# Toward a Verifiable Software Dataplane

Mihai Dobrescu and Katerina Argyraki
School of Computer and Communication Sciences
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 concern of introducing buggy or under-performing code into the network. We explore whether it is practical to formally prove that a software dataplane satisfies key properties that would ensure smooth network operation. In general, proving properties of real programs remains an elusive goal, but we argue that dataplanes are different: they typically follow a pipeline structure that enables our proposed approach, in which we verify pieces of the code in isolation, then compose the results to reason about the entire dataplane. We preliminarily demonstrate the potential of our approach by applying it on simple Click pipelines and proving that they are crash-free and execute a bounded number of instructions. This takes on the order of minutes, whereas a general-purpose state-of-the-art verifier fails to complete the same task within 12 hours.

## Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking; D.2.4 [**Software Engineering**]: Software/Program Verification

## General Terms

Design, Performance, Reliability, Verification

## Keywords

Dataplane Verification, Programmable Routers

## 1. INTRODUCTION

Software dataplanes are emerging as an alternative to traditional hardware switches and routers. In the last five years, the industry and research communities have produced a rapid succession of software prototypes and products that perform IP forwarding [12, 17], packet classification [24], encryption [18], or application acceleration [2] at line rates of tens of Gbps, which a few years back were achievable only by specialized hardware.

The main advantage of software dataplanes is flexibility: They make it possible to significantly cut network provisioning costs by dynamically allocating packet-processing tasks to network devices [27]; or to turn the Internet into an evolvable architecture that adapts to the needs of its users and operators, by continuously updating the functionality of devices located at strategic network points [26]. Industry is also listening: Intel recently announced its interest in the development of a "composable software data plane," which will enable dynamic composition of different packet-processing elements [4].

Such flexibility is set against the concern that software dataplanes will introduce buggy and under-performing code in the network. When we presented earlier results on software packet processing to router manufacturers, most of their questions were not about feasibility, but about the cost of programmability, e.g., "if we allow third-party code to be added into the dataplane, there will be functionality and performance bugs, and who will absorb the extra cost of customer support"? This concern is justified, given the history of transitioning from hardware to software implementations in other domains (e.g., the consumer electronics or car industry), where software has dramatically increased the speed with which new features are implemented, but there is a perception that it has also led to products that are more fragile and more likely to misbehave [3, 9].

Is it practically feasible to prove that a software dataplane satisfies key properties that would ensure smooth network operation? Or must we accept that they will always be less predictable than their hardware counterparts? When we say "software dataplane," we mean a directed graph of distinct packet-processing elements (e.g., an IP lookup element, a filtering element, an IP options element) that are combined into a pipeline using a framework like Click [22].

We explore the feasibility of an automated verification tool that takes as input either the source code or the executable binary of a software pipeline and proves that the pipeline does (or does not) satisfy a target property. We care for properties that, in the case of hardware dataplanes, are either taken for granted or can be proved using practical techniques [19–21, 25, 28]: "crash freedom," which means that no packet sequence can cause the dataplane to stop executing; "bounded latency," which means that no packet experiences more than a known, reasonable amount of latency; or higher-level reachability properties, e.g., "any packet with destination IP address $X$ will never be dropped unless it is malformed."

Each proof should hold for any sequence of incoming packets. If the tool proves that a target property is not satisfied, it should provide example packet sequences that cause the property to be violated (e.g, if it cannot prove that a pipeline will never crash, then it should provide example packet sequences for which it does crash). Certain properties (like crash-freedom and bounded-latency) should be proved for any pipeline configuration, while others can only be proved for a specific configuration. E.g., if we want to prove that a pipeline will never drop a well-formed packet with destination IP address $X$, such a proof is meaningful only for a specific forwarding and filtering table.

In general, automatically proving properties of real programs (unlike searching for bugs) remains an elusive goal for the systems community, especially for programs that are written in a low-level language like C++ and consist of more than a few hundred lines of code. Despite promising results from the programming-language community, we are still far from the point where we can input a program into a verifier and obtain a proof—even for simple programs like the UNIX coreutils and simple properties like crash-freedom [7]. This is due to the structure of real programs, where different pieces of code often share access to the same state, resulting in dependencies that are hard—often infeasible—to reason about, even with the help of sophisticated verification tools.

Our thesis is that, unlike general programs, software dataplanes have a special structure that is particularly amenable to verification: they typically consist of distinct packet-processing elements that communicate with each other through a well-defined, narrow interface and do not share mutable state. This special structure has the following implication: it is possible to reason about the behavior of the entire dataplane without treating it as a single piece of code; instead, we can reason about each element in isolation and efficiently compose the results to reason about the entire dataplane. We will argue that this can be leveraged to sidestep certain fundamental problems faced by software verification and enable the practical construction of proofs.

For general programs, verifiability and performance are typically competing goals: a low-level language like C++ is typically good for performance but makes verification hard, while a language like Haskell may make it easier to verify certain properties but harder to achieve good performance. For software dataplanes, it does not have to be this way: we will argue that we can write them in a way that preserves performance and enables verification. The key question then is: what defines a "software dataplane" and how much more restricted is it than a "general program"? I.e., how much do we need to restrict our dataplane programming model so that we can achieve verifiability without giving up on performance?

## 2. MOTIVATION AND SETUP

**Use Cases.**

The most obvious users of a verification tool for software dataplanes would be the *developers of packet-processing code*: Reasoning about the behavior of a packet-processing element $E$ is hard enough; reasoning about what $E$ will do when part of a bigger pipeline is even harder. Our tool would help by checking, for any given design or implementation choice in $E$, what would be the impact on one (or more) bigger pipelines that include $E$. It would also provide concrete examples of packet sequences that lead to a segmentation fault, a kernel panic, a division by $0$, a failed assertion, or a counter overflow. Today, developers are forced to perform extensive testing before release; our tool would make them more productive by focusing their attention on the most relevant test cases.

A second set of users would be *network operators*: When a new, interesting type of packet processing becomes available (e.g., a new form of intrusion detection or application acceleration), an operator may want to include this as a new element $E$ in the pipelines running on its network devices. Today, the operator has no effective way of assessing the consequences of such an upgrade on the network as a whole; at best, it can test for a while and deploy widely after gaining some level of confidence that there will be no dire consequences. As a result, trying out new packet-processing software is time-consuming and potentially dangerous. A dataplane verification tool would change this by providing a way to check what would be the impact on the currently running pipelines, e.g., what would be the maximum increase in latency or energy consumption that the new element would introduce. Such information would enable faster and safer deployment, ultimately making operators less conservative in trying out new packet-processing software.

A third—and perhaps most interesting—use case targets future *markets for packet-processing elements* that are similar to today's app markets (Apple AppStore, Google Play, etc.) Such markets would allow network operators to "go shopping" for new packet-processing elements that they can then drop into the dataplanes of their network devices. Our tool would help by enabling the app-market operator to formally certify that the desired element will not disrupt the customer's pipeline.

**Our Approach.**

We are exploring an approach that consists of two steps: The first one processes each pipeline element in isolation and identifies "suspect" packet sequences that may cause the target property to be violated. This can be done efficiently and completely (without false-negatives), assuming each element typically consists of relatively short and simple code (more on this below). However, the results of the first step may not be sound (may have false-positives), because it does not take into account the interactions between different elements. False-positives are eliminated in the second step, which examines each suspect packet sequence and verifies whether it can indeed cause the target property to be violated given the interactions between all the involved elements.

This would not work for any program; we believe (and have preliminary evidence that) it works for software pipelines because these consist of relatively short and simple elements that do not share mutable state. Because of this special structure, we can quickly eliminate the majority of packet sequences as harmless (first step), which leaves a significantly smaller number of potentially harmful packet sequences that need to be examined in more detail (second step).

We rely on fundamental ideas contributed by the programming-language community, but we cannot "simply adjust" existing tools to solve our problem: We use symbolic execution [7, 14], a form of path-sensitive dynamic program analysis, to identify suspect packet sequences in a single element; akin to compositional test generation [5, 13, 15], we process each element once, even if it may be called from different points in the pipeline. These ideas have existed for a long time, however, they have been applied toward different goals (increasing line coverage and/or finding bugs), so we cannot use existing incarnations of these ideas to solve our problem.

**Symbolic Execution.**

A program can be represented as a tree, where each path from the root to a leaf corresponds to a different instruction sequence, and each internal node corresponds to a branching point (Fig. 1). During normal execution of the program, each variable is assigned a concrete value, and only a single path of the tree is executed. In contrast, during symbolic execution (from now on "symbex," for brevity), a variable may be *symbolic*, i.e., assigned a set of values that is specified by an associated constraint. E.g., a symbolic integer $x$ with associated constraint $x > 2 \land x < 5$ is the set of concrete values $x = \{3, 4\}$. A symbex engine can take a program, make the program's input symbolic, and execute all the paths that are feasible given this input.

Consider the toy program 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, while the "symbolic state" maps each variable to its current value on this path. E.g., at the end of path $p_2$, the path constraint is $C = (in \geq 0 \land in < 10)$, while the symbolic state is $S = \{out \mapsto 10\}$; at the end of path $p_3$, the path constraint is $C = (in \geq 10)$, while the symbolic state is $S = \{out \mapsto in\}$.

**Proof by Execution.**

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. E.g., suppose we want to prove that the program in Fig. 1 never executes more than 10 instructions. We can do this by symbexing the program with a symbolic input $in$ that may take any value, executing all three feasible paths, and verifying that none of them includes more than 10 instructions. Such a proof assumes that the symbex engine itself is correct and that the hardware operates according to its specifications.

```
out Program ( in ):
  assert $in \geq 0$
  if $in < 10$ then
    out ← 10
  else
    out ← in
  end if
  return out
```

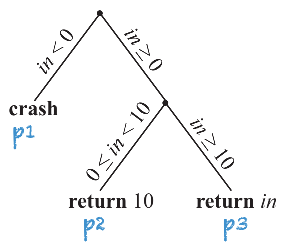


**Figure 1: A toy program and its execution tree.**

By constructing proofs in this manner, we can automatically determine all the problematic inputs that prevent us from completing the proof. E.g., suppose we try to prove that the program in Fig. 1 never crashes. We symbex the program with a symbolic input $in$ that may take any value, and we discover three feasible paths: one for $in < 0$, one for $0 \leq in < 10$, and one for $in \geq 10$. We can argue that the last two paths cannot cause the program to crash, however, the first path ends with a failed assertion—a crash. So, we have failed to prove that the program satisfies the target property, but we have also uncovered all the input values ($in < 0$) that cause the property to be violated.

**Proving $\neq$ Bug Finding.**

Proof by execution can be rarely used in practice, because of path explosion [6]: 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. Even small programs, like UNIX coreutils, have an intractable number of feasible paths because of their use of essential libraries like libc [7].

Because of this challenge, symbolic execution can rarely be used for proofs, even though it is often (and successfully) used for identifying good input values for testing, as well as input values that are likely to cause bugs to manifest. We should clarify that 100% line coverage (i.e., exercising each line of code at least once) is not the same as exploring 100% of a program's feasible paths. In fact, sophisticated tools may achieve good line coverage for small and simple programs, yet explore only a small fraction of the feasible paths. For instance, when Klee [7] symbolically executes UNIX coreutils like nice or cat, it achieves more than 70% line coverage, but executes less than 1% of the feasible paths [23]. This is fine when the goal is to discover interesting paths (e.g., to uncover bugs), but not when the goal is to exercise all feasible paths (to prove properties).

## 3. OUR PROPOSAL

We observe that symbolic execution is a good fit for packet-processing pipelines, because their special structure can help sidestep path explosion: Intuitively, the fact that there are no state interactions between different pipeline elements (other than one passing a packet to another) makes it feasible to reason about each element in isolation, then compose the results to reason 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 $k$ such elements has roughly $2^{k \cdot n}$ paths. Verifying 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 $2^{k \cdot n}$ to $k \cdot 2^n$.

**Pipeline Structure.**

We consider packet-processing pipelines where each element may access the following three types of state:

*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.

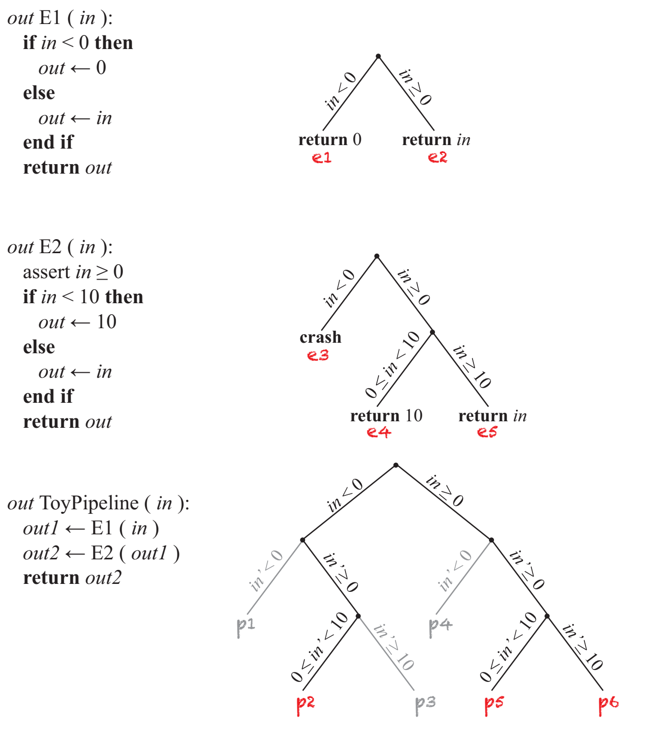
*Private state* is owned by one element and never changes ownership. It can be read or written only by its owner, and it persists across the processing of multiple packets. A typical example is a flow table in a NetFlow element, or a map in an element that performs Network Address Translation (NAT).

*Static state* can be read by any element but not written by any element. This state is immutable as far as the 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 require additional synchronization, e.g., through locking, a typical cause of contention and unpredictable performance. Pipelines that are created with Click conform to this structure, and these arguably constitute the majority of research prototypes. Similar information about industrial prototypes is not typically disclosed, but we know of at least one company that uses Click [1].

**Pipeline Decomposition.**

We now describe our two-step verification process in more detail: First, we cut each pipeline path into small segments (defined below). In Step 1, we capture the outcome of each segment symbolically and, once we've done so, we never need to execute that segment again, because we've distilled it into its "essence": how that segment transforms state. In



```
out E1 ( in ):
  if in < 0 then
    out ← 0
  else
    out ← in
  end if
  return out
```

```
out E2 ( in ):
  assert in ≥ 0
  if in < 10 then
    out ← 10
  else
    out ← in
  end if
  return out
```

```
out ToyPipeline ( in ):
  out1 ← E1 ( in )
  out2 ← E2 ( out1 )
  return out2
```

**Figure 2: A toy pipeline that consists of two elements.**

Step 2, we cast the proof process as a search (in isolation, at the element level) for segments that might violate the desired property, and then simply check whether, once we assemble elements into the desired pipeline, any of these potential violations are still feasible. This search is complete, in that we do not miss any potential violations, and sound, in that we do not introduce behavior that the code does not have; the net result is sound and complete, thus being a correct proof.

We illustrate our setup in Fig. 2: We represent the pipeline as a tree that consists of subtrees, one per packet-processing element; a subtree representing element $E_i$ will appear multiple times in the tree, once for each feasible path that may lead to $E_i$. The input $in$ corresponds to a newly received packet; we assume that this may contain anything, i.e., in symbex terminology, $in$ is a symbolic bit vector. We define a *segment* to be a complete path through one element, and a *path* to be a complete execution path through an entire pipeline; a path through a pipeline of $k$ elements is a concatenation of $k$ segments.

In Step 1, we verify each element individually: We symbex the element assuming unconstrained symbolic input, and we conservatively tag as "suspect" all the feasible segments that may cause the target property to be violated. E.g., if the target property is that the pipeline never crashes, then every segment that leads to a crash is tagged as suspect. As a by-product, we obtain the constraint $C$ and symbolic state $S$ at the end of every feasible segment.

If this step does not yield any suspect segments for any element, we are done: the pipeline satisfies the target property.

E.g., if none of the elements ever crashes for any input, we have proved that the pipeline will never crash.

A suspect segment does not necessarily mean that the pipeline can violate 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. E.g., in Fig. 2, if we consider element $E_2$ alone, segment $e_3$ is feasible, and it causes the element to crash; however, in a platform where $E_2$ always follows $E_1$, segment $e_3$ becomes infeasible, and the platform never crashes. In program analysis terminology, when we explore all the feasible segments of each individual element assuming arbitrary input, we are "overapproximating," i.e., we are executing some elements with inputs that they would never see when part of the pipeline we are aiming to verify.

In Step 2, we check if the suspect segments could violate the target property when part of the pipeline: First, we construct each potential path $p_i$ that includes at least one suspect segment; $p_i$ is a sequence of segments $e_j$. Next, we stitch together the path constraint for $p_i$ and the resulting symbolic state based on the constraints and symbolic state of its constituent segments (that we have already obtained in Step 1). Finally, we determine whether path $p_i$ is feasible (based on its constraint) and whether it violates the target property (based on its symbolic state), without ever actually executing $p_i$.

E.g., here is how we prove that the pipeline in Fig. 2 does not crash:

**Step 1**:

1. We symbex $E_1$ assuming input $in$ can take any integer[1] value. We collect constraints $C_1$ and $C_2$, and symbolic state $S_1$ and $S_2$, for its segments $e_1$ and $e_2$:
   - $C_1(in) = (in < 0)$, $S_1(in) = \{out \mapsto 0\}$.
   - $C_2(in) = (in \geq 0)$, $S_2(in) = \{out \mapsto in\}$.
2. We symbex $E_2$ assuming input $in$ can take any integer value. We collect the following constraints and symbolic state for its segments $e_3$, $e_4$, and $e_5$:
   - $C_3(in) = (in < 0)$, $S_3(in) = \{crash\}$.
   - $C_4(in) = (in \geq 0 \wedge in < 10)$,
     $S_4(in) = \{out \mapsto 10\}$.
   - $C_5(in) = (in \geq 10)$, $S_5(in) = \{out \mapsto in\}$.
3. Segment $e_3$ may cause a crash, so we tag it 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
$$\begin{aligned} C_{p_1}(in) &= C_1(in) \wedge C_3(\ S_1(in)\ [out]\ ) \\ &= C_1(in) \wedge C_3(\ 0\ ) \\ &= (in < 0) \wedge (0 < 0). \end{aligned}$$
3. Path $p_1$'s constraint always evaluates to false, hence $p_1$ is infeasible, i.e., there is no way the pipeline could execute path $p_1$. Similarly, we establish that $p_4$ is infeasible.
4. Since all the feasible paths consist of non-suspect segments (that never crash), the platform never crashes.

[1]We use integer input for illustration purposes. In reality, the input to each element is a symbolic bit vector.

**Element Verification.**

Our approach assumes that symbexing each element in isolation is feasible. So far, we have been able to symbex all the elements we have experimented with (reported below), but not before resolving two significant challenges: loops and mutable data structures. For lack of space, we only outline the main ideas behind our techniques.

The main challenge we encountered was the presence of loops (e.g., the one executed when processing IP options), which create many paths even within a single element. E.g., if we symbexed (in isolation) the IP options element that comes with Click, we roughly estimated that we would have to execute millions of segments, which would take months to complete. To resolve this, we reuse the idea of decomposition, but apply it at a different level: If a loop has $t$ iterations, we view it as a sequence of $t$ "mini-elements," each one corresponding to one iteration of the loop. In the "pipeline decomposition" part, we said that we symbex each element in isolation, then compose the results to reason about the entire pipeline. Similarly, we symbex one mini-element in isolation, then use the results to reason about the entire loop.

The other challenge we encountered was the presence of mutable data structures, e.g., a hash table for per-flow statistics or a map for network address translation (NAT). Symbex engines still lack the semantics to deal with data structures in a scalable manner; symbexing an element that contains, e.g., access to an array with 1 million entries will cause a symbex engine to essentially branch into 1 million different segments, independently from the array content or the logic of the code that uses the returned value. To resolve this, we separate the verification of a stateful element into two distinct parts: (1) verification of the code that accesses data structures and (2) verification of the rest of the code.

To verify the rest of the code, we model each data structure as a key/value store that supports only a read and a write function, and we assume that a read may return either a value that was previously written in the data structure or a default value. First, we symbex the element assuming that a read to a data structure may return *any* value (the variable that stores the return value is marked symbolic and unconstrained), and we identify all the "bad" values that, if read from the data structure, will cause the target property to be violated. Second, we go back and check whether any input to the element may have caused any of these bad values to be written to the data structure in the first place.

To verify code that accesses data structures, we are taking a pragmatic approach: use either data structures that have been previously verified by experts [29], or data structures that can be efficiently verified using static analysis. For instance, verifying code that accesses arrays can be efficiently done using static analysis, due to the simplicity of array semantics. Fortunately, most packet-processing functionality can be implemented using pre-allocated arrays, and this is not by accident: Packet-processing elements typically maintain their private state in pre-allocated hash tables/maps that provide $O(1)$ lookup time, so that they can access it at line rate; these data structures can be easily implemented as array chains. Even longest prefix matching (often implemented on tries) is amenable to an array-based implementation, as long as we are willing to throw memory at the problem [16].

**Preliminary Results.**

To test the potential of these ideas, we applied them on packet-processing pipelines developed with Click, to answer two questions: (a) is there any input that can make the pipeline crash? (b) which is the maximum number of instructions that each pipeline may ever execute and which input causes it? We used a Xeon-based server running SMPClick [10]. In all the pipelines we tried, packets are generated by a "generator" element and dropped by a "sink" element; what we verify is all the packet-processing code between the two. We used S2E [11] as our underlying symbex engine.

We first verified pipelines that combine elements from the default Click IP-Router configuration (Classifier, EthEncap/EthDecap, CheckIPhdr, IPlookup, DecTTL, IP options). We proved that any pipeline that consists of these elements will not crash for any input. We also proved that the longest pipeline (that consists of all these elements) executes up to about 3600 instructions per packet, and we also identified the packet that yields this maximum result.

For the longest pipeline, our verification time was about 18 minutes; in contrast, when we fed the same code to the symbex engine (without using pipeline decomposition or any of the other presented ideas), verification did not complete within 12 hours.

We are currently experimenting with pipelines that contain more sophisticated elements, e.g., that collect NetFlow-style statistics or perform NAT functionality.

The pipelines that we verified may be conceptually simple, but they include processing that is challenging from a verification point of view: large numbers of branching points, loops, and mutable data structures. The fact that we were able to verify them within minutes or tens of minutes constitutes encouraging evidence.

## 4. DISCUSSION

**How about model checking or a special language?** We have not precluded the option of using model checking to verify individual elements, e.g., it may be a good alternative to static verification of data structures. For certain low-level properties, using a special language or environment guarantees a priori that the property holds, e.g., writing a program in Java or running it in a sandbox guarantees that the program will never perform illegal memory accesses. For most properties, however, one would have to develop special languages, whereas we are interested in verifying software dataplanes written in a popular language like C++ and optimized for high performance.

**How much are we giving up for verifiability?** We are restricting ourselves to pipelines where different stages do not share mutable state. We think this is necessary, because state creates dependencies among the pieces of code that share it, and more dependencies are equivalent to more constraints, which increase verification time. Hence, controlling verification time comes down to restricting the range of code that accesses the same state. We think this restriction is acceptable, because pipeline states are mostly designed anyway to be independent from each other for performance reasons. Our claim then is that, in the context of packet-processing pipelines at least, state isolation per pipeline stage is reasonable to expect, since we mostly do it anyway for performance.

**What about non-verifiable elements?** There will always be some forms of packet processing that we will not be able to verify—perhaps deep-packet inspection fundamentally requires dynamically-growing data structures that are not statically verifiable. We are not advocating to reject such non-verifiable code; network operators will always have the option to deploy it, but they will have to be more conservative in doing so, as there will be no assurance about what that code will do to their network (which is what happens with *all* dataplane software today).

**Why not verify the control plane?** We are focusing on dataplane software verification, because we believe it is an equally worthy goal that has received no attention, perhaps because it is mistakenly considered easy. We should also clarify that we cannot use bug-finding tools for control-plane applications [8] to prove dataplane properties. These tools were designed to find bugs, not produce proofs, i.e., they cannot reason about all the paths of a software dataplane.

**Is this Active Networking?** No, because we are not considering code-carrying packets, and we are focusing on existing, popular languages like C++ and platforms like Click. But we are sharing a similar vision: of a programmable dataplane, where the operator "drops in" new packet-processing code without risking to destabilize network operation.

**Acknowledgments.** We would like to thank George Candea and Johannes Kinder for enlightening discussions, Vitaly Chipounov for guiding us through using S2E and even extending S2E output in response to our requests, and Stefan Bucur, Vova Kuznetsov, Simon Schubert, and Cristian Zamfir, as well as the anonymous reviewers, for helping us make this paper better.

## 5. REFERENCES

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

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

[3] Cars and Software Bugs. `http://www.economist.com/blogs/babbage/2010/05/techview_cars_and_software_bugs`, 2010.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proc. of the USENIX Security Symposium*, 2011.

[10] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocesor PC Router. In *Proc. of the USENIX Annual Technical Conference*, 2001.

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of the International Conference on Computer Aided Verification (CAV)*, 2008.