Software Dataplane Verification

Mihai Dobrescu and Katerina Argyraki
EPFL, Switzerland

Abstract

Software dataplanes are emerging as an alternative to traditional hardware switches and routers, promising programmability and short time to market. These advantages are set against the risk of disrupting the network with bugs, unpredictable performance, or security vulnerabilities. We explore the feasibility of verifying software dataplanes to ensure smooth network operation. For general programs, verifiability and performance are competing goals; we argue that software dataplanes are different—we can write them in a way that enables verification *and* preserves performance. We present a verification tool that takes as input a software dataplane, written in a way that meets a given set of conditions, and (dis)proves that the dataplane satisfies crash-freedom, bounded-execution, and filtering properties. We evaluate our tool on stateless and simple stateful Click pipelines; we perform complete and sound verification of these pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within several hours.

1 Introduction

Software dataplanes are emerging from both research [16,25,26,36] and industry [2,3] backgrounds as a more flexible alternative to traditional hardware switches and routers. They promise to cut network provisioning costs by half, by enabling dynamic allocation of packet-processing tasks to network devices [41]; or to turn the Internet into an evolvable architecture, by enabling continuous functionality update of devices located at strategic network points [40].

Flexibility, however, typically comes at the cost of reliability. A system of non-trivial size that is subject to frequent updates is typically plagued by behavior and performance bugs, as well as security vulnerabilities. It makes sense then that network operators are skeptical about the vision of software dataplanes that are continuously reprogrammed in response to user and operator needs—as they were skeptical a decade ago toward active networking. The question is, has anything changed? Have software verification techniques matured enough to enable us to reason about the behavior and performance of software dataplanes? Or must we accept that frequently reprogrammed software dataplanes will always be less reliable than their static hardware counterparts?

The subject of this work is a verification tool that takes as input the executable binary of a software dataplane and proves that it does (or does not) satisfy a target property; if the target property is not satisfied, the tool should provide counter-examples, i.e., packet sequences that cause the property to be violated. Developers of packet-processing apps could use such a tool to produce software with guarantees, e.g., that never seg-faults or kernel-panics, no matter what traffic it receives. Network operators could use the tool to verify that a new packet-processing app they are considering for deployment will not destabilize their network, e.g., it will not introduce more than some known fixed amount of per-packet latency. One might even envision markets for packet-processing apps—similar to today's smartphone/tablet app markets—where network operators would shop for new code to "drop" into their network devices. The operators of such markets would need a verification tool to certify that their apps will not disrupt their customers' networks.

For general programs, verifiability and performance are competing goals. Proving properties of real programs (unlike searching for bugs) remains an elusive goal for the systems community, at least for programs that consist of more than a few hundred lines of code and are written in a low-level language like C++. A high-level language like Haskell can guarantee certain properties (like the impossibility of buffer overflow) by construction, but typically at the cost of performance.

For software dataplanes, it does not have to be this way: we will argue that we can write them in a way that enables verification and preserves performance. The key question then is: what defines a "software dataplane" and how much more restricted is it than a "general program"? how much do we need to restrict our dataplane programming model so that we can reconcile verifiability with performance?

There are different ways to approach this question: one could start from a restricted, easily verifiable model and broaden it as much as possible without losing verifiability; or, one could start from a popular, but not verifiable model and restrict it as little as necessary to achieve verifiability. We chose the latter in an effort to be practical. We present in this paper the result of working iteratively on two tasks: designing a verification tool for software dataplanes, while trying to identify a minimal set of conditions that a software dataplane must meet in order to be verifiable.

We fundamentally rely on the assumption that software dataplanes follow a pipeline structure, i.e., they are composed of distinct packet-processing elements (e.g., an IP lookup element, an element that performs Network Address Translation or NAT) that are organized in a directed graph and do not share mutable state. Intuitively, the fact that there are no state interactions between elements (other than one passing a packet to another) makes it feasible to reason about each element in isolation, as opposed to having to reason about the entire pipeline as a whole. Software dataplanes that are created with Click [32] typically conform to this structure, and these arguably constitute the majority of research prototypes. We also know of at least one industry prototype that uses Click [1], while the vision of a "composable" dataplane put forward by Intel earlier this year [3] strongly implies a pipeline structure as well.

We aim to prove properties that, in the case of hardware dataplanes, are either taken for granted or can be proved using practical techniques [27–29, 37, 42]: *crash-freedom*, which means that no packet sequence can cause the dataplane to stop executing; *bounded-execution*, which means that no packet sequence can cause the execution of more than a known, reasonable number of instructions; or *filtering* properties, e.g., "any packet with source IP A and destination IP B will be dropped by the pipeline."

In this paper, we describe a verification tool that proves such properties for stateless pipelines (e.g., an IP router or static firewall) and two simple stateful pipelines (a NAT box and a traffic monitor). Certain proofs assume arbitrary configuration[1], while others assume a specific one. For instance, we prove crash-freedom or bounded-execution assuming arbitrary configuration, and such proofs are useful independently of the frequency of configuration changes. In contrast, proving that a pipeline will drop a packet with given headers makes sense only given a specific configuration, and such proofs are useful when configuration changes relatively slowly.

We evaluate our tool by proving crash-freedom and bounded-execution for different Click pipelines. Our proofs complete within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within hours. Keeping verification time within minutes is necessary and sufficient given our goals: We envision our tool being used by developers, for instance to ensure that a new piece of packet-processing code cannot seg-fault, or by network operators, for instance to ensure that a given configuration change will not result in undesirable network behavior. In both cases, having to wait for hours would be impractical; waiting for tens of minutes is non-negligible, but on par with the experience of waiting for compilation to complete or configuration to be downloaded to network devices.

Even though we focus on conceptually simple pipelines, performing complete and sound verification on them required overcoming significant challenges (dealing with path explosion, loops, and large data structures). Our contribution is to address these challenges by applying existing verification ideas (symbolic execution [10, 20] and compositionality [4, 19, 21]) and combining them with certain domain specifics of packet-processing software (pipeline structure, bounded loops over packet contents, pre-allocated data structures that expose a key/value store interface). We share common ground with many verification tools, especially the ones that use compositional symbolic execution [4, 19, 21], but, to the best of our understanding, those tools alone cannot solve our problem (they were not designed with software dataplanes in mind).

The rest of the paper is organized as follows: After providing the necessary background (§2), we describe our system (§3) and the properties that it can prove (§4). Then we present our evaluation (§5), discuss limitations (§6) and related work (§7), and conclude (§8).

2 Setup

In this section, we provide background on symbolic execution (§2.1), summarize our approach (§2.2), and describe our basic assumption about the structure of software dataplanes (§2.3).

2.1 Background

Symbolic Execution.

A program can be viewed as an "execution tree," where each node corresponds to a program state, and each edge is associated with a basic block. Running the program for a given input leads to the execution of a sequence of instructions that corresponds to a path through the execution tree, from the root to a leaf. For example, the program E_1 in Fig. 1 may execute two instruction sequences: one for input $in < 0$ and the other for input $in \geq 0$; hence, its execution tree (shown to the right of the program) consists of two edges, one for each "input class" and instruction sequence.

Symbolic execution [10, 20] is a practical way of generating execution trees. During normal execution of a program, each variable is assigned a concrete value, and only a single path of the tree is executed. During symbolic execution, a variable may be symbolic, i.e., assigned a set of values that is specified by an associated constraint. For example, a symbolic integer x with associated constraint $x > 2 \wedge x < 5$ is the set of concrete values $x = \{3, 4\}$. A symbolic-execution engine can

[1] By "configuration" we mean all state that the control plane writes into the dataplane, e.g., the contents of forwarding or filtering tables.

take a program, make the program's input symbolic, and execute all the paths that are feasible given this input.

Consider the program E_2 in Fig. 1 and assume that the input in can take any integer value. To symbolically execute this program, we start at the root of the tree and execute all the feasible paths. As we go down each path, we collect two pieces of information: the "path constraint" specifies which values of in lead to this path, and the "symbolic state" maps each variable to its current value on this path. For example, at the end of path e_4, the path constraint is $C = (in \geq 0 \wedge in < 10)$, and the symbolic state is $S = \{out = 10\}$; at the end of path e_5, the path constraint is $C = (in \geq 10)$, and the symbolic state is $S = \{out = in\}$.

Construction of Proofs.

If we can execute all the feasible paths of a program and verify that none of them violates a target property, that constitutes proof that the entire program satisfies this property. By constructing proofs in this manner, we can also automatically determine all the problematic inputs that prevent us from completing the proof.

However, proof by execution can be rarely used in practice, because of path explosion [9]: The sheer number of feasible paths in a real program (even one that consists of a few hundred lines of code) is typically so large that it is impossible to execute all of them in useful time. This is because the number of paths generally grows exponentially in the number of branching points, and real software has a branching point every few instructions. For instance, when Klee [10] symbolically executes UNIX coreutils like nice or cat, it achieves more than 70% line coverage, but executes less than 1% of the feasible paths [34]. This is fine when the goal is high line coverage or discovery of interesting paths (e.g., to uncover bugs), but not when the goal is to reason about all feasible paths (i.e., to prove properties).

Researchers have been proposing smarter ways to address path explosion [4,19,21,34], but constructing complete and sound proofs for real programs that consist of more than a few hundred lines of code still takes a lot of manual effort [31].

2.2 Our Approach

We observe that symbolic execution is a good fit for packet-processing pipelines, because their special structure can help sidestep path explosion. In a typical pipeline, two elements (stages) never concurrently hold read or write permissions to the same mutable state, regardless of whether that state is a packet being processed or some other data structure. This level of isolation can help significantly with path explosion.

Our approach is to first analyze each pipeline element in isolation, then compose the results to prove properties about the entire pipeline. This reduces by an exponential factor the amount of work that needs to be done to prove something about the pipeline: If each element has n branches and roughly 2^n paths, a pipeline of m such elements has roughly $2^{m \cdot n}$ paths. Analyzing each element in isolation—as opposed to the entire pipeline in one piece—cuts the number of paths that need to be explored roughly from $2^{m \cdot n}$ to $m \cdot 2^n$. In the worst case, the per-element analyses yield that every single pipeline path warrants further analysis—so we end up having to consider all the paths anyway. In practice, we expect that most pipeline paths are irrelevant to the target property, and we only need to consider a small fraction.

Our verifier relies on S2E [13], an automated path explorer with pluggable path analyzers: the explorer uses symbolic execution to drive the target system down multiple execution paths, while the analyzers measure and/or check properties of each such path. We chose S2E for two reasons: First, it performs what is called "in-vivo" (as opposed to "in-vitro") program analysis, i.e., analyzes code that runs within a real (not modeled) software stack; this enables us to analyze a software pipeline without having to model the underlying system—libraries, kernel, drivers, etc. Second, it can directly analyze binaries (as opposed to source code); this enables us to analyze proprietary packet-processing elements, for which we do not have access to the source code. We use S2E as a building block, to symbolically execute pieces of packet-processing code and obtain, for each piece, a set of path constraints and symbolic states.

2.3 Starting Point: Pipeline Structure

We focus on packet-processing pipelines that consist of packet-processing elements, where each element may access three types of state (Table 1):

Packet state is owned by exactly one element at any point in time. It can be read or written only by its owner; the current owner (and nobody else) may atomically transfer ownership to another element. Packet state is used for communicating packet content and metadata between elements. For each newly arrived packet, there is typically an element that reads it from the network, creates a packet object, and transfers object ownership to the next element in the pipeline. Once an element has transferred ownership of a packet, it cannot read or write it any more.

Private state is owned by one element and never changes ownership. It can be read or written only by its owner, and it persists across the processing of multiple packets. A typical example is a map in a NAT element, or a flow table in a traffic-monitoring element.

Static state can be read by any element but not written by any element. This state is immutable as far as the

	Written by	Read by	Transferable ownership
Packet state	owner	owner	yes
Private state	owner	owner	no
Static state	–	any	–

Table 1: Types of packet-processing state.

pipeline is concerned. A typical example is an IP forwarding table.

This structure is not accidental: it is a natural fit for any platform that must perform high-performance streaming. The alternative would be to allow multiple stages of the pipeline to share read/write access to the same data, which would necessarily require synchronization and the unavoidable contention and complexity that comes with it.

3 System

In this section, we describe our system: first how it leverages the pipeline structure to sidestep inter-element path explosion (§3.1); second, how it leverages other aspects of packet processing to sidestep intra-element path explosion resulting from loops (§3.2), large data structures (§3.3), and mutable state (§3.4).

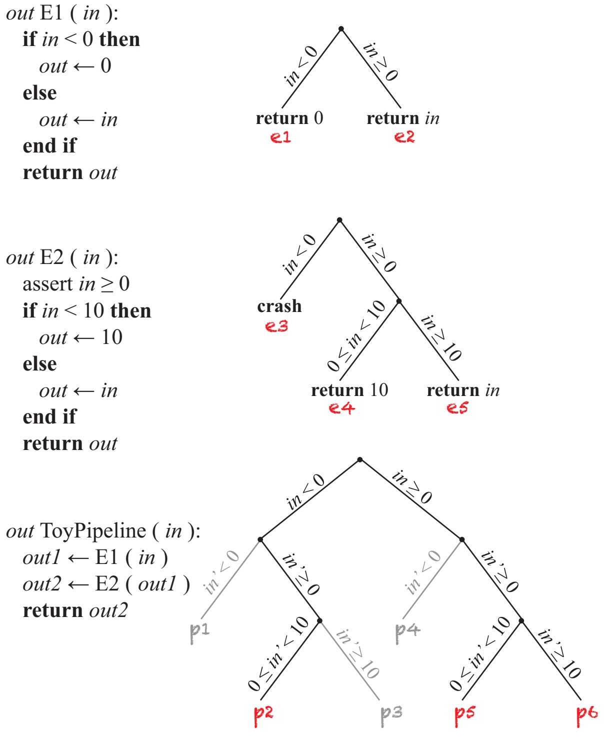
As we describe each technique used by our system, we also state any extra conditions (on top of pipeline structure) that this technique requires from the target software in order to work well. If a software dataplane does not meet these conditions, we may not be able to construct complete and sound proofs for it. When we fail, we know it, i.e., we never construct incomplete or unsound proofs.

We will illustrate our system through Fig. 1, which shows a pipeline consisting of two elements. We will use the term *segment* to refer to an instruction sequence through a single element, and the term *path* to refer to an instruction sequence through the entire pipeline. The input in corresponds to a newly received packet, and we assume that this may contain anything, i.e., we make in symbolic and unconstrained.

For illustration purposes, our examples simplify two aspects of our system: first, our example input in is an integer, whereas in reality the input packet object is an array of bytes; second, our example code snippets consist of pseudo-code, whereas in reality S2E takes as input X86 code.

3.1 Pipeline Decomposition

Verification consists of two main steps: step 1 searches inside each element, in isolation, for code that may violate the target property, while step 2 determines which of these potential violations are feasible once we assemble the elements into a pipeline. More specifically, we cut each pipeline path into element-level segments (Fig. 1). In step 1, we obtain, for each segment, a logical expression that specifies how this segment transforms state; this allows us to identify all the "suspect segments," which may cause the target property to be violated. In step 2, we determine which of the suspect segments are feasible and indeed cause the target property to be violated, once we assemble segments into paths.



```
out E1 ( in ):
  if $in < 0$ then
    $out \leftarrow 0$
  else
    $out \leftarrow in$
  end if
  return out

out E2 ( in ):
  assert $in \geq 0$
  if $in < 10$ then
    $out \leftarrow 10$
  else
    $out \leftarrow in$
  end if
  return out

out ToyPipeline ( in ):
  $out1 \leftarrow$ E1 ( in )
  $out2 \leftarrow$ E2 ( out1 )
  return out2
```

Figure 1: A toy pipeline that consists of two elements.

In step 1, we analyze each element in isolation: First, we symbolically execute the element assuming unconstrained symbolic input. Next, we conservatively tag as "suspect" all the segments that may cause the target property to be violated. E.g., in Fig. 1, if the target property is crash-freedom, segment e_3 is tagged as suspect, because, if executed, it leads to a crash.

If we stopped at step 1, our verification would catch all property violations, but could yield false positives: If this step does not yield any suspect segments for any element, then we have proved that the pipeline satisfies the target property. For instance, if none of the elements ever crashes for any input, we have proved that the pipeline never crashes. However, a suspect segment does not necessarily mean that the pipeline violates the target property, because a segment that is feasible in the context of an individual element may become infeasible in the context of the full pipeline. For example, in Fig. 1, if we consider element E_2 alone, segment e_3 leads to a crash; however, in a pipeline where E_2 always follows E_1, segment e_3 becomes infeasible, and the pipeline never crashes. In program-analysis terminology, in step 1, we over-approximate, i.e., we execute some segments that would never be executed within the pipeline that we are aiming to verify.

Step 2 discards suspect segments that are infeasible in the context of the pipeline: First, we construct each potential path p_i that includes at least one suspect segment; each p_i is a sequence of segments e_j. Next, we compose the path constraint and symbolic state for p_i based on the constraints and symbolic state of its constituent segments (that we have already obtained in step 1). Finally, for every p_i, we determine whether it is feasible (based on its constraints) and whether it violates the target property (based on its symbolic state). Note that the last step does not require actually executing p_i, only composing the logical expressions of its constituent segments.

For instance, here is how we prove that the pipeline in Fig. 1 does not crash:

Step 1:

1. We symbolically execute E_1 assuming input in can take any integer value. We collect the following constraints and symbolic state for its segments e_1 and e_2.
   - $C_1(in) = (in < 0)$, $S_1(in) = \{out = 0\}$.
   - $C_2(in) = (in \geq 0)$, $S_2(in) = \{out = in\}$.
2. We symbolically execute E_2 assuming input in can take any integer value. We collect the following constraints and symbolic state for its segments e_3, e_4, and e_5:
   - $C_3(in) = (in < 0)$, $S_3(in) = \{crash\}$.
   - $C_4(in) = (in \geq 0 \wedge in < 10)$, $S_4(in) = \{out = 10\}$.
   - $C_5(in) = (in \geq 10)$, $S_5(in) = \{out = in\}$.
3. We tag segment e_3 as suspect.

Step 2:

1. The paths that include the suspect segment are p_1 (i.e., sequence $< e_1, e_3 >$) and p_4 (i.e., sequence $< e_2, e_3 >$).
2. We compute p_1's path constraint as $C_1^*(in) =$

   $$C_1(in) \wedge C_3(\ S_1(in)\ [out]\ ) =$$
   $$C_1(in) \wedge C_3(\ 0\ ) = (in < 0) \wedge (0 < 0) = False.$$

3. We compute p_4's path constraint as $C_4^*(in) =$

   $$C_2(in) \wedge C_3(\ S_2(in)\ [out]\ ) =$$
   $$C_2(in) \wedge C_3(\ in\ ) = (in \geq 0) \wedge (in < 0) = False.$$

4. Both path p_1's and path p_4's constraints always evaluate to false, hence p_1 and p_4 are infeasible, i.e., there are no feasible paths that include suspect segments, hence the platform never crashes.

Pipeline decomposition enables us to prove properties about the pipeline without having to consider every single pipeline path; but it still requires us to consider every single element segment. This is not straightforward for elements that involve loops, large data structures, and/or mutable private state. We will next discuss how we address each of these scenarios.

3.2 Loops

In general, loops can be a challenge for program verification, especially when the number of loop iterations depends on the input. For example, a loop of n iterations, where n is a 64-bit integer input, can yield as many as 2^{64} execution paths.

In contrast to general programs, a software dataplane typically will not contain input-dependent loops with such a large number of maximum iterations. A worst-case realistic example is a packet-processing element that loops over the bytes of a packet for encryption or compression; in this case, the number of loop iterations is bounded by the maximum packet size, typically 1500.

Still, loops can create an impractical number of segments within an element. Consider an element that implements the processing of IP options: for each received packet, it loops over the options stored in the packet's IP header and performs the processing required by each specified option type. If the processing of one option yields up to 2^n segments, then the processing of m options yields up to $2^{n \cdot m}$ segments. For example, in the IP-options element that comes with the Click distribution, the processing of 3 options yields millions of segments that—we estimated—would take months to symbolically execute.

To address this, we reuse the idea of decomposition, this time applying it not to the entire pipeline, but to each loop: If a loop has t iterations, we view it as a "mini-pipeline" that consists of t "mini-elements," each one corresponding to one iteration of the loop. We have described how, if we have a pipeline of k elements, we symbolically execute each element in isolation, then compose the results to reason about the entire pipeline. Similarly, if we have a loop of t mini-elements (iterations), we symbolically execute each mini-element in isolation, then compose the results to reason about the entire loop. Unlike a pipeline that consists of different element types, a loop of t iterations consists of the same mini-element type, repeated t times; hence, for each loop, we only need to symbolically execute one mini-element.

This brings us to our first extra condition on packet-processing code: To use decomposition as we do, the only mutable state shared across components must be the packet object itself. For instance, to decompose a pipeline into individual elements, we rely on the fact that the only mutable state shared across elements is the packet object. Similarly, to decompose a loop into individual iterations, the only mutable state shared across iterations must be part of the packet object.

For example, consider again an IP-options element: Such an element typically includes a next variable, which points to the IP-header location that stores the next option to be processed; each iteration of the main loop starts by reading this variable and ends by incrementing it. In a conventional element, next would be a local variable. In our verification-optimized element, next is part of the packet metadata, hence part of packet. And since, in step 1, we make packet symbolic and unconstrained, next is also symbolic and unconstrained, allowing us to reason about the behavior of one iteration of the main loop, assuming that iteration may start reading from *anywhere* in the IP header.

Condition 1 *Any mutable state shared across loop iterations is part of the packet metadata.*

To make a packet-processing element satisfy this condition, a developer needs to identify any variables that are read and written across loop iterations and make these variables part of the packet metadata. For the Click IP-options element, this process required changing 14 lines of code and took less than an hour. Alternatively, this can be done automatically by a compiler (that would force developers to explicitly declare mutable state shared across iterations of a loop). Either way, this condition does not restrict the functionality that a packet-processing element can implement; it only forces the developer to create—either manually or with compiler help—an explicit interface between loop iterations.

3.3 Data Structures

Symbolic-execution engines lack the semantics to reason about data structures in a scalable manner. For instance, symbolically executing an element that uses a packet's destination IP address to index an array with a thousand entries will cause a symbolic-execution engine to essentially branch into a thousand different segments—independently from the array content or the logic of the code that uses the returned value. So, if we naïvely feed an element with a forwarding or filtering table of more than a few hundred entries to a symbolic-execution engine, step 1 of our verification process will not complete in useful time.

```
         value   =   read ( key )
                     write ( key, value )
{True, False}    =   test ( key )
                     expire ( key, value )
```

Figure 2: An interface for dataplane data structures.

To address this, when we reason about an element, we abstract away any data-structure access; this allows us to symbolically execute the element and identify suspect segments, without requiring the symbolic-execution engine to handle any data structures. To reason about the data structures themselves, we rely on other means, e.g., manual or static analysis; this restricts us to using only data structures that are manually or statically verifiable, but we have evidence that these are typically sufficient for packet-processing functionality.

This brings us to our second extra condition on packet-processing code: To reason about different components of the same executable separately, there must exist a well-defined interface between them. For instance, to reason about each pipeline element separately and compose the results, we rely on the existence of a well-defined interface between each pair of elements, which specifies all the state that can be exchanged between them (the packet object). Similarly, to reason about a data structure separately from the element that uses it and compose the results, the data structure must expose a well-defined interface to the element.

We need an interface that abstracts a data structure as a key/value store that supports at least read, write, membership test, and expiration. The first three operations are straightforward; the last one—expiration—allows an element to indicate that a {key, value} pair will not be accessed by the element any more, hence is ready to be removed and processed by the higher layers. For example, suppose an element maintains a data structure with per-flow packet counters; when a flow completes (e.g., because a FIN packet from that flow is observed), the element can use the expiration operation to signal this completion to the control-plane process that manages traffic statistics.

Condition 2 *Elements use data structures that expose a key/value-store interface like the one in Fig. 2.*

Moreover, we need data structures that expose the above interface *and* can be verified in useful time. When we say that a data structure is "verified," we mean that the implementation of the interface exposed by the data structure is proved to satisfy crash-freedom, bounded-execution, and correctness. The latter depends on the particular semantics of the data structure, e.g., a hash-table should satisfy the following property: a "write (key, value)" followed by a "read (key)" should return "value." If an element uses only data structures for which these properties hold, then, when we reason about the element, we can abstract away all the data-structure implementations and consider only the rest of the element code plus the data-structure interface.

Condition 3 *Elements use data structures that are implemented on top of verifiable building blocks, e.g., pre-allocated arrays.*

As evidence that such data structures exist, we implemented a hash table and a longest-prefix-match table that satisfy crash-freedom and bounded-execution. They both consist of chains of pre-allocated arrays. Our hash table is a sequence of N such arrays; when adding the n-th key/value pair that hashes to the same index, if $n \leq N$, the new pair is stored in the n-th array, otherwise it cannot be added (the write operation returns False). For the longest-prefix-match table, we use the idea of "flattening" of all entries to $/24$ prefixes [24].

We chose arrays as the main building block, because they combine two desirable properties: (a) They enable line-rate access to packet-processing state, because of their $O(1)$ lookup time. (b) They are easy to verify, because of the simplicity of their semantics. For example, a write to an array (that is within the array bounds) is guaranteed not to cause a crash and not to cause the execution of more than a known number of instructions that depends on the particular CPU architecture. In contrast, a write to a dynamically-growing data structure, e.g., a linked list, or a radix trie, may result in a variable number of memory allocations, deallocations and accesses, which can fail in unpredictable ways.

Conditions 2 and 3 introduce two kinds of overhead:

First, existing elements may need to be rewritten to satisfy them; this involves replacing existing data structures with ones that satisfy the two conditions and changing any line of code that accesses a data structure. We have not yet applied our approach widely enough to have statistically meaningful results on the corresponding amount of effort. In one case (Click IP lookup element), we had to change about 300 lines of code, which took a few hours. In another case (Click NAT element), we had to write the element from scratch (because most of the NAT code is about accessing data structures), which took a couple of days. To reduce this overhead, part of our work is to create a library of data structures that satisfy the two conditions.

Second, implementing sophisticated data structures on top of pre-allocated arrays typically requires more memory than conventional implementations. For instance, the original Click NAT element stores per-connection state in a hash table, implemented as an array of dynamically growing linked lists (when adding the n-th key/value pair that hashes to the same index, the new pair is stored as the n-the item of a linked list). In contrast, our NAT element uses the hash-table implementation outlined above, with $N = 3$ pre-allocated arrays (this value makes the probability of dropping a connection negligible). Hence, our NAT element may use up to 3 times more memory to store the same amount of state. In our opinion, sacrificing memory for verifiability is worth considering given the relative costs of memory and the human support for dealing with network problems.

3.4 Mutable Private State

Mutable private state is hard to reason about because it may depend on a sequence of observed packets (as opposed to the currently observed packet alone). For instance, if an element maintains connection state (e.g., NAT) or traffic statistics (e.g., NetFlow), then its private state is a function of all traffic observed since the element was initialized. Hence, it is not enough to reason about the segments of the element that can result from all possible contents of the current packet; we need to reason about the segments that can result from all possible contents of all possible packet sequences that can be observed by the element. The challenge is that symbolic-execution engines (and verification tools in general) are not yet at the point where they can handle symbolic inputs of arbitrary length in a scalable manner.

We can currently verify two kinds of elements that maintain mutable state: a NAT element (maintains per-connection state and rewrites packet headers accordingly) and a traffic monitor (collects per-flow statistics). We believe that our approach can be generalized to other elements, but we do not expect to be able to perform complete and sound verification of an element that performs arbitrary state manipulation—claiming that would be close to claiming that we could verify arbitrary software.

Our approach is to break verification step 1 (§3.1) into two sub-steps: the first one searches for "suspect" values of the private state that would cause the target property to be violated, while the second one determines which of these potential violations are feasible given the logic of the element. In the first sub-step, we assume that the private state can take *any* value allowed by its type (i.e., we over-approximate). In the second sub-step, we take into account the fact that private state cannot, in reality, take any value, but is restricted by the particular type of state manipulation performed by the given element.

So far, we have not needed to exercise the second sub-step in practice: in the two stateful elements that we have experimented with, the first sub-step did not reveal any suspect states, hence the second one was not exercised. We describe both sub-steps through a manufactured example:

```
1: if $map$.exists($flowId$) $=$ false then
2:     $map$.write($flowId, 0$)
3: end if
4: $pktCnt \leftarrow map$.read($flowId$)
5: $newPktCnt \leftarrow pktCnt + 1$
6: $map$.write($flowId, newPktCnt$)
```

Figure 3: Code that collects per-flow packet counters.

Fig. 3 shows an element that maintains a private data structure (map) with per-flow packet counters. Lines 1–3 add a new entry for the current flow (where the current packet belongs), if such an entry does not already exist. Line 4 reads the counter of the current flow, line 6 increments the variable that stores the read, and line 11 updates the counter. The target property is that a packet counter never overflows.

In sub-step (i), we identify and tag as "suspect" all the values of the private state that would cause the target property to be violated. To achieve this, we make symbolic and unconstrained not only the input packet and its metadata, but also any variable that stores a read from a private data structure. In Fig. 3, there is one such variable: $pktCnt$. Making $pktCnt$ symbolic and unconstrained allows us to identify the segments that may result from all the $pktCnt$ values allowed by $pktCnt$'s type. The output of the symbolic-execution engine is:

$$\begin{aligned} &C_1(in, pktCnt) = (), \\ &S_1(in, pktCnt) = \{newPktCnt \mapsto pktCnt + 1\}. \end{aligned} \tag{1}$$

This indicates that, if the current flow counter is equal to $\max$ (the maximum value allowed by its type), then the counter will overflow.

If we stopped at sub-step (i), our verification would be complete but not sound: A suspect value does not necessarily mean that the element violates the target property, because the logic of the element may preclude some of these values. Sub-step (ii) makes our verification sound, by checking the feasibility of the suspect values. In our running example, we use the output of sub-step (i) to prove by induction that $\max$ is a feasible value for $pktCnt$, and $newPktCnt$ will overflow, as long as the element observes a sequence of $\max + 1$ packets.

As described so far, our approach for dealing with mutable private state is semi-manual: we said that we use the output of sub-step (i) to prove by induction that $newPktCnt$ can overflow; we construct this proof manually, as we are not aware of any practical, publicly available tool that will do it for us automatically. However, semi-manual verification is not practical—we do not envision software engineers writing formal proofs.

To remove the need for manual construction of proofs, we can turn sub-step (ii) into a pattern-matching problem: (a) Identify patterns of symbolic state that occur frequently in packet-processing elements. (b) Manually construct proofs about the state contained in these specific patterns. (c) When reasoning about an element, to prove that a suspect value of the private state is or is not feasible, try to match the symbolic state that corresponds to the private state to one of the common patterns (for which we have pre-constructed proofs). E.g., if an element contains a piece of private state that matches the pattern in Eq. 1, then this element contains a counter that will eventually overflow. To determine whether such pattern matching would be useful in practice, we need to experiment with more stateful elements than NAT and NetFlow—which will be our next step.

4 Target Properties

In this section, we describe the three target properties that our current prototype can (dis)prove. These properties are expressed imperatively using the S2E analyzer interface [13].

Crash-freedom.

We say that a pipeline is *crash-free* when it is guaranteed not to execute any instruction that would cause it to terminate abnormally, e.g., an assertion with a false argument or a division by zero. The definition of "abnormal termination" depends on the environment where the pipeline runs: in the case of user-mode Click, it is the receipt of a signal (e.g., SIGSEGV, SIGABRT, SIGFPE) that is not handled by the Click process and causes the process to terminate; in the case of kernel-mode Click, it is a call to the kernel's panic method. As stated in §2.2, our verifier is built on top of an in-vivo path explorer, which can detect any of these conditions.

We prove crash-freedom for a pipeline given an arbitrary input packet and arbitrary configuration state. If a pipeline does not include any instruction that may cause abnormal termination, proving crash-freedom is trivial. On the other hand, if a pipeline does include an instruction that may cause abnormal termination, that does not necessarily mean that this instruction may be executed. So, proving crash-freedom is equivalent to proving that any such instruction will never be executed, and proving lack of crash-freedom is equivalent to providing a specific packet and specific state that causes such an instruction to be executed.

Bounded-execution.

We say that a pipeline satisfies *bounded-execution* when it is guaranteed to execute no more than I_{max} instructions per packet. This ensures that no packet is ever caught in an infinite loop. It can also be used to produce a "latency envelope," i.e., argue that once a packet enters the pipeline, it will exit within a bounded amount of time. To translate instruction sequences into latency bounds, we need to map each instruction to the minimum and maximum number of cycles that it can take to complete, which can be typically obtained from the CPU and/or chip manual.

We prove bounded-execution for a pipeline given an arbitrary input packet and arbitrary configuration state. We find the longest path of a pipeline as follows: In step 1 of the verification process, when we symbolically execute an element, we also record the length (number of instructions) of each of its segments. In step 2, we search for the longest feasible path by considering different segment combinations. We use a simple search heuristic that first checks if the path that consists of the longest segment of each element is feasible (if yes, we are done), then checks if any path that involves either the first or second longest segment of each element is feasible, and so on. In the worst case, we have to check all possible segment combinations; in practice, we find the longest feasible path after considering only a few combinations.

Filtering.

Given a pipeline with specific configuration state, can we guarantee that a packet that enters the pipeline with source IP A and destination IP B will be dropped?

We leverage existing work that answers this type of question for hardware dataplanes [27–29, 37, 42]. This work abstracts each network device as a function that maps an input packet header to an output port, and then it composes different device functions to reason about the entire network; the mapping function of each device is determined by the contents of its forwarding table. In contrast, we abstract each packet-processing element as a function that maps an input packet header to an output port, and then we compose different element functions to reason about the entire pipeline; the mapping function of each element is automatically derived by symbolically executing the element's code given an arbitrary input packet.

The main difference lies in the derivation of the mapping function of each packet-processing element (that we do by symbolically executing the element in isolation). This is useful in cases where an element, e.g., includes a line of code that drops all packets with source IP A, even though the device's forwarding table indicates otherwise. Composing element functions to reason about a pipeline is equivalent to composing device functions to reason about a network, and we can reuse the algorithms proposed by the above work.

5 Evaluation

We tested our system on pipelines created with Click. In each tested pipeline, packets are generated by a "generator" element and dropped by a "sink" element; what we verify is the packet-processing code between generator and sink. We answer the following questions: Can we perform complete and sound verification of software dataplanes (§5.1)? How does verification time increase with pipeline length (§5.2)? Can we use our tool to uncover bugs, useful performance characteristics, or unintended dataplane behavior (§5.3)?

Element	New LoC (% of total)	Loops	Data Structs	Mutable State
Click:				
Classifier				
CheckIPhdr				
EthEncap				
EthDecap				
DecTTL				
DropBcast				
Click+:				
IPoptions	26 (12%)	X		
IPlookup	130 (20%)		X	
Ours:				
NAT	870		X	X
TrafficMonitor	650		X	X

Table 2: Verified packet-processing elements. "Click" indicates an unmodified element from Click distribution 2.0.1; "Click+" indicates an element from the same distribution that we modified modestly; "ours" indicates an element that we wrote from scratch. "New LoC" is the number of lines of code that we modified or introduced in each element. "X"s indicate which technique(s) we applied to each element.

5.1 Feasibility

We verified pipelines that consist of various combinations of the elements in Table 2. The table indicates the origin of each element (whether it is an original Click element, one that we modified, or one that we wrote from scratch). Our modifications consisted of loop rewriting and replacing data structures with our verifiable ones. The table also indicates the number of new lines of code (LoC) and which of our techniques were needed to complete step 1 of the verification process for each element.

For each pipeline, we proved crash-freedom and bounded-execution. More generally, for each pipeline, we were able to answer questions of the following kind: can line X in element Y be executed with arguments Z in the context of this pipeline? if yes, what is a packet that would cause this line to be executed with these arguments?

5.2 Scalability

We now examine how verification time increases with pipeline length. Given the intended uses of our tool, it should not take more than a few tens of minutes to prove a target property per pipeline. We first look at meaningful pipelines (that it makes sense to actually deploy), then at microbenchmarks that illustrate different aspects of our system. To show the benefit of our domain-specific techniques, we use as a baseline vanilla S2E—a state-of-the-art, publicly available verification framework for general software. We refer to our tool as "dataplane-specific verification" and to S2E as "generic verification." We feed the same code to the two systems.

Meaningful Pipelines.

We consider three meaningful pipelines: (a) *edge router* implements a standard IP router (the first 8 elements in Table 2) with a small forwarding table (10 entries); (b) *core router* is similar but has a large forwarding table (100, 000 entries); (c) *network gateway* implements NAT and per-flow statistics collection. Each of them presents an extra verification challenge: the first one includes a loop (in IPoptions), the second one a large data structure (in IPlookup), and the third one mutable private state (in NAT and TrafficMonitor).

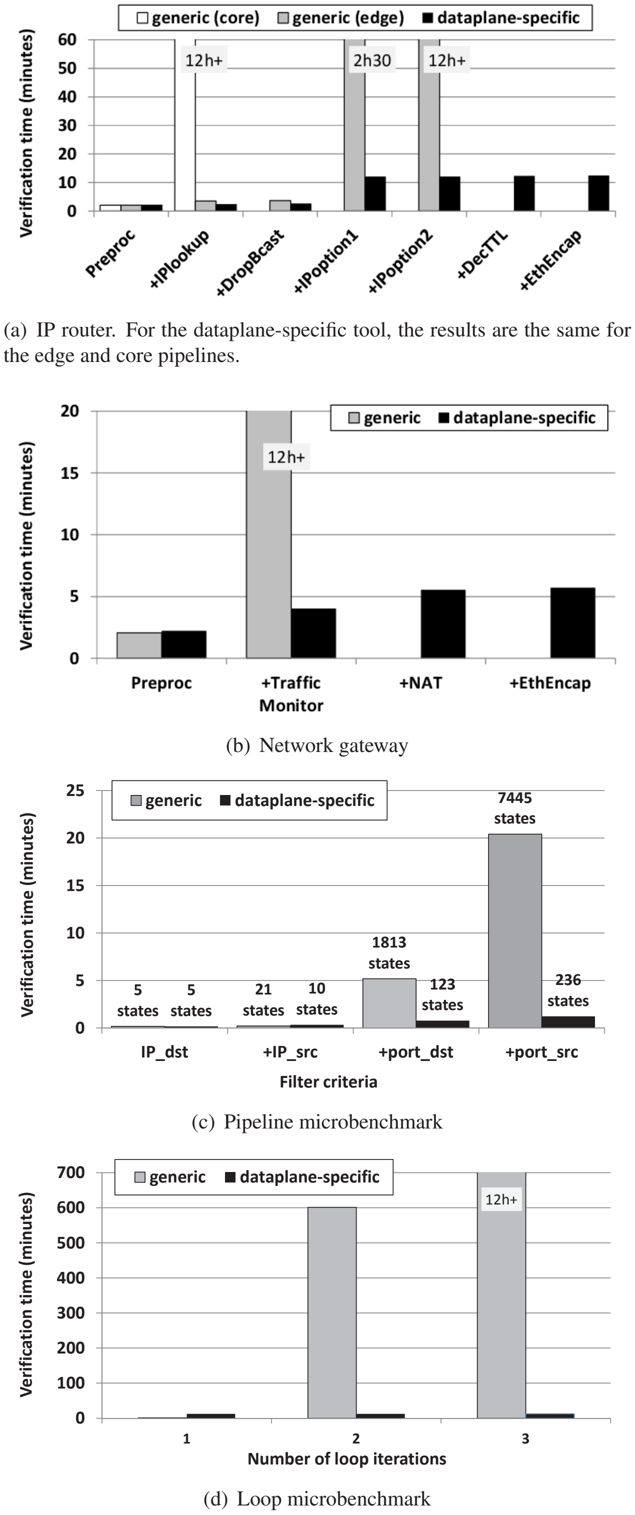
Fig. 4(a) shows how verification time increases as we add more elements to the IP-router pipelines: Dataplane-specific verification completes in less than 20 minutes. Most of this time is spent on IP options, because this element has significantly more branching points than the rest. Generic verification of the edge router exceeds 12 hours (at which point we abort it) the moment we allow packets to carry 2 IP options (so we do not show any data point for it beyond "+IPoption2"). Generic verification of the core router exceeds 12 hours the moment we add the IP lookup element to the pipeline (so we do not show any data point for it beyond "+IPlookup"). The difference between the two tools comes from our special treatment of loops and large data structures.

Fig. 4(b) shows the same information for the network-gateway pipeline: Dataplane-specific verification completes in less than 6 minutes, whereas generic verification exceeds 12 hours the moment we add either the Traffic-Monitor or the NAT element. The difference comes from the fact that we abstract away data-structure implementations.

Compositionality Microbenchmarks.

We consider two synthetic pipelines to illustrate the benefit of pipeline and loop decomposition. The first one consists of a sequence of simple filtering elements, each of which reads a different part of the input packet's IP header to make a filtering decision. The second pipeline implements a simplified version of the IP options processing loop, i.e., in each iteration, it reads some portion of the IP header, updates it, and advances a next variable that indicates where the next read should start.

Fig. 4(c) shows how verification time increases as we add more filtering elements to the first pipeline: generic verification time increases significantly faster than dataplane-specific verification time. This is because the former executes all feasible segments of each ele-



(a) IP router. For the dataplane-specific tool, the results are the same for the edge and core pipelines.

(b) Network gateway

(c) Pipeline microbenchmark

(d) Loop microbenchmark

Figure 4: Verification time as a function of pipeline length. "preproc" consists of the first 3 elements in Table 2.

ment in isolation, whereas the latter executes all feasible paths of the pipeline. In this scenario, generic verification does complete in useful time, because this pipeline involves few elements, without loops, that access minimal state. Still, it takes an order of magnitude more time than dataplane-specific verification because of the exponential increase in the number of paths. To make this clear, we note, on top of each bar, the number of verification states that each tool generates and processes.

Fig. 4(d) shows how verification time increases as we add more iterations to the loop of the second pipeline: dataplane-specific verification time remains constant, whereas generic verification time increases exponentially. This is because the former executes all feasible segments of one loop iteration, whereas the latter executes all feasible paths of the entire loop. Dataplane-specific verification is slower than generic verification only in the special case where we have a loop with a single iteration. That is because it symbolically executes one loop iteration, assuming that iteration may start reading from *anywhere* in the IP header; this pays off as soon as we add a second loop iteration, but it is unnecessary in the special case of a loop with a single iteration.

5.3 Usefulness

We said that our tool can help developers debug their code, and network operators better understand the performance and behavior of their dataplanes; we now look at a few specific examples.

Bugs in Click Elements.

We found the following while trying to prove crash-freedom and bounded-execution for various Click pipelines:

Bug #1: Any pipeline that includes the Click IP fragmenter element will enter an infinite loop, if it tries to fragment a packet with IP options. This is because the for loop that processes IP options in the fragmenter does not have an increment (the programmer forgot to add one).[2]

Bug #2: Any pipeline that does not include an IP options element but includes the Click IP fragmenter element will enter an infinite loop, if it tries to fragment a packet that carries a zero-length IP option. This is because the current option length determines where the next iteration of the loop will start reading, so, a zero-length option causes the loop to get stuck.[3] The Click IP options element discards any packet with a zero-length option, so including it in the pipeline prevents the bug from being exercised.

Bug	Pipeline	Time	# Paths
#1	Edge router with 1 IP option + Click IP fragmenter	3 min	432
#2	Edge router with 1 IP option + Click IP fragmenter	47 min	8423
#2	Edge router without options + Click IP fragmenter	5 sec	26
#3	Network gateway with Click NAT	5 sec	10

Table 3: Time spent and number of paths composed in verification step 2, when the pipeline contains buggy elements.

Bug #3: Any pipeline that includes the Click NAT element[4] will hit a failed assertion[5], if it receives a packet with source IP address/port tuple $T_s = T$ and destination tuple $T_d = T$, where T is the public IP address/port of the NAT box.

All three bugs constitute security vulnerabilities: they enable any end-host to disable the pipeline by sending a specially crafted packet.

How hard would it be to find these bugs manually? The first one is probably not that hard: a loop missing its increment stands out visually, plus any serious testing of the fragmenter element would involve a packet with IP options. The other two bugs, however, manifest in scenarios that a developer is unlikely to test, but an attacker can easily exploit: fragmentation of an illegal packet while processing of IP options is turned off (which does happen, in practice, for performance or security reasons); and processing of an IP header that would be meaningless in a legitimate packet.

In §2.2, we said that we expected verification step 2 to compose the constraints only for a small fraction of the pipeline paths. Table 3 reports, for a given bug and pipeline, the amount of time spent and the number of paths composed in this step. Consider bug #2: When verifying a pipeline that includes the Click IP fragmenter element, step 1 determines that this element has a suspect segment. If the pipeline does not support IP options, step 2 determines that the suspect segment is feasible in this pipeline (hence the pipeline does not satisfy bounded-execution); this requires finding *one* feasible path that contains the suspect segment, and we succeed after composing the constraints for 26 paths, which takes 5 seconds. If the pipeline supports one IP option, step 2 determines that the suspect element is not feasible in this pipeline; this requires showing that *all* the paths that contain the suspect segment are not feasible, and we succeed after composing the constraints for 8423 paths, which takes 47 minutes. These numbers are consistent with our expectation that, in practice, verification step 2 completes in useful time.

[2] elements/ip/ipfragmenter.cc, line 64

[3] elements/ip/ipfragmenter.cc, line 69

[4] elements/tcpudp/iprewriter.cc

[5] include/click/heap.hh, line 149

Longest paths in IP router.

We used our tool to construct adversarial—from a performance point of view—workloads for a pipeline implementing a standard IP router. Recent research showed that such a router is capable of multi-Gbps lines rates [16], but this result was obtained using workloads of well-formed packets, not meant to exercise the pipeline's exception paths. Instead, we obtained the pipeline's 10 (it could have been any number) longest paths, as well as the packets that cause them to be executed.

It is not surprising that the longest paths are executed in response to problematic packets that trigger further packet examination and logging; what may be surprising is that these paths execute 2.5 times as many instructions than the most common path. Moreover, these extra instructions are CPU-heavy, i.e., they include memory accesses and system calls for logging; an attacker may cause significant performance degradation by sending a sequence of packets that are specially crafted to exercise these particular paths. This is useful information to a developer, because it reveals to him paths that may require his attention. It is also useful to a network operator, because it reveals to her the performance limits of a pipeline and the workloads that trigger them—allowing her to decide whether it is suitable for her network.

Unintended behavior.

Certain implementations of the Loose Source Record Route (LSRR) IP option may enable illegal traffic to bypass a firewall [22]: An IP router that supports the LSRR option may replace the source IP address of an incoming packet with its own IP address. In this case, any filtering based on the source IP address of the packet that happens *after* the processing of IP options is ineffective. This has been exploited to bypass firewalls, eventually causing network operators to disable LSRR and router manufacturers to change their LSRR implementations.

Our tool would have uncovered this vulnerability. To verify that, we created a pipeline that includes an IP options element followed by a firewall, and we tried to prove that it satisfies the following property: "any packet whose source IP address is blacklisted by the firewall will be dropped." The tool responded that the property is not satisfied, and it provided an example packet that causes it to be violated: a packet with a blacklisted source IP address that carries the LSRR option.

6 Limitations

The key enabler and at the same time limitation of our work is that we focus on software dataplanes that follow a pipeline structure and also satisfy three other conditions (§3). The pipeline structure is a natural fit for dataplanes; most research prototypes are already written this way, and we know of at least one industry prototype as well. Favoring an already popular programming model is, in our opinion, a modest price to pay for verifiability. The other three conditions introduce overheads: existing code may need to be changed to satisfy them (but the resulting code is, in our opinion, easier to read and maintain); compared to their more dynamic counterparts, data structures that satisfy Conditions 2 and 3 typically require more memory (but trading off memory for verifiability is, in our opinion, worth considering).

We currently handle only two specific, simple forms of mutable private state. As stated earlier, we do not expect to be able to completely remove this limitation, but we do expect to expand the range of state-manipulation patterns that we can formally reason about.

Our approach is applicable to packet-processing platforms where each packet is handled by a single processing core and different cores never need to synchronize. We focused on such platforms, because there is compelling evidence that they lead to better performance (by minimizing the number of compulsory cache misses) [16]. We would need new results in order to verify platforms where different cores contend for access to the same data structures.

7 Related Work

Our work is feasible because of advances in program analysis tools for C/C++ code, from Verisoft [18] to modern model checkers [33,39] and tools based on symbolic execution [10,11,13,20]. These tools target general code, so they cannot typically construct complete and sound proofs (which is what we want). Instead, they try to increase line coverage or identify buggy paths *without* having to reason about all the paths of the analyzed program (whereas we want to reason about *all* feasible paths of the analyzed pipeline). There exist tools that prove properties of real programs, but, to the best of our knowledge, they are tailored to specific domains other than dataplanes; notable examples are Astrée [8], SLAM [5, 6], and Terminator [14].

Compositionality has been leveraged before to address path explosion, in compositional dynamic test generation [19] and follow-on work [4,21]. The particular tools evaluated in these proposals use "top-down" composition: when symbolically executing a program, encountering a function triggers the construction of a summary (logical representation) of that function in the context of its caller. This makes sense, because—to the best of our understanding—the goal of this line of work is to maximize line coverage with as little work as possible (ideally, hit each program statement exactly once). We use "bottom-up" composition: we first compute context-free summaries of all elements, then we compose them as necessary to reason about the entire pipeline.

Verification techniques have been used before to debug or verify networked systems (but not dataplanes): Musuvathi and Engler adapted the CMC model checker to test the Linux TCP implementation for interoperability with the TCP specification [38]. Bishop et al. contributed a formal specification of TCP/IP and the sockets API, and they tested existing implementations for conformance to their specification [7]. Killian et al. contributed new algorithms for finding liveness bugs in systems like Pastry and Chord [30]. NICE finds bugs in OpenFlow applications [12]. SOFT tests OpenFlow switches for interoperability with reference implementations [35]. Guha et al. contributed "the first machine-verified [Software Defined Networking] controller" [23].

Ennals et al. contributed a new language for packet-processing applications [17]. The goal of that language was to simplify the "compilation of high-level programs to the distributed memory architectures of modern Network Processors." The proposed language ensured that no two threads referenced the same packet, which is akin to our requirement that no two pipeline elements have access to the same packet.

Finally, an earlier version of this work was presented in a workshop paper [15]. That paper reported the feasibility of proving crash-freedom and bounded-execution only for stateless pipelines; it did not include a scalability analysis (§5.2) or report on bugs found using our approach (§5.3).

8 Conclusions

We presented a verification tool that takes as input a software dataplane and proves that it does (or does not) satisfy properties like crash-freedom, bounded-execution, and filtering. Proving such properties for general software faces fundamental challenges, unsolvable with existing tools; we sidestepped them by applying existing ideas (symbolic execution and compositionality), and combining them with domain specifics of packet-processing code (most importantly, that it is structured as a pipeline of elements that do not exchange mutable state outside the packet itself and its metadata). We evaluated our tool on stateless and two simple stateful Click dataplanes; we were able to perform complete and sound verification of these pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool failed to complete the same task within several hours.

Acknowledgments. We are grateful for the help offered by Stefan Bucur, George Candea, Vitaly Chipounov, Johannes Kinder, Vova Kuznetsov, Christian Maciocco, Dimitris Melissovas, David Ott, Iris Safaka, Simon Schubert, and Cristian Zamfir, as well as our shepherd, Brighten Godfrey, and the anonymous reviewers. This work is supported by an Intel grant and a Swiss National Science Foundation grant.

References

[1] Meraki. http://meraki.cisco.com.

[2] Vyatta Hardware Appliances. http://www.vyatta.com/solutions/physical/appliances.

[3] Intel RFP Announcement: SDN Extensions for Programmable Data Services, 2012.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. of the ACM EuroSys Conference*, 2006.

[6] T. Ball and S. K. Rajamani. SLAM: Debugging System Software via Static Analysis. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2002.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *Proc. of the ACM SIGCOMM Conference*, 2005.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. MinÃl’, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[9] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[10] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[11] C. Cadar and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM Conference on Computer Communication Security (CCS)*, 2006.

[12] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1), 2012.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[15] M. Dobrescu and K. Argyraki. Toward Verifiable Software Dataplanes. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[17] R. Ennals, R. Sharp, and A. Mycroft. Linear Types for Packet Processing. In *European Symposium on Programming*, 2004.

[18] P. Godefroid. Model Checking for Programming Languages Using Verisoft. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 1997.

[19] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2007.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[21] P. Godefroid, A. Nori, S. Rajamani, and S. D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2010.

[22] F. Gont, R. Atkinson, and C. Pignataro. Recommendations on Filtering of IPv4 packets Containing IPv4 Options. `http://tools.ietf.org/html/draft-ietf-opsec-ip-options-filtering-05#section-4.3`.

[23] A. Guha, M. Reitblatt, and N. Foster. Machine-Verified Network Controllers. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[24] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proc. of the IEEE INFOCOM Conference*, 1998.

[25] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proc. of the ACM SIGCOMM Conference*, 2010.

[26] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[27] P. Kazemian, M. Chang, H. Zeng, S. Whyte, G. Varghese, and N. McKeown. Real Time Network Policy Checking using Header Space Analysis. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[28] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[29] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[30] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[32] E. Kohler, R. Morris, B. Chen, J. Jannoti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[33] D. Kroening, E. Clarke, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[34] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[35] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proc. of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[36] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. of the ACM SIGMETRICS Conference*, 2010.

[37] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proc. of the ACM SIGCOMM Conference*, 2011.

[38] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[39] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking.

In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[40] B. Raghavan, T. Koponen, A. Ghodsi, M. Casado, S. Ratnasamy, and S. Shenker. Software Defined Internet Architecture. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[41] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[42] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static reachability Analysis of IP Networks. In *Proc. of the IEEE INFOCOM Conference*, 2005.