StateAlyzr: Deep Diving into Middlebox State to Enable Distributed Processing

Junaid Khalid, Aaron Gember-Jacobson, Roney Michael,
Anubhavnidhi Abhashkumar, Aditya Akella

University of Wisconsin-Madison

Abstract

We consider the problem of modifying network middleboxes to enable live state redistribution. The need for this arises when an input workload is redistributed across middlebox instances in important scenarios such as elastic scale in/out, high availability, and load balancing. While techniques exist today for safe migration/cloning of live state, the task of modifying middlebox code to identify needed state is manual, and hence extremely complex and error prone. We present a system, StateAlyzr, that embodies a novel set of algorithms adapted from program analysis techniques to provably and automatically identify all state that must be migrated/cloned to ensure consistent middlebox output in the face of dynamic redistribution. StateAlyzr leverages middlebox code structure and common design patterns to simplify analysis and to minimize migrating/cloning unneeded state. We apply StateAlyzr to four open source middleboxes. We find that a large amount of live state matters toward packet processing in these middleboxes. We build upon the output of StateAlyzr to develop a highly-available version of one of the middleboxes. We find that StateAlyzr's algorithms can reduce the amount of state that needs to be transferred across live and hot standby instances by up to 600×.

1. Introduction

Recent advances in virtualization and software-defined systems have made it markedly easier to operate dynamically scalable and highly available applications. Virtual machines (VMs) and containers can both be easily replicated [4, 12, 14] for scale out or high availability, and application traffic can be easily rerouted using consistent hashing or software-defined networking (SDN). However, the ability to redistribute application tasks at a fine granularity among instances (e.g., reallocating the handling of a client request)—to achieve better performance, lower operating costs, or higher availability—is severely limited today.

Such redistribution is complicated by the presence of *live application state*. This state is dynamically updated as the application processes each request, or even each packet, and the state's current value determines the actions the application will take on the input workload. Thus, blindly reallocating tasks without explicitly handling application state can result in erroneous processing, because the relevant state may be unavailable at the task's new location.

To this end, recent proposals have argued for explicit, fine-grain handling of application state. Escape Capsule [24], Split/Merge [25], Pico Replication [23], and OpenNF [16] have provided frameworks for safely and efficiently transferring application state between instances to ensure up-to-date state is available when and where it's needed. This can potentially open the door for cost-effective application scaling, better control of application performance, faster and more efficient failover, and other new services [16].

However, application designers are still left with an enormous task: manually modifying their existing software, or rebuilding from scratch, to provide the necessary support for external state control and cross-instance coordination. Three factors make this is a daunting task: (i) application software may be extremely complex; (ii) there may be tens if not hundreds of object types that correspond to state that needs explicit handling; and (iii) applications are extremely diverse. Factors i and ii make it nearly impossible to reason about the completeness or soundness of modifications made. And, iii means manual techniques that apply to one application don't necessarily extend to another.

The ultimate goal of our work is to make the above code modification process automatic, systematic, and general. The work presented in this paper considers a version of this problem that is narrow in two respects: First, we focus a majority of our discussion on *middleboxes*, a special class of in-network applications that perform stateful, custom packet processing. Examples of middleboxes include Web proxies, firewalls, load balancers, WAN accelerators, application gateways, and intrusion detection systems (IDSs). Second, instead of focusing on fully automated code modification, we focus on addressing the basic research challenges in *code analysis* that are the precursor to automated modification.

More precisely, we aim to *significantly ease* the task of modifying *arbitrary* middlebox code to ensure state transfer across instances provably maintains the *output equiva-*

lence property [25], which states that the aggregate output produced by a collection of instances following redistribution should be equivalent to the output that would have been produced had redistribution not occurred. To ensure redistribution offers good performance and incurs low overhead, we seek the *minimal* such state needed.

In Section 2, we argue that this entails solving four subproblems: (i) identifying critical state objects, (ii) determining if state is read-only or write-able, (iii) determining how state impacts output, and (iv) identifying which subset of input workload (i.e., the "flow space") can cause reads/updates to some state. While the first matters toward output equivalence, the latter three impact redistribution efficiency.

Unfortunately, middlebox state only exists at runtime. If we choose to identify critical state and its properties at runtime, middleboxes may run 13× slower. Therefore, our system, StateAlyzr, relies on *static program analysis* to identify the possible existence and properties of critical state *before* it comes into existence. StateAlyzr's algorithms are based on a synthesis of escape analysis [21, 26], pointer analysis [9, 27], and program slicing [17, 29] techniques, modified to leverage typical middlebox code structure and design patterns. We prove that the algorithms ensure output equivalence, and show empirically that they are effective in improving state transfer efficiency.

We run StateAlyzr on four production quality, open source middleboxes: Passive Real-time Asset Detection System (PRADS) [6], HAProxy load balancer [3], Snort IDS [7], and OpenVPN gateway [5]. We find that the amount of live state that matters toward packet processing can be large—captured by as many as 29-300 variables across the four middleboxes. A fraction of these (33-60%) represent updateable state; of these, a subset (60-88%) impact packet/log output. We also show that StateAlyzr's output for PRADS aligns with, and, crucially, improves on, the manual modifications the authors of OpenNF [16] made to this middlebox. Finally, we build upon the output of StateAlyzr to transform a stand-alone version of PRADS into a highly available distributed version. Compared to naively cloning all updateable state to a hot standby PRADS instance, efficient instrumentation based on StateAlyzr's output can reduce the amount of state transferred by 300× by focusing on relevant flow-spaces, and by a further 2× by monitoring the precise state that got updated; furthermore, the run time overhead of the instrumentation is just 0.14%.

Even though we focus on middleboxes, we believe that the high-level ideas—leveraging common code structure and design patterns across applications of a given class—can be employed toward similar static analysis-driven code modification of other generic applications.

2. Background and Motivation

We start by describing the structure of middlebox software and the nature of middlebox state maintain. We then describe

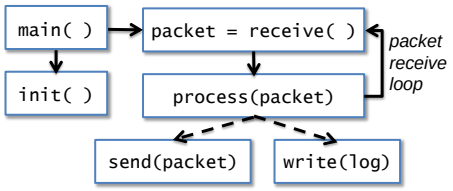


Figure 1: Logical parts of middlebox code

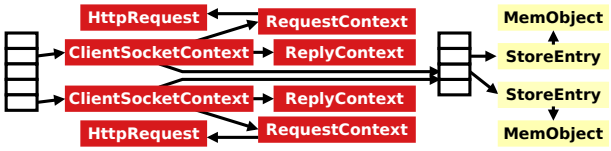


Figure 2: Internal state for the Squid caching proxy: each shaded box is a different piece of state; red (dark) state is maintained for each client connection; yellow (light) state is maintained for each cached web object; both are organized using hash tables.

the notion of output equivalence that is central to any scenario needing dynamic redistribution of processing across middlebox instances; we describe prior works to ensure output equivalence and the challenges they pose to middlebox application developers. We conclude with a discussion of requirements for any framework aiming to reduce application developers' effort in this regard.

2.1 Middlebox Structure

Irrespective of the specific function of a middlebox, its code can be logically divided into five basic parts (Figure 1): initialization, packet receive loop, packet processing, packet write, and log write. 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. The procedure(s) reads/writes internal middlebox state to inform the processing of the current (and future) packet. Lastly, the packet write and log write functions, both of which are optional, 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).

Performing sophisticated packet processing requires middleboxes to maintain detailed internal state at run time. Figure 2 shows a subset of the state maintained by a Squid caching proxy [8]. Each piece of state pertains to a specific connection[1], session, application, host, subnet, URL, or other network unit, and is organized using hash tables, lists,

[1] Defined using the traditional 5-tuple: source IP, destination IP, protocol, source port, and destination port.

trees, or other data structures that facilitate easy lookups based on packet fields. A middlebox's state is highly coupled with the traffic it receives—every packet may trigger reads and updates to multiple pieces of state—and a middlebox's actions are highly coupled with its current state.

As a result, rerouting traffic between middlebox instances—e.g., for high availability or load balancing—can compromise a middlebox's effectiveness, and potentially break end-to-end connectivity. For example, if a client's traffic is rerouted to a new caching proxy after the client has already transmitted part of an HTTP request, the new proxy won't be able to reconstruct the full request, and the client's request will fail.

2.2 Output Equivalence

As the above example shows, maintaining effective middlebox operation following traffic rerouting requires cloning or transferring *critical* middlebox state across instances. How we define "critical state" depends on the scenario. To avoid a failed client request in our example above, we must move the ClientSocketContext, RequestContext, ReplyContext, and HttpRequest objects associated with the client's connection to the new proxy instance. If we are instead in the midst of sending a reply to the client, and we want to avoid an incomplete file transfer, we must also copy the StoreEntry and MemObject associated with the URL the client requested.

More generally, we want to move or copy the middlebox state required for *output equivalence*: i.e., the aggregate output produced by a collection of middlebox instances following traffic rerouting should be equivalent to the aggregate output that would have been produced had the rerouting not occurred [25].[2]

Frameworks. Fortunately, frameworks like Split/Merge [25], Pico Replication [23], and OpenNF [16] have made great strides in safely and efficiently copying and moving middlebox state to achieve output equivalence. These frameworks enable an external controller to `get()` and `put()` the needed internal state, while relying on mechanisms such as events or counters (which indicate packet arrivals during `get()` or `put()`) to guarantee that the state transfer/clone satisfies certain safety and liveness properties (e.g., loss-free, order-preserving or strongly consistent).

Challenges they pose. These frameworks are based on two non-trivial assumptions: (i) what state is critical is known prior to run time, and (ii) the middlebox has been exhaustively instrumented to "leave no critical state behind". For these to hold, developers need to have intimate familiarity with the various types of state a middlebox maintains, and an understanding of which state affects a middlebox's output and how. This is not easy: as shown in Table 1, several popular middleboxes have between 60K and 275K lines of code, dozens of different structures and classes, and complex event-based control flow.

[2] In some cases, we may accept a relaxed notion of output equivalence: e.g., the proxy's interactions with clients must be equivalent, but we are willing to tolerate more cache misses, and corresponding requests to remote servers.

Middlebox	LOC (C/C++)	Event based?
Snort IDS [7]	275K	No
HAProxy load balancer [3]	63K	No
OpenVPN [5]	62K	No
PRADS asset detector [6]	10K	No
Bro IDS [22]	97K	No
Squid caching proxy [8]	166K	Yes

Table 1: Code size for popular middleboxes. The ones above the line are analyzed in greater detail later.

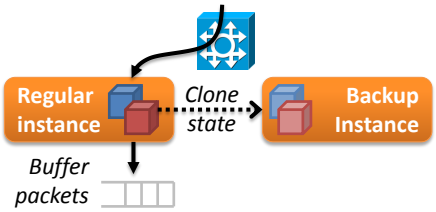


Figure 3: Steps to enable fast, transparent failover

Output equivalence vs. efficiency. We aim to design a framework that enables a developer to (semi) automatically instrument middlebox software to transfer/clone the *minimal state needed to ensure output equivalence*. Transferring all state ensures output equivalence, but adds significant overhead and latency due to the inclusion of unneeded state.

In what follows, we break this down into four requirements using the example of middlebox modifications necessary for *fast, transparent failover* to ensure high availability. We stress that the same requirements apply to modifying middleboxes to support dynamic redistribution in other contexts, such as, load balancing and scaling.

2.3 Middlebox Modifications Required

Meeting strict uptime guarantees, such as the five-nines availability demanded by service providers, requires fast, transparent failover of middlebox instances. To achieve this, we must occasionally clone a middlebox's internal state, either to persistent storage or to another middlebox instance (Figure 3). Packets output by the middlebox cannot be released into the network until the state has been successfully cloned [23]. To support such cloning, a developer needs to know *what middlebox state is critical?*

A naive failover system could clone all middlebox state (assuming the cloning process itself satisfied certain safety guarantees [16, 23]). If we assume failure recovery always happens at packet boundaries—i.e., we resume processing at the start of the next packet, not in the middle of processing a packet—then we can slightly constrain our answer: we only need to *clone middlebox state used during the processing of more than one packet* (R1).

However, simply exporting all state used in the processing of multiple packets causes duplicate and unnecessary

state cloning, impacting speed and efficiency. First, some middlebox state may only be read, never written, during packet processing. For example, the Squid caching proxy parses a configuration file on startup, and it stores the settings in memory for fast access when processing packets. Similarly, the Snort IDS parses signature files on startup, and compiles the signatures into regular expressions for fast pattern matching when processing packets. Such read-only states only need to be exported and cloned once—more than once just adds unnecessary overhead (Section 6.3). Thus, an analysis framework should *indicate which state is read but never updated during packet processing* (R2).

Second, operators will likely tolerate some deviation in certain forms of middlebox output. For example, the packets produced by a caching proxy following failover should obey output equivalence, but output equivalence of a proxy's log of requests and cache misses is probably not necessary. A middlebox developer can reduce cloning overhead by choosing to exporting only the state that affects packet output. Thus, an analysis framework should *indicate whether specific state affects packet output and/or log output* (R3).

When an instance fails, we want to avoid overloading the fallback instance(s), otherwise the network may experience cascaded failures. We can minimize the likelihood of overload by spreading traffic from the failed instance among multiple instances [23]. With such load distribution, every fallback instance does not need the same set of middlebox state; each fallback instance only needs the state that may be used during the processing of the traffic it will receive following failover. For example, the state the Squid caching proxy maintains on a per-connection basis (the red/dark state in Figure 2), only needs to be cloned to the fallback instance responsible for a particular connection. To facilitate such filtered exporting, an analysis framework should *identify which traffic will cause read/updates to specific state* (R4).

3. StateAlyzr Foundations

We present the basic components of our system, StateAlyzr, that help meet the requirements R1–4 above. StateAlyzr leverages middlebox code structure and common design patterns to combine and adapt different techniques from static analysis to provably guarantee soundness in meeting the above requirements, while also ensuring usable precision.

3.1 Critical Middlebox State

Formally, middlebox state is a collection of scalar and compound values. These values are stored in the stack, heap, or data segments of a program's memory space. For example, a count of the number of received packets may be an integer value stored on the stack, while a compound value (i.e., an object) of type `ClientSocketContext` class may be created on the heap for each new TCP connection a middlebox sees. A value may also be a reference to (i.e., the memory addresses of) another value: e.g., an object of type `ClientSocketContext` may point to an object of type `RequestContext`. A *value's lifetime* is the duration for which its storage location is allocated.

Critical middlebox state is the set of values that (i) have a lifetime longer than the lifetime of any packet processing procedure, and (ii) are used within some packet processing procedure. As discussed in Section 2, achieving output equivalence in the face of dynamic redistribution requires identifying and transferring/cloning such critical state.

3.1.1 Run Time Analysis

Unfortunately, the precise lifetime of many values is not known until run time, because middleboxes tend to dynamically allocate values as packets are processed. To determine the precise lifetime of these values, we must track all memory allocations and deallocations at run time. Furthermore, to determine if a value is reachable from within a packet processing procedure, we must track the creation (and destruction) of references to the value. Alternatively, we can track all memory accesses and updates that occur while a packet processing procedure is executed; if part of a value's storage location is accessed or updated during this time, then we know the value must be reachable, and we must mark the value as critical state. Running this analysis offline is not an option, because the observed creation, destruction, reading, and writing of values depends on the input packets.

While an online analysis is simple in principle, the computational overhead can cripple a middlebox's performance. We used Dyninst [2] to augment the PRADS asset detection system [6] with code that tracks all memory (de)allocations, as well as all memory reads/writes during packet processing. Our modified middlebox ran 13× slower!

Fortunately, exact measures of object lifetime and reachability are not essential to identifying critical middlebox state. While this information improves *precision*—we want few non-critical values to be identified as critical—our first-order concern is *soundness—all* critical values must be identified.

3.1.2 Escape Analysis

Escape analysis is one way to identify dynamically allocated values whose lifetimes exceed the lifetime of any packet processing procedure. This analysis identifies references that "escape" the scope of a procedure—through a return value, a global variable, or a reference value provided as a parameter—thus allowing the referenced values, and any values reachable through arbitrarily many dereferences, to be used outside of the procedure [21, 26].

Escape analysis works as follows [26]: Within each procedure, identify expressions that dynamically create new values (e.g., calls to `malloc()`). These expressions are called *resolved sources*, because they bring values into existence at run time; the expressions also serve as static representations of values [18]. Next, perform intra-procedural data-flow analysis to determine which of the resolved sources may be returned by the procedure. Lastly, perform inter-

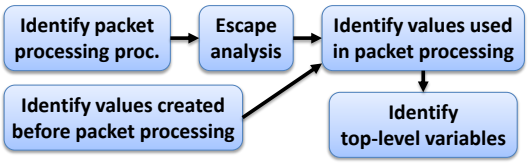


Figure 4: Steps to identify critical middlebox state

procedural data-flow analysis to determine which resolved sources may be passed between procedures, and thus may be returned by a procedure earlier in the call graph.

Escape analysis is just a part of what we need to identify all critical middlebox state. We also need analyses that identify: (*i*) all packet processing procedures, (*ii*) values whose lifetime begins prior to packet processing, and (*iii*) values which are actually used in packet processing procedures. We describe each of these analyses below. The complete set of steps is shown in Figure 4.

3.1.3 Identifying Packet Processing Procedures

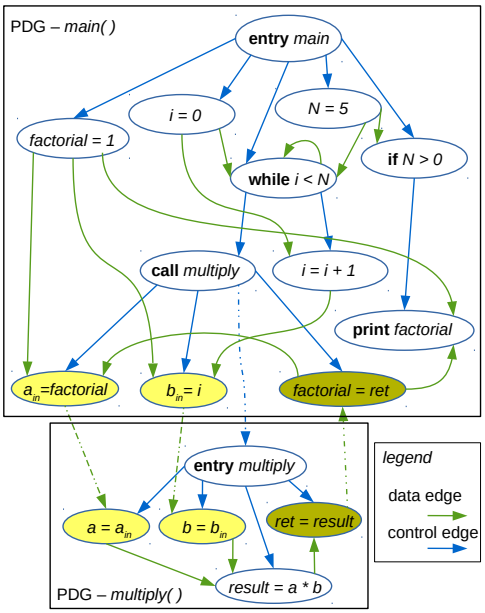
To identify values that escape from packet processing procedures, we need to know which procedures fall into this category. Our analysis to identify these procedures is based on a key property of middleboxes: *all packet processing starts with receiving a packet*. Thus, any procedure whose invocation depends on the result of a packet receive function may be a packet processing procedure. Most middleboxes use standard library/system functions to receive packets—e.g., `pcap_loop`, or `recv`—so we can easily identify these calls and the variable pointing to the received packet.

We use *program slicing* [17] to identify procedures whose invocation depends on the received packet. A *forward slice* is the set of *statements that are affected* by the values of a set of variables starting from a specific point in the program. A *backward slice* is the set of *statements that do affect* the values of variables at a specific point in the program.

Slices are computed using a *system dependence graph* (SDG), which captures the control and data dependencies between procedures in a program. An SDG consists of multiple *program dependence graphs* (PDGs)—one for each procedure. Each PDG contains vertices for each statement (i.e., program point) in the procedure, along with data and control dependence edges between those statements. A *data dependence* edge from program point p to program point q is created if there is an execution path between p and q, and, for some variable v, p may update v's value (or a value reachable through arbitrarily many dereferences), and q may read v's value (or a value reachable through arbitrarily many dereferences). A *control dependence* edge from p to q is created if p is a conditional statement, and whether or not q executes depends on p. In the SDG, interprocedural control dependence edges connect procedure call sites with the entry point of the called procedure, and interprocedural data dependence edges represent the flow of the data between a procedure's input

```
1  void main() {
2    int N = 5;
3    int factorial = 1;
4    int i = 0;
5    while(i <= N) {
6      i = i + 1;
7      factorial = multiply(factorial,i);
8    }
9    if (N > 0) {
10     printf(factorial);
11 } }
12 int multiply(int a, int b) {
13   return a*b;
14 }
```

(a) Program to compute factorial; statements in red are part of the backward data dependence slice for the variable `factorial` at line 10



(b) System dependence graph (SDG) for the factorial program; 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

Figure 5: Slicing example

parameter(s) and return value. Figure 5b shows the SDG and PDGs for the example program shown in Figure 5a.

Given a middlebox's SDG, we can compute a forward slice from a packet receive function call for the variable which stores the received packet. We consider any procedure appearing in the slice to be a packet processing procedure. We compute forward slices from every packet receive function call site and take the union of the procedures appearing within all such forward slices.

We originally considered using calls graph to identify all packet processing procedures. The analysis is simple: construct a call graph starting from each procedure called from within the packet receive loop; any procedure appearing in one of these call graphs is a packet processing procedure. However, this approach does not capture packet processing procedures that are called indirectly. For example, the Squid caching proxy does some initial processing of data received from a socket, then puts an event on a queue to trigger more

processing of the input through later calls to additional procedures. Our slice based approach is able to capture these indirect calls, because the SDG contains a data dependence edge from the statement that creates the event to the statement that calls a later procedure based on the event value(s).

3.1.4 Values Created Before Packet Processing

While escape analysis can be applied to the identified packet processing procedures to construct a list of dynamically allocated values that escape these procedures, this analysis will not identify critical values whose lifetime begins prior to packet processing. This includes values created: (i) when the program starts—this is the case for the values of global and static variables; or (ii) after the program starts but before any packet processing procedure is invoked—this is the case for values dynamically allocated in middlebox initialization procedures, or values of local variables declared in procedures earlier in the call stack (i.e., before the invocation of any packet processing procedure).

Global and static variable declarations can serve as static representations for values of type i. Identifying values of type ii requires us to apply the techniques from Sections 3.1.2 and 3.1.3 in a slightly different way: identify all procedures called prior to packet processing by computing a forward slice from `main`, and identify all values that escape these procedures using escape analysis.

We have now identified all values that satisfy the first criteria for critical middlebox state: *values with a lifetime longer than the lifetime of any packet processing procedure*.

3.1.5 Values Used in Packet Processing Procedures

Next, we need to narrow the above set of values to those that also satisfy the second criterion: *values used within some packet processing procedure*. For a value v to be used within some packet processing procedure, there must be a statement within a packet processing procedure that contains a variable whose value is v, or a variable whose value can be used to reach v through arbitrarily many pointer dereferences.

In the simplest case, a statement contains one of the global or static variables identified in our first phase of analysis. If any such statement occurs within any packet processing procedure, then we definitively know that the value of that global or static variable, or a value reachable through arbitrarily many dereferences, is critical middlebox state.

The more challenging case is when a statement contains a non-static local variable, including formal parameters. If the variable is a pointer, then it's possible the variable points to critical middlebox state. (Non-pointer local variables can be ignored.) Standard Andersen or Steengsaard pointer analysis [9, 27] can provide this information. Andersen's algorithm (the flow-insensitive and context-insensitive version) works as follows: Create an *abstract value node* for each global variable, local variable, and dynamic memory allocation expression in the entire program. Start with an empty points-to set for each variable in the program. For each assignment statement, apply a pre-defined subset constraint, selected based on the form of the assignment, to update the points-to set for the variable on the left-hand-side of the assignment. For example, for the statement `y = &x`, the points-to set for y is updated to contain the abstract value node for x in order to satisfy the subset constraint $y \supseteq x$. Continue to update points-to sets until no sets change further.

After computing the points-to sets for all local variables in all packet processing procedures, we compute the intersection between the points-to set for each variable and the set of values with a lifetime longer than the lifetime of any packet processing procedure. Any values that intersect are marked as critical middlebox state.

3.1.6 Critical Top-Level Variables

While we have achieved the goal laid out at the beginning of this section—identify all critical middlebox state—we need to provide more information to enable a middlebox to be modified to export/import these values. In particular, we need to provide a "handle" that can be used in code to access the critical values identified by our analyses.[3] These can be identified by first computing a points-to set for each global, static, and local variable that is in scope immediately outside the packet receive loop (Figure 1), and then computing the intersection of each variable's points-to set with the set of critical values identified by our analysis. If the intersection is non-null, then we mark that variable as a *critical top-level variable*, and it then becomes a handle.

3.1.7 Soundness

We now prove the soundness of our analyses. Escape analysis [26], slicing [17], 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 critical middlebox state, then our analysis outputs a critical top-level variable containing this value, or containing a reference from which the value can be reached (through arbitrarily many dereferences).*

[3] The expressions output by escape analysis are not appropriate handles, because they simply indicate how a value came into existence, not how the value can be accessed outside of the procedure in which it was allocated.

Proof. Assume no critical top-level variable is identified for a particular critical value. By the definition, a critical value 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 critical 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 critical 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 critical 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 critical top-level variable. Since points-to analysis is sound [9] this variable would have been identified, which contradicts our assumption. □

A list of top-level variables is insufficient information to optimize state transfers; simply transferring all identified variables can lead to unnecessary overhead and hurt performance. In Sections 3.2–3.4, we present a set of techniques to progressively improve the precision of state transfer/cloning, without impacting output equivalence.

3.2 Updateable Middlebox State

Knowing whether a value is updated during packet processing can aid in minimizing the size of future state transfers. Static analysis can give us an approximate answer as to whether a value will be updated at run time.

To ensure we don't miss transferring some updated state necessary for output equivalence, *updateable state must never be marked read-only*; Section 3.2.2 proves our analysis is sound in this regard. We can tolerate read-only state being marked as updateable; this degrades efficiency but doesn't impact output equivalence.

3.2.1 Analysis Details

Identifying state updates in packet processing procedures is similar to identifying uses of state in these procedures (Section 3.1.5). The key difference is that we only consider assignment statements. If an assignment is made to one of the top-level variables identified in our first phase of analysis, then we know the value of that variable may be updated during packet processing, and the variable should be marked as updateable.

Otherwise, we need to determine if the variable is a pointer,[4] and perform pointer analysis to identify all values reachable from the variable. We compute the intersection of the points-to set for the variable on the left-hand-side of the assignment statement and the points-to set for each top-level variable. If the intersection is non-null, then we mark the top-level variable as updateable.

After considering all assignment statements in packet processing procedures, any top-level variables not marked as updateable are marked read-only.

3.2.2 Soundness

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. □

3.3 State Impacting Different Types of Output

As mentioned in Section 2.3, forgoing output equivalence for some forms of output (e.g., logs) in exchange for smaller/less frequent state transfers/clones may be an important optimization in some scenarios. To provide this flexibility, we need to know *whether a particular value affects packet output, log output, or both*.[5] Similar to before, we must not under-estimate which state affects packet output, or log output, to ensure soundness. And, we want to avoid overestimating as much as possible to maximize precision.

We achieve this using two key insights. First, middleboxes typically use *standard libraries and system calls* to

[4] Assignments to non-pointer non-top-level variables can be ignored since the variable's value will be destroyed when the procedure returns.

[5] Values which do not affect either form of output are unnecessary for the middlebox to maintain. Our analysis identifies these values, so statements creating, reading, or updating these values can be pruned from the code.

produce packet and log output: either PCAP (`pcap_dump`, `pcap_inject`, etc.) or socket (`send`, `sendto`, etc.) functions for the former, and regular I/O functions (`write`, `printf`, etc.) for the latter.[6] Second, the output produced by these functions can only be impacted by a *handful of parameters* passed to these functions. Next, we show how to leverage these insights together with program slicing.

3.3.1 Analysis Details

We start by identifying all statements that call library or system functions that produce packets or log output. Then, we compute a *backward slice* from each call site to determine what statements affect (i) whether the call occurs, and (ii) the output the function produces. Since the output produced by a packet (or log) output function depends on the values of the actual parameters, or values reachable through arbitrarily many dereferences starting from these values, we can find all relevant program points by setting the variables for which a backward slice is being computed to those in the parameters of the call to the output function.

For all statements in a backward slice, we determine whether any variable in a given statement (i) is a top-level variable, or (ii) is a pointer to a value of a top-level variable, or a value reachable through arbitrary many dereferences starting from the value of a top-level variable. We mark the corresponding top-level variable as impacting packet (or log) output if at least one of these conditions is true for at least one statement in the backward slice. We compute backward slices from all packet (or log) output function call sites, and repeat this process for each slice.

The result of this analysis is three lists of top-level variables: those that may impact packet output, those that may impact log output, and those that may impact both types of output. If output equivalence is only required for packets, or logs, then only the values of those top-level variables (or values reachable through arbitrarily many dereferences starting from those values) must be transferred or cloned.

3.3.2 Soundness

Theorem 4. *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, 17].[7] □

The same theorem and proof applies to log output as well.

3.4 Traffic Impacting Middlebox State

Another opportunity to optimize state transfers/clones arises from the fact that specific values are only accessed/updated when processing specific traffic (Section 2.3). To implement this optimization without compromising output equivalence, we need to know: *for each value, what are all possible packets that will trigger a read or update to that value*.

We can define a set of possible packets in terms of a *flow space*—a set of tuples each consisting of a packet header field (e.g., EtherType, protocol, and source and destination IPs and ports) and a value range (e.g., a subnet address, port number, or wildcard). However, we can only determine the values of the tuples at run time, due to dynamic allocation of values and multiple possible execution paths in packet processing procedures.

To overcome this challenge, StateAlyzr leverages common patterns in middlebox code, discussed next. It is important that our analysis does not identify more header fields than those that actually define the flow space for a value, otherwise we may inadvertently assume the value has a finer-grained flow space and incorrectly skip transferring the value at run time. In Section 3.4.3, we prove our analysis is sound in this regard. In Section 6, we show that our analysis is precise (i.e., not identifying header fields that are part of the definition of a value's flow space is rare).

3.4.1 Organization of Middlebox State

We leverage a common design pattern: When a middlebox needs to maintain state of the same type for different connections, applications, subnets, URLs, etc., it typically uses a simple data structure (e.g., hash table or linked list) to keep the state organized. When processing a packet, the middlebox uses packet header fields to lookup the entry in the data structure that contains a reference to the value(s) that should be read/updated for this packet. In the case of a hash table, the middlebox computes a hash table *index* from the packet header fields to identify the entry pointing to the relevant value(s). For a linked list, the middlebox *iterates* over entries in the data structure and compares packet header fields

[6] If middleboxes use non-standard output functions, our analysis can easily be extended to consider these functions.

[7] In more detail: 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, 17], 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.

against the value(s) pointed to by the entry to determine which entry points to the relevant value(s).

It is possible a middlebox has just one value of a particular type (e.g., a count of TCP packets) that is only accessed/updated when processing particular packets. Access/update of such values depends on a conditional statement that checks the value(s) of some field(s) in the packet.

3.4.2 Analysis Details

We leverage the above design pattern to develop a heuristic for statically identifying the header fields that define the flow space for a value.

We assume middleboxes use hash tables or linked lists to organize their values,[8] and we assume entries in these data structures will be accessed using: square brackets, e.g.

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

pointer arithmetic, e.g.

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

iteration, e.g.

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

or recursion. The first step of our analysis is thus to identify all statements like these where a top-level variable is on the right-hand-side or in the conditional expression.

When square brackets or pointer arithmetic are used, we compute a *chop* between the variables in this statement and the variable containing the packet returned by the packet receive procedure. Chopping combines backward and forward slicing. A chop between a set of variables U at program point p and a set of variables V at program point q is the subset of all points 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.

When iteration is used, we identify all conditional statements in the body of the loop. For each of these conditional statements, we compute a chop between the packet returned by the packet receive procedure and the variables in the conditional expression. 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.

3.4.3 Soundness

Theorem 5. *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 [17] 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. □

[8] Our analysis can easily be extended to other data structures.

4. Enhancements

The analyses discussed in Sections 3.2, 3.3, and 3.4 all provide an opportunity to clone state more efficiently: If critical state is read-only, then it only needs to be cloned once. If output equivalence of logs is unnecessary, then only state that impacts packet output needs to be cloned. If traffic will be distributed among multiple middlebox instances in the case of failure, then only state whose flow space overlaps with the flow space assigned to a specific instance needs to be cloned to that instance.

However, the potential performance gains from these optimizations are limited by the precision achievable with static analysis. For example, static analysis can only identify whether a top-level variable's value, or value(s) reachable through arbitrarily many dereferences, *may* be updated at *some time* during the middlebox's execution; we cannot determine exactly which values are updated, and when.

To achieve higher precision, we must use (simple) run time monitoring. For example, we can track, at run time, whether part of a compound value is updated during packet processing, allowing us to know exactly what state we need to clone to achieve transparent failover.

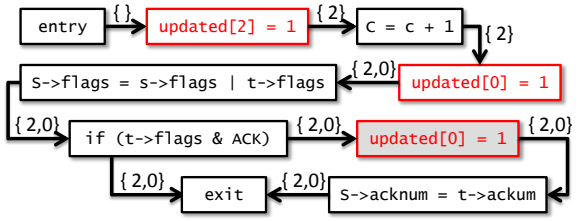
To implement this monitoring, we must modify the middlebox to set an "updated bit" whenever a value reachable from a critical top-level variable is updated during packet processing. Figure 6a shows such modifications, in red, for a simple middlebox. We create a unique updated bit for each critical top-level 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 top-level variable.

We use the same analysis discussed in Section 3.2 to determine where to insert statements to set updated bits. In particular, we find all assignment statements in packet processing procedures, and we compute the intersection between the points-to sets for the variable on the left-hand-side of the assignment statement and the points-to set for each top-level variable. For any top-level variable for which the intersection is non-null, we insert a statement—just prior to the assignment statement—that sets the appropriate updated bit.

While inserting code before every update statement guarantees we always set the appropriate updated bits, this adds a lot more code than necessary: if one assignment statement

```
1  struct conn tbl[1000]; // Assigned id 0
2  int count; // Assigned id 1
3  int tcpcount; // 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       handleTcp(&tcpcount, &tbl[hash(pkt)], getTcpHdr(pkt));
13 } } }
14 void handleTcp(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 6: Implementing update tracking at run time

always executes before another assignment statement, and they *always* update the same value, then we only need to add code before the first assignment statement to set the updated bit. For example, line 21 in Figure 6a updates the same compound value as line 18, so the code on line 20 is redundant and unnecessary.

We use a straightforward control flow analysis to prune unneeded updated-bit-setting statements. First, we construct a control flow graph for each modified packet processing procedure. Next, we perform a depth-first traversal of each control flow graph, 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 6b shows this annotated control flow graph for the `handleTcp` procedure shown in lines 14-22 of Figure 6a. Lastly, for each updated-bit-setting statement in a procedure's control flow graph, 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. For example, we prune line 20 in Figure 6a.

5. Implementation

Analysis. We implement StateAlyzr using CodeSurfer [1]. CodeSurfer has built-in support for constructing control flow graphs, performing Andersen's flow-insensitive and context-insensitive pointer analysis [9], constructing program and system dependence graphs, and computing forward and backward slices and chops for C/C++ code. We use CodeSurfer's Scheme API to access the output from these analyses and perform additional analysis necessary to produce the appropriate output.

High Availability. We use the output from StateAlyzr to add high availability support to PRADS. PRADS is an off-path monitoring middlebox that identifies and logs basic information about active hosts in the network (e.g., operating system, uptime, etc.) and the services running on them (e.g., application name, version, etc.). PRADS maintains per-connection and per-host state, using two hash tables to organize the compound values of each type. It also stores statistics and configuration settings in a global compound value.

We added code to PRADS to export/import the aforementioned critical state. We used the output of our first analysis phase (Section 3.1) to know which top-level variables' values we needed to export, and where in a hot-standby we should store them. We used the output of our fourth analysis phase (Section 3.4) as the basis for code that looks up per-connection and per-host values in the top-level hash tables. 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. Import code, again written on the basis of the output from our first and fourth phases of analysis, deserializes the state and stores it in the appropriate location. We also implemented the enhancement discussed in Section 4 to track updates to write-able variables. We added statements at 44 points in the code to mark updated bits. Note: PRADS does not produce/impact packet output.

6. Evaluation

We report on the outcomes of applying StateAlyzr to four popular open source middleboxes: PRADS [6], Snort IDS [7], HAproxy load balancer [3], and OpenVPN [5]. All four middleboxes are written in C. We address the following sets of general questions: (i) How much critical state do these middleboxes maintain? And, what relative fractions of this state are updateable, and packet or log output-impacting? (ii) How efficient is StateAlyzr? (iii) How does StateAlyzr compare against other candidate approaches for identifying critical state? Additionally, in the context of a highly available PRADS implementation, we address the following questions: (iv) Does StateAlyzr ensure output equivalence? (v) To what extent do StateAlyzr's mechanisms for identifying a state object's key space and whether a particular piece of state was updated by a packet help?

6.1 Critical State and its Properties

Table 2 shows the number of critical top-level variables identified by StateAlyzr in the four open source middleboxes mentioned above. Snort is the most complex middlebox we analyze (≈275K lines of code) and has the largest number of critical top-level variables (333); the opposite is true for

Mbox	Top-level vars	Updateable vars	Vars impacting packet output
PRADS	29	10	N.A.
Snort	333	148	N.A.
HAproxy	168	107	91
OpenVPN	126	101	89

Table 2: Critical top-level variables and their properties

Mbox	Code compilation time (hrs)	Analysis time (hrs)	Peak memory usage (GBs)
PRADS	0.2	0.25	0.3
Snort	1.5	19	6
HAproxy	0.25	6	6
OpenVPN	0.5	5	7.3

Table 3: Time and memory usage

PRADS which has just 10K lines of code (and 29 critical variables). Table 2 also includes the number of updateable top-level variables, as well as the number of top-level variables that impact packet and log output. We observe that 33-60% of the critical top-level variables are updateable. Being off-path, PRADS has no variables that impact packet output, and 60% (6 out of 10) of updateable variables impact log output; Snort is similar (88 out of 148). For on-path HAproxy (OpenVPN), 85% (88%) of updateable variables affect packet output.

The complexity of middlebox code, (Table 1) coupled with the high number of critical variables we identify, points to the fact that manually identifying critical state, and optimizing its transfer, is extremely difficult. And, the non-trivial reductions we observe in going from all critical top-level variables to updateable ones and further to those impacting packet output show that our techniques in Section 3.2 and Section 3.3 offer useful levels of precision. However, we are as yet unsure if our techniques' precision is optimal, a topic we leave for future work.

6.2 Analysis Efficiency

Table 3 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 the peak memory usage results. Again, the most complex middlebox, Snort, takes the longest to compile and analyze: ≈20.5 hours in total. However, StateAlyzr only needs to be run once, and can be run offline, so this is not a major concern.

6.2.1 Comparison with Other Approaches

We compare StateAlyzr's efficiency and completeness against three alternative approaches: (i) run time analysis, (ii) symbolic execution, and (iii) manual code inspection.

We run PRADS and Snort with/without the run time monitoring discussed in Section 3.1.1—i.e., tracking all memory (de)allocations and all memory reads/writes during packet processing. We employ a university-to-cloud trace of 1 million packets collected at our campus border router for this analysis. As shown in Figure 7, it takes PRADS and Snort 13× and 11× longer, respectively, to process the packet trace when run time analysis is used. Thus, exhaustive run-time monitoring is impractical w.r.t. enabling redistribution.

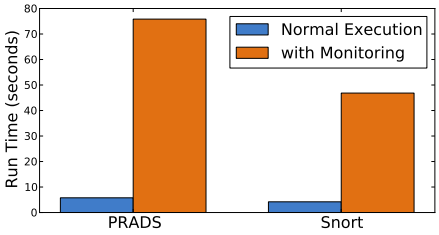


Figure 7: Overhead of using run time analysis to identify critical middlebox state

We symbolically execute PRADS using S2E [10] to identify critical state. After 8 hours—16× longer than it takes to run StateAlyzr—S2E had only covered 13% of the code. Given that PRADS is the simplest middlebox we test (only ≈10K lines of code), it is clear that symbolic execution is impractical.

We consider manually inspecting the middlebox code to identify critical middlebox state. The authors of OpenNF [16] informed us that it took them several days to manually identify the critical state in PRADS. We compared StateAlyzr's output for PRADS against the variables contained in the state transfer code added by the authors of OpenNF. We found that they *missed* an important compound value that contains a few counters along with configuration settings; the latter are never updated during packet processing. StateAlyzr also found four other global variables—`tos`, `tstamp`, `in_pkt`, and `mtu`—but did not mark these variables as affecting packet output or log output. We verified through manual inspection of the code that these values are updated as packets are processed, but they are never used; these variables can thus be removed from PRADS without any impact on its output, pointing to another benefit of StateAlyzr—code clean-up.

6.3 Highly Available PRADS

We use the highly-available version of PRADS (Section 4) to evaluate StateAlyzr's output equivalence, as well as to quantify the run-time benefits of StateAlyzr's optimization techniques in avoiding unneeded state transfer, further underscoring the usefulness of the precision they offer.

6.3.1 Output Equivalence

To evaluate the output equivalence of our highly available PRADS, we use two instances, one as primary and the other in hot standby mode. The primary middlebox sends a copy of the state to the hot standby after processing each packet. We use another university-to-cloud packet trace with around 700k packets for this evaluation. The primary instance pro-

cesses the first half of the trace file till a random point and the hot standby takes over after that. We compare the assets logged by PRADS in the scenario where a single instance processes the complete trace file against concatenated logs of the primary and hot standby. We considered the following models, reflecting progressive application of three different optimizations in Sections 3.2, 3.4, and 4:[9] 1) where primary instance sends a copy of all the updateable states to the hot standby, 2) where primary instance only sends the state which applies to the flowspace of the last processed packet, and 3) where in addition to considering the flowspace, we also consider which top level variables are marked as updated for the last processed packet. For all three models, there was no difference in the assets logged in comparison with the scenario where all the traffic was processed by a single instance, implying output equivalence.

6.3.2 Fine Grained Marking

To evaluate the benefits of code instrumentation for marking whether some updateable state is actually updated for a particular packet, we compare the amount of per packet data transferred between the primary instance and its hot standby for all three models discussed in the previous section. Figure 8 shows the average case results for PRADS for all three models. Transferring state which only applies to the flow space of last processed packet, i.e., the second model, reduces the data transferred by 305×; we also found that output equivalence was maintained. This provides empirical evidence of the soundness and high precision of our techniques for identifying which traffic can update/read a particular piece of state (Section 3.4).

Furthermore, we found that the third model, i.e., run time marking of updated state variables (Section 4) further reduces the amount of data transferred by 2×, on average, relative to the second model. This is due to the fact that not all values are updated for every packet: the values pertaining 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 9, which shows the size of the state transfer after processing each of the first 200 packets in a randomly selected flow. Sending deltas and compression can help further optimize data transfer.

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 0.04usec, which is around 0.14% of the average per packet processing time for unmodified PRADS.

7. Other Related Work

Aside from the works discussed in Sections 2 and 3 [9, 16, 17, 21, 23–27, 29] StateAlyzr is related to a few other efforts.

[9] Note: PRADS has no packet output-impacting state.

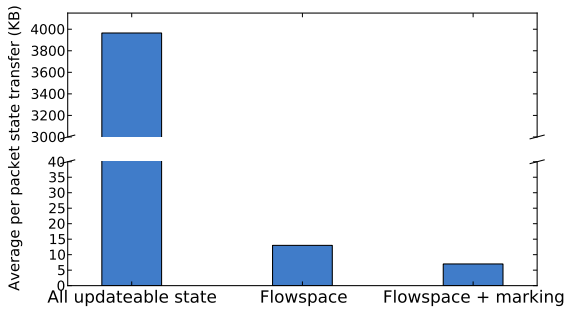


Figure 8: Per packet state transfer for three different models

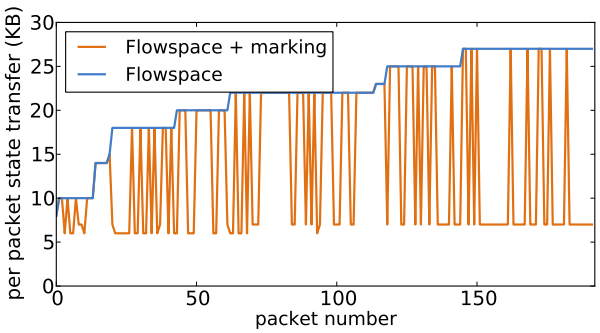


Figure 9: Per packet state transfer for a single connection

Some prior studies have focused on transforming non-distributed applications into distributed applications [19, 28]. However, these works aim to run different parts of an application at different locations/machines. We want all analysis steps performed by a middlebox to run on one machine, but we want different machines to run these analysis steps on a different set of input without changing the collective output from all machines.

Dobrescu and Argyarki have used symbolic execution to verify middlebox code satisfies crash-freedom, bounded-execution, and other important safety properties [13]. They employ small, Click-based middleboxes [20] 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.

8. Summary

Our goal was to enable developers of arbitrary applications to automatically identify locations in their code that maintain live state that must be migrated or cloned when an input workload is dynamically redistributed. Today, this has to be done manually by application developers. Given the complexity of application code, this is a daunting task. In this paper, we showed how static analysis techniques can be applied toward achieving this goal. Focusing on middleboxes, we showed how to leverage their code structure and common design patterns to identify what state is critical, updateable, packet or log output-impacting, and read/updated when processing traffic in a given flow space. We formally proved

the soundness of our techniques. We applied StateAlyzr to 4 open source middleboxes and showed that our techniques remove unneeded state transfers. Finally, we showed how our techniques can be used toward an efficient, highly available middlebox design. While our focus was on middleboxes, we believe that the same lessons—leveraging common structures and design patterns across applications of a given class—can be employed to enable live state handling in more generic application settings.

References

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

[2] Dyninst: Putting the Performance in High Performance Computing. `http://dyninst.org`.

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

[4] Kubernetes. `http://kubernetes.io`.

[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] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

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

[14] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. COLO: COarse-grained LOck-stepping virtual machines for non-stop service. In *SoCC*, 2013.

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

[18] P. Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *ACM Conference on LISP and Functional Programming (LFP)*, 1986.

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

[20] 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.

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

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

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

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

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

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

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

[28] 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.

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