—for 8 hours using S2E [10], and only 13% of PRAD's code was covered. The complexity worsens exponentially for middleboxes with larger codebases. Recent advances in symbolic execution of middleboxes [14] do not help as they over come state space explosion by abstracting away middlebox state, which is precisely what we aim to analyze.

[1] Abstract interpretation [12] is another candidate, but it suffers from the well known problem of incompleteness, i.e., it over-approximates the middlebox's processing and may not identify all relevant state.

Middlebox	LOC (C/C++)	Classes/ Structs	Event based?	Level of pointers	Number of procedures	Size of callgraph
Snort IDS [7]	275K	898	No	10	4617	3391
HAProxy load balancer [2]	63K	191*	No	8*	2560	1018
OpenVPN [5]	62K	194*	No	2*	2023	1184
PRADS asset detector [6]	10K	40	No	4	297	115
Bro IDS [23]	97K	1798	No	-	3034	-
Squid caching proxy [8]	166K	875	Yes	-	2133	-

*Shows the lower bound. It does not include the number of structs used by the libraries and kernel.

Table 2: Code complexity for popular middleboxes. Those above the line are analyzed in greater detail later.

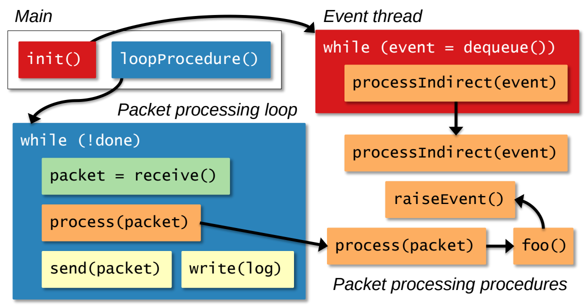


Figure 2: Logical structure of middlebox code

In this paper, we make clever use of *data-/control-flow analysis* to automatically evaluate how to handle middlebox state. Naively applying standard data-/control-flow analysis identifies all variables as pertaining to 'state that needs handling' (e.g., variables pertaining to per-packet state, read-only state, and state that falls outside the scope of a flowspace of interest); if developers modify a middlebox to specially handle all these variables, it can result in arbitrarily poor runtime performance during redistribution. We show how *middlebox code structure and design patterns* can be used to design *novel algorithms* that employ *static program analysis* techniques in a way that significantly improves *precision* without compromising *soundness*. Our approach is general and does not assume use of any particular state management framework.

3 Overview of StateAlyzr

Most middleboxes' code can be logically divided into three basic parts (Figure 2): initialization, packet receive loop, and packet processing. The initialization code runs when the middlebox starts. It reads and parses configuration input, loads supplementary modules or files, and opens log files. All of this can be done in the `main()` procedure, or in separate procedures called by `main`. The packet receive loop is responsible for reading a packet (or byte stream) from the kernel (via a socket) and passing it to the packet processing procedure(s). The latter analyzes, and potentially modifies, the packet. This procedure(s) reads/writes internal middlebox state to inform the processing of the current (and future) packet.

Our approach consists of three primary stages that leverage this structure. In each stage we further refine our characterization of a middlebox's state. The stages and their main challenges are described next:

1) Identify Per-/Cross-Flow State. In the first stage, we identify the storage location for all per- and cross-flow state created by the middlebox. The final output of this stage is a list of what we call *top-level variables* that contain or indirectly refer to such state.

Unlike state that is only used for processing the current packet, per-/cross-flow state influences other packets' processing. Consequently, the *lifetime* of this state extends beyond the processing a single packet. We leverage this property, along with knowledge of the relation between variable and value lifetimes, to first identify variables that may contain or refer to per-/cross-flow state.

We improve precision by considering which variables are actually *used* in packet processing code, thereby eliminating variables that contain or refer to state that is only used for middlebox initialization. We call the remaining variables "top-level". The main challenge here is dealing with indirect calls to packet processing in event-based middleboxes (Figure 2), which complicate the task of identifying all packet processing code. We develop an algorithm that adapts *forward program slicing* [18] to address this challenge (§4.1).

2) Identify Updateable State. The second stage further categorizes state based on whether it may be updated while a packet is processed. If state is read-only, we can avoid repeated cloning (in Pico Replication and OpenNF), avoid unnecessary logging of accesses (FTMB), and allow simultaneous access from multiple instances (StatelessNF); all of these will reduce the frameworks' overhead. We can trivially identify updateable state by looking for assignment statements in packet processing procedures. However, this strawman is complicated by heavy use of pointers in middlebox code which can be used to indirect state update. To address this challenge we show how to employ flow-, context-, and field-insensitive *pointer analysis* [9, 31] (§4.2).

3) Identify States' Flowspace Dimensions. Finally, the third stage determines a state's *flowspace*: a set of packet header fields (e.g. src_ip, dest_ip, src_port, dest_port & proto) that delineate the subset of traffic that relates to the state. Flowspace must be considered when modifying a middlebox to use custom allocation functions [24, 26] or filter state in preparation for export [16]. It is important to avoid the inclusion of irrelevant header fields and the exclusion of relevant fields in a state's flowspace, because it impacts runtime correctness and performance, respectively. To solve this problem we developed an algorithm that leverages common state access patterns in middleboxes to identify program points where we can apply *program chopping* [27] to determine relevant header fields (§4.3).

Soundness. In order for StateAlyzr to be sound it is necessary for these three stages to be sound. In Appendix B, we prove the soundness of our algorithms.

Assumptions about middlebox code. Our proofs are based on the assumption that middleboxes use standard API or system calls to read/write packets and hashtables or link-lists to store state. These assumption are not limitations of our analysis algorithms. Instead, they are made to ease the implementation of StateAlyzr. Our implementation can be extended to add additional packet read-/write methods or other data structures to store the state.

4 StateAlyzr Foundations

We now describe our novel algorithms for detailed state classification. To describe the algorithms, we use the example of a simple middlebox that blocks external hosts creating too many new connections (Figure 3).

4.1 Per-/Cross-Flow State

Our analysis begins by identifying the storage location for all relevant per- and cross-flow state created by the middlebox. This has two parts: (*i*) exhaustively identifying persistent variables to ensure soundness, and (*ii*) carefully limiting to top-level variables that contain or refer to per-/cross-flow values to ensure precision.

4.1.1 Identifying Persistent Variables

Because per-/cross-flow state necessarily influences two or more packets within/across flows, values corresponding to such state must be created during or prior to the processing of one packet, and be destroyed during or after the processing of a subsequent packet. Hence, the corresponding variables must be *persistent*, i.e., their values persist beyond a single iteration of the packet processing loop. In Figure 3, variables declared on lines 7 to 11 are persistent, whereas `curr` on line 61 is not. Our algorithm first identifies such variables.

Analysis Algorithm. We traverse a middlebox's code, as shown in Figure 4. The values of all global and static variables exist for the entire duration of the middlebox's execution, so these variables are always persistent. Variables local to the *loop-procedure*[2]—i.e., the procedure containing the packet processing loop—exist for the duration of this procedure, and hence the duration of the packet processing loop, so they are also persistent.

Local variables of procedures that precede the loop-procedure on the call stack are also persistent, because the procedures' stack frames last longer than the packet processing loop. However, these variables cannot be used within the packet processing loop, or a procedure called therein, because the variables are out of scope. Thus we exclude these from our list of persistent variables, improving precision.

The above analysis implicitly considers heap-allocated values by considering the values of global, static, and local variables, which can point to values on the heap. Values on the heap exist until they are explicitly freed (or the middlebox terminates), but their *usable lifetime* is lim-

[2] To automatically detect packet processing loops, we use the fact that middleboxes read packets using standard library/system functions.

```
1  struct host {
2    uint ip;
3    int count;
4    struct host *next;
5  }
6
7  pcap_t *intPcap, *extPcap;
8  int threshold;
9  char * queue[100];
10 int head = 0, tail = 0;
11 struct host *hosts = NULL;
12
13 int main(int argc, char **argv) {
14   pthread_t thread;
15   intPcap = pcap_create(argv[0]);
16   extPcap = pcap_create(argv[1]);
17   threshold = atoi(argv[2]);
18   pthread_create (&thread,(void*)&processPacket);
19 }
20
21 int loopProcedure() {
22   while(1) {
23     struct pcap_pkthdr pcapHdr;
24     char *pkt = pcap_next(extPcap, &pcapHdr);
25     ifFull_Wait();
26     enqueue(pkt);
27     if (entry->count < threshold)
28       pcap_inject(intPcap, pkt, pcapHdr->caplen);
29 } }
30
31 void enqueue(char* pkt){
32   head = (head + 1)%100;
33   queue[head] = pkt;
34 }
35
36 char* dequeue(){
37    int *index = &tail;
38    *index = (*index + 1)%100;
39    return queue[*index];
40 }
41
42 void processPacket(){
43   while(1){
44     ifEmpty_Wait();
45     char* pkt = dequeue();
46     struct ethhdr *ethHdr= (struct ethhdr)pkt;
47     struct iphdr *ipHdr= (struct iphdr*)(ethHdr+1);
48     struct tcphdr *tcpHdr= (struct tcphdr*)(ipHdr+1);
49     struct host *entry= lookup(ipHdr->saddr, hosts);
50     if (NULL == host){
51       struct host *new = malloc(sizeof(struct host));
52       new->ip = ipHdr->saddr;
53       new->next = hosts;
54       hosts = new;
55     }
56     if (tcpHdr->syn && !tcpHdr->ack)
57       entry->count++;
58 }  }
59
60 struct host *lookup(uint ip) {
61   struct host *curr = hosts;
62   while (curr != NULL) {
63     if (curr->ip == ip)
64       return curr;
65     curr = curr->next;
66 } }
```

Figure 3: Code for our running example.

```
Input: prog
Output: persistVars
1 persistVars = {}
2 persistVars = persistVars ∪ GlobalVarDecls(prog)
3 foreach proc in Procedures(prog) do
4     persistVars = persistVars ∪ StaticVarDecls(proc)
5 persistVars = persistVars ∪ LocalVarDecls(loopProc)
6 persistVars = persistVars ∪ FormalParams(loopProc)
```

Figure 4: Identifying persistent variables

ited to the time frame in which they are reachable from a variable's value.[3] Therefore, we can conclude that a heap-allocated value's persistence is predicated on the persistence of a variable identified by our algorithm.

4.1.2 Limiting to Top-level Variables

The above algorithm identifies a superset of variables that may be bound, or point, to per-/cross-flow state. It includes variables bound to state used in initialization for loading/processing configuration/signature files: e.g., variables `intPcap` and `extPcap` in Figure 3. Such variables don't need handling during traffic redistribution; they can simply be copied when an instance is launched. To eliminate such variables and improve precision, the key insight we leverage is that, by definition, per-/cross-flow state is *used* in some way during packet processing. However, identifying all such variables is non-trivial, and missing variables impact analysis soundness.

```
Input: prog, persistVars
Output: pktProcs, percrossflowVars
1  pktProcs = {}
2  sdg = SystemDependenceGraph(prog)
3  foreach stmt in Statements(loopProc) do
   //Statements() returns all statements in a procedure
4      if stmt calls PKT_RECV_FUNC then
5          slice = ForwardSlice(sdg, stmt, stmt.LHS)
6          pktProcs = pktProcs ∪ Procedures(slice)
           //Procedures() returns all procedures in a slice
7  percrossflowVars = {}
8  foreach proc in pktProcs do
9      foreach stmt in Statements(proc) do
10         foreach var in Vars(stmt) do
           //Vars() returns all variables used in a statement
11             if var in persistVars then
12                 percrossflowVars = percrossflowVars ∪ {var}
```

Figure 5: Identifying per-/cross-flow variables

Identifying Packet Processing Procedures. Figure 5 shows our algorithm for identifying top-level variables that contain or refer to per-/cross-flow values. The first half of the algorithm (lines 1–6) focuses on identifying packet processing code. Obviously any code contained in the packet processing loop is used for processing packets, but, crucially, the code of procedures (indirectly) called from within the loop is also packet processing code.

We considered a strawman approach of using call graphs to identify packet processing procedure. A call graph is constructed by starting at each procedure call within the packet processing loop, and classifying each appearing procedure as a packet processing procedure. However, this analysis does not capture packet processing procedures that are called indirectly. The Squid proxy, e.g., does initial processing of the received packet, then enqueues an event to trigger further processing through later calls to additional procedures. Hence the analysis may incorrectly eliminate some legitimate per-/cross-flow state which is used in such procedures.

Thus, we need an approach that exhaustively considers the dependencies between the receipt of a packet and both direct and indirect invocations of packet processing procedures. Below, we show how *system dependence graphs* [15] and *program slicing* [18] can be used for this.

A *system dependence graph* (SDG) consists of multiple program dependence graphs (PDGs) — one for each procedure. Each PDG contains vertices for each statement along with their data and control dependency edges. A *data dependence* edge is created between statements p and q if there is an execution path between them, and p may update the value of some variable that q reads. A *control dependence* edge is created if p is a conditional statement, and whether or not q executes depends on p. A snippet of the control and data edges for our example in Figure 3 is in Figure 6.

Whereas control edges capture direct invocations of packet processing, we can rely on data edges to capture indirect procedure calls. For example, the dashed yellow lines in Figure 6 fail to capture invocation of the `processPacket` procedure on bottom right (because there is no control edge from the while loop or any of its subsequent procedures to `processPacket`). In contrast, we can follow the data edges, the dashed red line, to track such calls.

Given a middlebox's SDG, we compute a *forward program slice* from a packet receive function call for the variable which stores the received packet. A forward slice contains the set of *statements that are affected* by the value of a variable starting from a specific point in the program [18]. Most middleboxes use standard library/system functions to receive packets—e.g., `pcap_next`, or `recv`—so we can easily identify these calls and the variable pointing to the received packet. We consider any procedure appearing in the computed slice to be a packet processing procedure. For middleboxes which invoke packet receive functions at multiple points, we compute forward slices from every call site and take the union of the procedures appearing in all such slices.

Values Used in Packet Processing Procedures. The second half of our algorithm (Figure 5, lines 7–12) focuses on identifying persistent values that are used within some

[3] A heap value whose lifetime is longer than its usable lifetime is a memory leak.

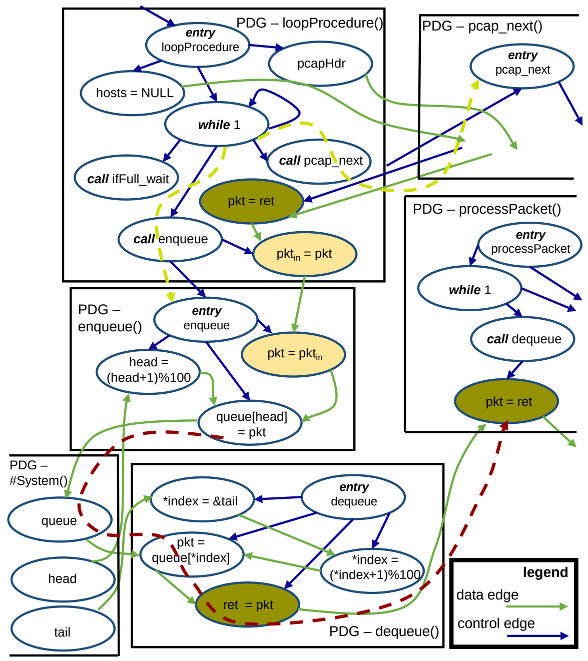


Figure 6: Snippet of System dependence graph (SDG) for the code in Figure 3; green edges indicate data dependencies and blue edges indicate control dependencies; light yellow nodes represent formal and actual parameters, while dark yellow nodes represent return values.

packet processing procedure. We analyze each statement in the packet processing procedures. If the statement contains a persistent variable, then we mark that persistent variable as a top-level variable.

4.2 Updateable State

Next, we delineate *updateable* top-level variables from *read only* variables to further improve precision. In Figure 3, variable `head`, `tail`, `hosts` and `queue` are updateable, whereas `threshold` is not. Because state is updated through assignment statements, one strawman choice here is to statically identify top-level variables on the left-hand-side (LHS) of assignment statements. In Figure 3, this identifies `head`, `hosts` and `queue`.

However, this falls short due to *aliasing*, where multiple variables are bound to the same storage location due to the use of pointers [11]. Aliasing allows a value reachable from a top-level variable to be updated through the use of a different variable. Thus our strawman can mis-label top-level variables as read-only, compromising soundness. For example, `tail` is mislabeled in Figure 3, because it never appears on the LHS of assignment statements. But on line 38 `index` is updated which points to `tail`.

Analysis Algorithm. We develop an algorithm to identify *updateable top-level variable* (Figure 7). Since we are concerned with variables whose (referenced) values are updated during packet processing, we analyze each assignment statement contained in the packet process-

```
Input: pktProcs, percrossflowVars
Output: updateableVars
percrossflowVars = {}
foreach proc in pktProcs do
    foreach stmt in AssignmentStmts(proc) do
    //AssignmentStmts() returns all assignment
    statements in a procedure
        foreach var in percrossflowVars do
            if stmt.LHS == var
                or var in PointsTo(stmt.LHS)
                or PointsTo(var) ∩ PointsTo(stmt.LHS) ≠ ∅
            then
                updateableVars = updateableVars ∪ {var}
```

Figure 7: Identifying updateable variables

ing procedures identified in the first stage of our analysis (§4.1.2). If the assignment statement's LHS contains a top-level variable, then we mark the variable as updateable (similar to our strawman). Otherwise, we compute the *points-to* set for the variable on the LHS and compare this with the set of updateable top-level variables and their points-to sets. A variable's points-to set contains all variables whose associated storage locations are reachable from the variable. To compute this set, we employ flow-, context-, and field-insensitive pointer analysis [9]. If the points-to set of the variable on the LHS contains a top-level variable, or has a non-null intersection with the points-to set of a top-level variable, then we mark the top-level variable as updateable.

Due to limitations of pointer analysis, our algorithm may still mark read-only top-level variables as updateable. E.g., field insensitive pointer analysis can mark a top-level struct variable as updateable even if just one of its subfields is updateable.

4.3 State Flowspaces

Finally, we identify the packet header fields that define the flowspace associated with the values of each top-level variable. Identifying too fine-grained of a flowspace for a value—i.e., more header fields than those that actually define the flowspace—is unsound; such an error will cause a middlebox to incorrectly filter out the value when it is requested by a middlebox state management framework [16, 20, 24, 26]. Contrarily, assuming an overly permissive flowspace (e.g., the entire flowspace) for a value hurts precision.

To identify flowspaces, we leverage common middlebox design patterns in updating or accessing state. Middleboxes typically use simple data structures (e.g., a hash table or linked list) to organize state of the same type for different network entities (connections, applications, subnets, URLs, etc.). When processing a packet, a middlebox uses header fields[4] to lookup the entry in the

[4]In cases where keys are not based on the packet header fields e.g. URL, a middlebox usually keeps another data structure to maintain the

data structure that contains a reference to the values that should be read/updated for this packet. In the case of a hash table, the middlebox computes an *index* from the packet header fields to identify the entry pointing to the relevant values. For a linked list, the middlebox *iterates* over entries in the data structure and compares packet header fields against the values pointed to by the entry.

```
    Input: pktProcs, percrossflowVars
    Output: chop, flowspace
1   keyedVars = {}
2   foreach var in percrossflowVars do
3       if Type(var) == pointer
          or Type(var) == struct then
4           keyedVars = keyedVars ∪ {keyedVars}
5   foreach proc in pktProcs do
6       foreach loopStmt in LoopStmts(proc) do
7           condVars = {}
8           foreach var in Vars(loopStmt.condition) do
9               if var in keyedVars
                  or PointsTo(var) ∩ keyedVars ≠ ∅ then
10                  for condStmt in
                    ConditionalStmts(loopStmt.body) do
11                      for condVar in Vars(condStmt) do
12                          if condVar ≠ var then
13                              condVars = condVars ∪ {condVar}
14          chop = Chop(sdg,pktVar,condVars)
15          flowspace = ExtractFlowspace(chop)
```

Figure 8: Identifying packet header fields that define a per-/cross-flow variable's associated flowspace

Algorithm. We leverage the above design patterns in our algorithm shown in Figure 8. In the first step (lines 2-4), if the top-level variable is a struct or a pointer, we mark it as a possible candidate for having a flowspace associated with it. This filters out all the top-level variables which cannot represent more than one entry; e.g., variables `head` and `tail` in Figure 3.

We assume that middleboxes use hash tables or linked lists to organize their values,[5] and that these data structures are accessed using:
square brackets, e.g.

```
entry = table[index];
```

pointer arithmetic, e.g.

```
entry = head + offset;
```

or iteration[6], e.g.

```
while(entry->next!=null){entry=entry->next;}
for(i=0; i<list.length; i++) {...}
```

The second step is thus to identify all statements like these where a top-level variable marked above is on the right-hand-side (RHS) of the statement (square brackets or pointer arithmetic scenario) or in the conditional expression (iteration scenario).

When square brackets or pointer arithmetic are used, we compute a *chop* between the variables in the access statement and the variable containing the packet returned by the packet receive procedure. A chop between a set of variables U at program point p and a set of variables V at program point q is the set of statements that (i) may be affected by the value of variables in U at point p, and (ii) may affect the values of variables in V at point q. Thus, the chop we compute above is a snippet of executable code which takes a packet as input and outputs the index or offset required to extract the value from the hashtable.

In a similar fashion, when iteration is used, we identify all conditional statements in the body of the loop. We compute a chop between the packet returned by the packet receive procedure and the set of all the variables in the conditional expression which do not point to any of the top-level variables; in our example (Figure 3), the chop starts at line 24 and terminates at line 63. We output the resulting chops, which collectively contain all conditional statements that are required to lookup a value in a linked list data structure based on a flow space definition. Assuming that the middlebox accesses packet fields using standard system-provided structs (e.g., `struct ip` as defined in `netinet/ip.h`), we conduct simple string matching on the code snippets to produce a list of packet header fields that define a state's flowspace.

5 Enhancements

Data and control flow analysis can help improve precision, but they have some limitations in that they cannot guarantee that exactly the relevant state and nothing else has been identified. In particular, static analysis cannot differentiate between multiple memory regions that are allocated through separate invocations of malloc from the same call site. Therefore, we cannot statically determine if only a subset of these memory regions have been updated after processing a set of packets. To overcome potential efficiency loss due to such limitations, we can employ custom algorithms that boost precision in specific settings. We present two candidates below.

5.1 Output-Impacting State

In addition to the three main code blocks (Figure 2), middleboxes may optionally have packet and log output functions. These pass a packet to the kernel for forwarding and record the middlebox's observations and actions in a log file, respectively. These functions are usually called from within the packet processing procedure(s).

In some cases, operators may desire output equivalence only for specific types of output. For example, an operator may want to ensure client connections are not broken when a NAT fails—i.e., packet output should be equivalent—but may not care if the log of NAT'd connec-

mapping between such keys and packet header fields

[5]Our approach can easily be extended to other data structures.

[6]Middleboxes may also use recursion, but we have not found this access pattern in the middleboxes we study, so we do not consider it in our algorithm.

tions is accurate. In such cases, internal state that only impacts non-essential forms of output does not need special handling during redistribution and can be ignored.

To aid such optimizations, we develop an algorithm to identify the type of output that updateable state affects. We use two key insights. First, middleboxes typically use *standard libraries and system calls* to produce packet and log output: either PCAP (e.g. `pcap_dump`) or socket (e.g. `send`) functions for the former, and regular I/O functions (e.g. `write`) for the latter.[7] Second, the output produced by these functions can only be impacted by a *handful of parameters* passed to these functions. Thus, we focus on the call sites of these functions, and their parameters.

Algorithm. We use *program slicing* [18] to identify the dependencies between a specific type of output and updateable variables. We sketch the algorithm and relegate details to Appendix A. We first identify the call sites of packet or log output functions by checking each statement in each packet processing procedure (§4.1.2). Then we use the SDG produced in the first stage of our analysis (Figure 5) to compute a *backward slice* from each call site. Such a slice contains the set of statements that affect (*i*) whether the procedure call is executed, and (*ii*) the value of the variables used in the procedure call, such as the parameters passed to the output function. We examine each statement in a backward slice to determine whether it contains an updateable per-/cross-flow variable. Such variables are marked as impacting packet (or log) output.

5.2 Tracking Runtime Updates

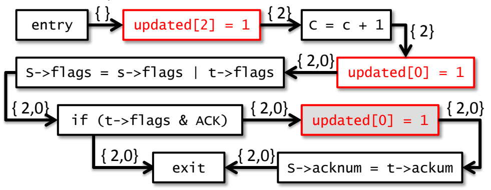
Developers aiming to design fault-tolerant middleboxes can use the algorithms in §4 and §5.1 to efficiently clone state to backup instances. For example, if traffic will be distributed among multiple instances in the case of failure, then only state whose flowspace overlaps with that assigned to a specific instance needs to be cloned to that instance. However, the potential performance gains from these optimizations may be limited due to constraints imposed by data/control-flow analysis. For example, our analysis can only identify whether a persistent variable's value *may* be updated during the middlebox's execution. If we can determine at runtime exactly which values are updated, and when, then we can further improve the efficiency of state cloning and speed up failover.

To achieve higher precision, we must use (simple) run time monitoring. For example, we can track, at run time, whether part of an object is updated during packet processing. To implement this monitoring, we must modify the middlebox to set an "updated bit" whenever a value reachable from a top-level variable is updated during packet processing. Figure 9a shows such modifications, in red, for a simple middlebox. We create a unique

[7] Our approach can be easily extended to consider non-standard output functions.

```
1  struct conn tbl[1000]; // Assigned id 0
2  int count; // Assigned id 1
3  int tcpcnt; // Assigned id 2
4  char updated[3];
5  void main() {
6    while(1) {
7      char *pkt = recv();
8      updated[1] = 1;
9      count = count + 1;
10     struct *iphdr i = getIpHdr(pkt);
11     if (i->protocol == TCP) {
12       hdl(&tcpcnt, &tbl[hash(pkt)], getTcpHdr(pkt));
13 } } }
14 void hdl(int *c,struct conn *s,struct tcphdr *t) {
15   updated[2] = 1;
16   c = c + 1;
17   updated[0] = 1;
18   s->flags = s->flags | t->flags;
19   if (t->flags & ACK)
20     updated[0] = 1; // Pruned
21     s->acknum = t->acknum;
22 } }
```

(a) Example middlebox code instrumented for update tracking at run time; statements in red are inserted based on our analysis



(b) Annotated control flow graph used for pruning redundant updated-bit-setting (shaded) statements

Figure 9: Implementing update tracking at run time

updated bit for each top-level variable—there are three such variables in the example—and we set the appropriate bit before any statement that updates a value that may be reachable from the corresponding variable.

We use the same analysis discussed in §4.2 to determine where to insert statements to set updated bits. For any statement where a top-level variable is updated, we insert a statement—just prior to the assignment statement—that sets the appropriate updated bit.

However, this approach can add a lot more code than needed: if one assignment statement *always* executes before another, and they *always* update the same value, then we only need to set the updated bit before the first assignment statement. For example, line 21 in Figure 9a updates the same compound value as line 18, so the code on line 20 is redundant.

We use a straightforward control flow analysis to prune unneeded updated-bit-setting statements. First, we construct a control flow graph (CFG) for each modified packet processing procedure. Next, we perform a depth-first traversal of each CFG, tracking the set of updated bits that have been set along the path; as we traverse each edge, we label it with the current set of updated bits. Figure 9b shows this annotated CFG for the `handleTcp` procedure shown in lines 14-22 of Figure 9a. Lastly, for each updated-bit-setting statement in a procedure's CFG, we check whether the bit being set is included in the label for every incoming edge. If this is true, then we prune the statement; e.g., we prune line 20 in Figure 9a.

6 Implementation

We implement StateAlyzr using CodeSurfer [1] which has built-in support for constructing CFGs, performing flow- and context-insensitive pointer analysis, constructing PDGs/SDGs, and computing forward/backward slices and chops for C/C++ code. CodeSufer uses proven sound algorithms to implement these static analysis techniques. We use CodeSurfer's Scheme API to access output from these analyses in our algorithms. We applied StateAlyzr to four middleboxes: PRADS asset monitoring [6] and Snort Intrusion Detection System [7], HAproxy load balancer [2], and OpenVPN gateway [5].

Fault Tolerance. We use the output from StateAlyzr to add fault tolerance to PRADS and Snort, both off-path middleboxes. We added code to both to export/import internal state (to a standby). We used the output of our first two analysis phases (§4.1 and §4.2) to know which top level variables' values we need to export, and where in a hot-standby we should store them. We used the output of our third analysis phase (§4.3) as the basis for code that looks up per-/cross-flow state values. This code takes a flowspace as input and returns an array of serialized values. We use OpenNF [16] to transfer serialized values to a hot-standby. Similarly, import code deserializes the state and stores it in the appropriate location. We also implemented both enhancements discussed in §5.

7 Evaluation

We report on the outcomes of applying StateAlyzr to four middleboxes. We address the following questions:

- *Effectiveness*: Does StateAlyzr help with making modifications to today's middleboxes? How many top-level variables do these middleboxes maintain, relative to all variables? What relative fractions of these pertain to state that may need to be handled during redistribution? How precise is StateAlyzr?
- *Runtime efficiency and manual effort:* To what extent do StateAlyzr's mechanisms help improve the runtime efficiency of state redistribution? How much manual effort does it save?
- *Practical considerations*: Does StateAlyzr take prohibitively long to run (like symbolic execution; §2.4)? Is it sound in practice?

7.1 Effectiveness

In Table 3, we present a variety of key statistics derived for the four middleboxes using StateAlyzr. We use this to highlight StateAlyzr's ability to improve precision, thereby underscoring its usefulness for developers.

The complexity of middlebox code is underscored by the overall number of variables in Table 3, which can vary between 1500 and 18k, and other relevant code complexity metrics shown in Table 2. Thus, manually identifying state that needs handling, and optimizing its transfer, is extremely difficult.

Mbox	All	Persistent	Top -level	Update -able	pkt/log output impacting	require serial- ization
PRADS	1529	61	29	10	N.A. / 6	14
Snort	18393	507	333	148	N.A. /148	176
HAproxy	7876	272	176	115	101 / 109	59
OpenVPN	8704	156	131	106	97 / 102	8

Table 3: Variables and their properties

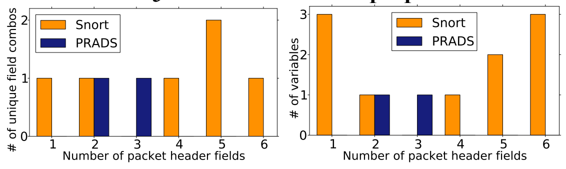


Figure 10: Flowspace dims. of keyed per-/cross-flow vars

We also note from Table 3 that StateAlyzr identifies 61-507 variables as persistent across the four middleboxes. A subset of these, 29-333, are top-level variables. Finally, 6-148 top-level variables are updateable; operators only have to deal with handling the values pertaining to these variables at run time. Snort is the most complex middlebox we analyze (≈275K lines of code) and has the largest number of top-level variables (333); the opposite is true for PRADS (10K LOC and 29 top-level variables).

The drastic reduction to the final number of updateable variables shows that naive approaches that attempt to transfer/clone values corresponding to *all* variables can be very inefficient at runtime. (We show this empirically in §7.2.) Even so, the number of updateable variables can be as high as 148, and attempting to manually identify them and argument code suitably can be very difficult. By automatically identifying them, StateAlyzr simplifies modifications; we provide further details in §7.2.

Finally, the reductions we observe in going from persistent variables to top-level variables (16-53% reduction) and further to updateable ones (19-65% reduction) show that our techniques in §4.1 and §4.2 offer useful improvements in precision.

In Figure 10, we characterize the flowspaces for the variables found in Snort and PRADS. From the left figure, we see that Snort maintains state objects that could be keyed by as many as 5 or 6 header fields; the maximum number of such fields for PRADS is 3. The figure on the right shows the number of variables that use a particular number of header fields as flowspace keys; for instance, in the case of Snort, 3 variables each are keyed on 1 and 6 fields. The total number of variables keyed on at least one key is 2 and 10 for Snort and PRADS, respectively (sum of the heights of the respective bars).

These numbers are significantly lower than the updateable variables we discovered for these middleboxes (6 and 148, respectively). Digging deeper into Snort (for example) we find that:

- 111 updateable variables pertain to all flows (i.e., a flowspace key of "*"). Of these, 59 variables are related to configurations and signatures, while 30 are function pointers (that point to different detection and processing plugins). These 89 variables can be updated from the command line at middlebox run time (when an operator provides new configurations and signatures, or new analysis plugins).
- 27 updateable variables—or 18%—are only used for processing a *single packet*; hence they don't correspond to per-/cross-flow state. This points to StateAlyzr's *imperfect precision*. These variables are global in scope and are used by different functions for processing a single incoming packet, which is why our analysis labels them as updateable. A developer can easily identify these variables and can either remove them from the list of updateable variables or modify code to make them local in scope.

7.2 Runtime efficiency and manual effort

7.2.1 Fault Tolerant Middleboxes

Using fault tolerant PRADS/Snort versions (§5), we show that StateAlyzr helps significantly cut unneeded state transfers, improving state operation time/overhead.

Man-hours needed. Modifying PRADS based on StateAlyzr analysis took roughly 6 man-hours, down from over 120 man-hours when we originally modified PRADS for OpenNF (Two different persons made these modifications.). Modifying Snort, a much more complex middlebox, took 90 man-hours. In both cases, most of the time (> 90%) was spent in writing serialization code for the data structures identified by StateAlyzr (14 for PRADS and 176 for Snort; Table 3). Providing support for exporting/importing state objects according to OpenNF APIs took just 1 and 2 hours, respectively.

Runtime benefits. We consider a primary/hot standby setup, where the primary sends a copy of the state to the hot standby after processing each packet. We use a university-to-cloud packet trace [17] with around 700k packets for our trace-based evaluation of this setup. The primary instance processes the first half of the trace file until a random point, and the hot standby takes over after that. We consider three models for operating the hot standby which reflect progressive application of the different optimizations in §4 and §5: (i) the primary instance sends a copy of all the updateable states to the hot standby, (ii) the primary instance only sends the state which applies to the flowspace of the last processed packet, and (iii) in addition to considering the flowspace, we also consider which top level variables are marked as updated for the last processed packet.

Figure 11a shows the average case results for the amount of per packet data transferred between the primary and secondary instances for all three models for PRADS.

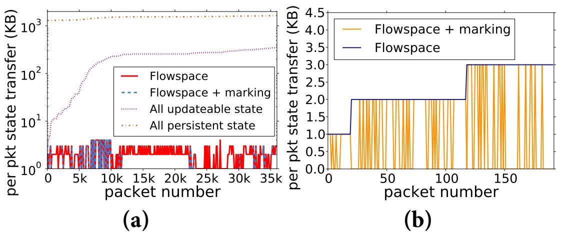


Figure 11: (a) Per packet state transfer (b) Per packet state transfer for a single connection

Transferring state which only applies to the flowspace of the last processed packet, i.e., the second model, reduces the data transferred by 305× compared to transferring all per-/cross-flow state. Furthermore, we find that the third model, i.e., run time marking of updated state variables, further reduces the amount of data transferred by 2×, on average. This is because not all values are updated for every packet: the values pertaining to a specific connection are updated for every packet of that connection, but the values pertaining to a particular host and its services are only updated when processing certain packets. This behavior is illustrated in Figure 11b, which shows the size of the state transfer after processing each of the first 200 packets in a randomly selected flow.

We measured the increase in per packet processing time purely due to the code instrumentation needed to identify state updates for highly available PRADS. We observed an average increase of 0.04μsec, which is around 0.14% of the average per packet processing time for unmodified PRADS.

Figure 12 shows the corresponding results for Snort. Transferring just the updateable state results in a 8800× reduction in the amount of state transferred compared to transferring all per-/cross-flow state. This is because, a significant portion of the persistent state in Snort consists of configuration and signatures which are never updated during packet processing. Transferring state which only applies to a particular flowspace further reduces the data transfer by 2.75×. Unlike PRADS, the amount of state transfer in the second model remains constant for a particular flow because most of the state is created on the first few packets of a flow. Finally, runtime marking further reduces the amount of state transferred by 3.6×.

7.2.2 Packet/Log Output

Table 3 includes the number of variables that impact packet or log output. For on-path HAproxy (OpenVPN), 87% (91%) of updateable variables affect packet output; a slightly higher fraction impact log output. 95 (93) variables impact both outputs. A much smaller number impacts packet output but not log (6 and 4, respectively). Another handful impact logs but not packets (14 and 9); operators who are interested in just packet output consistency can ignore transferring the state pertaining to these variables, but the benefit will likely not be significant for

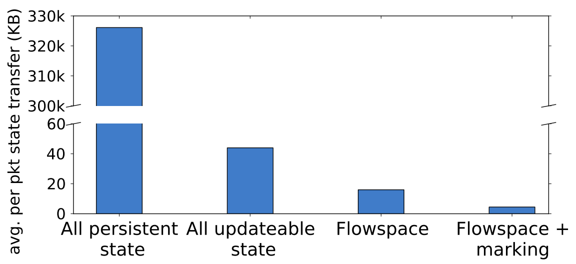


Figure 12: Per packet state transfer in Snort

these middleboxes given the low counts.

Being off-path, PRADS and Snort have no variables that impact packet output. For PRADS, 6 out of 10 updateable variables impact log output. StateAlyzr did find 4 other updateable variables—`tos`, `tstamp`, `in_pkt`, and `mtu`—but did not mark them as affecting packet output or log output. Upon manual code inspection we found that these values are updated as packets are processed, but they are never used; thus, these variables can be removed from PRADS without any impact on its output, pointing to another benefit of StateAlyzr—code clean-up.

7.3 Practicality

Table 4 shows the time and resources required to run our analysis. CodeSurfer computes data and control dependencies and points-to sets at compile time, so the middleboxes take longer than normal to compile. This phase is also memory intensive, as illustrated by peak memory usage. Snort, being complex, takes the longest to compile and analyze (≈20.5h). This is not a concern since StateAlyzr only needs to be run once, and it runs offline.

7.3.1 Empirically Verifying Soundness

Empirically showing soundness in practice is hard. Nevertheless, for the sake of completeness, we use two approaches to verify soundness of the modifications we make on the basis of StateAlyzr's outputs.

First, we use the experimental harness from §7.2. We compare logs at PRADS/Snort in the scenario where a single instance processes the complete trace file against concatenated logs of the primary and hot standby, using the trace and the three models as above. In all cases, there was no difference in the two sets of logs.

Next, we compare with manually making all changes. Recall that we had manually modified PRADS to make it OpenNF-compliant. We compared StateAlyzr's output for PRADS against the variables contained in the state transfer code we added during our prior modifications to PRADS. StateAlyzr found all variables we had considered in our prior modifications, and more. Specifically, we found that our prior modifications had *missed* an important compound value that contains a few counters along with configuration settings.

Mbox	Compile time	Analysis time	Memory
PRADS	0.2	0.25	0.3
Snort	1.5	19	6
HAproxy	0.25	6	6
OpenVPN	0.5	5	7.3

Table 4: Time (h) and memory usage (GB)

8 Other Related Work

Aside from the works discussed in §2 and §4 [9, 16, 18, 22, 24, 25, 26, 28, 31, 33] StateAlyzr is related to a few other efforts. Some prior studies have focused on transforming non-distributed applications into distributed applications [19, 32]. However, these works aim to run different parts of an application at different locations. We want all analysis steps performed by a middlebox instance to run at one location, but we want different instances to run on a different set of inputs without changing the collective output from all instances.

Dobrescu and Argyarki use symbolic execution to verify middlebox code satisfies crash-freedom, bounded-execution, and other safety properties [14]. They employ small, Click-based middleboxes [21] and abstract away accesses to middlebox state. In contrast, our analysis focuses on identifying state needed for correct middlebox operation and works with regular, popular middleboxes.

Lorenzo et al. [13] use similar static program analysis techniques to identify flowspace, but their identification is limited to just hashtables.

9 Summary

Our goal was to aid middlebox developers by identifying state objects that need explicit handling during redistribution operations. In comparison with today's manual and necessarily error-prone techniques, our program analysis based system, StateAlyzr, vastly simplifies this process, and ensures soundness and high precision. Key to StateAlyzr is novel state characterization algorithms that marry standard program analysis tools with middlebox structure and design patterns. StateAlyzr results in nearly 20× reduction in manual effort, and can automatically eliminate nearly 80% of variables in middlebox code for consideration during framework-specific modifications, resulting in dramatic performance and overhead improvements in state reallocation. Ultimately, we would like to fully automate the process of making middlebox code framework-compliant, thus fulfilling the promise of using NFV effectively for middlebox elasticity and fault tolerance. Our work addresses basic challenges in code analysis, a difficult problem on its own which is necessary to solve first.

Acknowledgments

We thank our shepherd, Mona Attariyan, and the anonymous reviewers for their insightful feedback. This work is supported in part by National Science Foundation (grants

CNS-1302041, CNS-1330308 and CNS-1345249) and the Wisconsin Institute on Software-Defined Datacenters of Madison.

References

[1] Codesurfer. http://grammatech.com/research/technologies/codesurfer.

[2] HAProxy: The reliable, high performance TCP/HTTP load balancer. http://haproxy.1wt.eu/.

[3] The LLVM compiler infrastructure. http://llvm.org.

[4] Network functions virtualisation – update white paper. https://portal.etsi.org/nfv/nfv_white_paper2.pdf.

[5] OpenVPN. http://openvpn.net.

[6] Passive Real-time Asset Detection System. http://prads.projects.linpro.no.

[7] Snort. http://snort.org.

[8] Squid. http://squid-cache.org.

[9] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[11] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, 1993.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT*, 1977.

[13] L. De Carli, R. Sommer, and S. Jha. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1378–1390. ACM, 2014.

[14] M. Dobrescu and K. Argyarki. Software dataplane verification. In *NSDI*, 2014.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[16] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *SIGCOMM*, 2014.

[17] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in ec2 and azure. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 177–190. ACM, 2013.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[19] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI*, 1999.

[20] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *HotMiddlebox*, 2015.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18:263–297, 2000.

[22] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI*, 1992.

[23] V. Paxson. Bro: a system for detecting network intruders in real-time. In *USENIX Security (SSYM)*, 1998.

[24] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A high availability framework for middleboxes. In *SoCC*, 2013.

[25] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Escape capsule: Explicit state is robust and scalable. In *HotOS*, 2013.

[26] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[27] T. Reps and G. Rosay. Precise interprocedural chopping. In *ACM SIGSOFT*, 1995.

[28] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL*, 1988.

[29] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[30] J. Sherry, P. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Macciocco, M. Manesh, J. Martins, S. Ratnasamy, and L. R. S. Shenker. Rollback recovery for middleboxes. In *SIGCOMM*, 2015.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[32] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19(1):1:1–1:40, Aug. 2009.

[33] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

Appendix

A. Output-Impacting State - Algorithm

Figure 13 outlines the algorithm for identifying state that impacts packet/log output (from §5.1).

```
Input: sdg, updateableVars
Output: pktoutputVars, logoutputVars
pktoutputVars = {}
logoutputVars = {}
foreach proc in pktProcs do
  foreach stmt in Statements(proc) do
    if stmt calls PKT_OUTPUT_FUNC
      or stmt calls LOG_OUTPUT_FUNC then
      slice = BackwardSlice(sdg, stmt,
                Vars(stmt.RHS))
      foreach sliceStmt in Statements(slice) do
        foreach var in Vars(sliceStmt) do
          if var in updateableVars then
            if stmt calls PKT_OUTPUT_FUNC then
              pktoutputVars = pktoutputVars ∪ {var}
            else
              logoutputVars = logoutputVars ∪ {var}
```

Figure 13: Identifying output-impacting variables

B. Proofs of soundness

We now prove the soundness of our algorithms.

Identifying Per-/Cross-Flow State

Slicing [18] and pointer analysis [9] have already been proven sound.

Theorem 1. *If a middlebox uses standard packet receive functions, then our analysis identifies all packet processing procedures.*

Proof. For a procedure to perform packet processing: (i) there must be a packet to process, and (ii) the procedure must have access to the packet, or access to values derived from the packet. The former is true only after a packet receive function returns. The latter is true only if some variable in a procedure has a data dependency on the received packet. Therefore, a forward slice computed from a packet receive function over the variable containing (a pointer to) the packet will identify all packet processing procedures. □

Theorem 2. *If a value is per-/cross-flow state, then our analysis outputs a top-level variable containing this value, or containing a reference from which the value can be reached (through arbitrarily many dereferences).*

Proof. Assume no top-level variable is identified for a particular per-/cross-flow value. By the definition, a per-/cross-flow must (i) have a lifetime longer than the lifetime of any packet processing procedure, and (ii) be used within some packet processing procedure. For a value to be used within a packet processing procedure, it must be the value of, or be a value reachable from the value of, a variable that is in scope in that procedure. Only global variables and the procedure's local variables will be in scope.

Since we identify statements in packet processing procedures that use global variables, and points-to analysis is sound [9], our analysis must identify a global variable used to access/update the value; this contradicts our assumption.

This leaves the case where a local variable is used to access/update the value. When the procedure returns the variable's value will be destroyed. If the variable's value was the per-/cross-flow value, then the value will be destroyed and cannot have a lifetime beyond the packet processing procedure; this is a contradiction. If the variable's value was a reference through which the per-/cross-flow value could be reached, then this reference will be destroyed when the procedure returns. Assuming a value's lifetime ends when there are no longer any references to it, the only way for the per-/cross-flow value to have a lifetime beyond any packet processing procedure is for it be reached through another reference. The only such reference that can exist is through a top-level variable. Since points-to analysis is sound [9] this variable would have been identified, which contradicts our assumption. □

Identifying Updateable State

Theorem 3. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may be updated during the lifetime of some packet processing procedure, then our analysis marks this top-level variable as updateable.*

Proof. According to the language semantics, scalar and compound values can only be updated via assignment statements. According to Theorem 1, we identify all packet processing procedures. Therefore, identifying all assignment statements in these procedures is sufficient to identify all possible value updates that may occur during the lifetime of some packet processing procedure.

The language semantics also state that the variable on the left-hand-side of an assignment is the variable whose value is updated. Thus, when a top-level variable appears on the left-hand-side of an assignment, we know its value, or a reachable value, is updated. Furthermore, flow-insensitive context-insensitive pointer alias is provably guaranteed to identify all possible points-to relationships [9]. Therefore, any assignment to a variable that may point to a value also pointed to (indirectly) by a top-level variable is identified, and the top-level variable marked updateable. □

Identifying Flowspaces

Theorem 4. *If a middlebox uses standard patterns for fetching values from data structures, and the flowspace for a top-level variable's value (or a value reachable through arbitrarily many dereferences starting from this value) is not constrained by a particular header field, then our analysis does not include this header field in the flowspace fields for this top-level variable.*

Proof. A header field can only be part of a value's flowspace definition if there is a data or control dependency between that header field in the current packet and the fetching of an entry from a data structure. It follows from the proven soundness and precision of flow-sensitive context-insensitive pointer analysis [11] that the SDG will not include false data or control dependency edges. It also follows from the proven soundness of program slicing [18] that only data and control dependencies between source variables (i.e., the packet variable) and target variables (i.e., the index variable, increment variable, or variable in a conditional inside a loop) will be included in the chop. □

Identifying Output-Impacting State

Theorem 5. *If a top-level variable's value, or a value reachable through arbitrarily many dereferences starting from this value, may affect a call to a packet output function or the output produced by the function, then our analysis marks this top-level variable as impacting packet output.*

Proof. Follows from SDG construction soundness [15, 18]. If/when a packet output function is called is determined by a sequence of conditional statements. The path taken at each conditional depends on the values used in the condition. Control and data dependency edges in a system dependence graph capture these features. Since SDG construction is sound [15, 18], we will identify all such dependencies, and thus all values that may affect a call to a packet output function.

Only parameter values, or values reachable through arbitrarily many dereferences starting from these values, can affect the output produced by a packet output function. Thus, knowing what values a parameter value depends on is sufficient to know what values affect the output produced by an output function. Again, since SDG construction is sound, we will identify all such dependencies. □