Sandboxing Data Plane Programs for Fun and Profit

Miguel Neves
UFRGS

Kirill Levchenko
UC San Diego

Marinho Barcellos
UFRGS

ABSTRACT

This paper describes the design and implementation of a general-purpose compile-time sandbox for P4 data plane programs. Our mechanism allows a supervisor to interpose on another program's interaction with the forwarding device. The sandboxing technique we use provides also a powerful new program structuring model, allowing a data plane developer to combine crosscutting program modules in a safe way. To demonstrate the capabilities of our construct, we describe the implementation of a data plane security kernel that enforces end host isolation policies on top of a programmable data plane.

CCS CONCEPTS

• **Networks** → **Programmable networks**; • **Security and privacy** → *Network security*;

KEYWORDS

P4; Sandbox; Programmable data plane

ACM Reference format:
Miguel Neves, Kirill Levchenko, and Marinho Barcellos. 2017. Sandboxing Data Plane Programs for Fun and Profit. In *Proceedings of SIGCOMM Posters and Demos '17, Los Angeles, CA, USA, August 22–24, 2017,* 2 pages.
https://doi.org/10.1145/3123878.3132001

1 INTRODUCTION

The ability to program the data plane enables network operators to quickly deploy new protocols, customize network behavior, and develop innovative services. Unfortunately, the current data plane programming environment consists of a single monolithic data-plane program interacting directly with the underlying platform. This makes it difficult to establish guarantees about the program. In this work, we propose a software sandboxing mechanism that allows one data plane program to *supervise* another by interposing on its interaction with the system (e.g., controlling packet transformations or accesses to built-in functions). Our mechanism automatically transforms a supervisor and a user data plane program into a single program that guarantees:

(1) The user program cannot interfere with the supervisor,
(2) All incoming packets will pass through the supervisor (where they can be intercepted or modified), and
(3) Any system function called by the user program will be dispatched to the supervisor (which can block it or allow it to proceed).

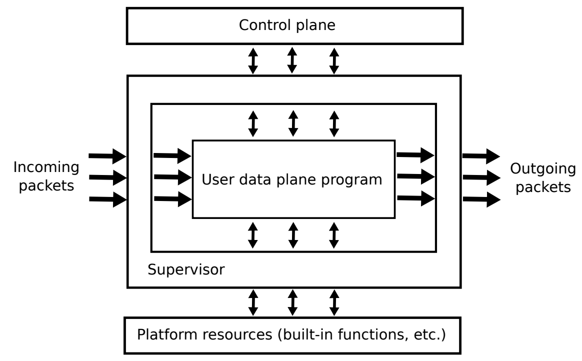

Figure 1: Data plane program sandbox high-level overview.

In effect, the supervisor program plays the role of an operating system by mediating access to system resources. However, our mechanism is implemented entirely as a target-independent compiler transformation and requires no hardware support. The program supervisor mechanism is particularly well-suited for data plane programs where one module needs to observe or modify the external interaction of another. It could be used, for example, for program diagnostics and tracing. In this paper, we describe its usage for implementing a security kernel that enforces end host isolation policies. The kernel ensures that a P4 implementation of VPN switching does not allow any information flow between distinct networks due to bugs in the programmable data plane.

2 PROGRAM SANDBOXING

Conceptually, the supervisor is a wrapper around the sandboxed program (see Figure 1), implementing callbacks for system-level events (e.g., a packet emission or a call for a built-in function) as well as for traps inserted into the packet processing pipeline. Our mechanism can provide a variety of capabilities useful for guaranteeing that a programmable data plane operates safely. Some examples include:

Packet header protection. This capability is useful for guaranteeing that a given header value will not be modified by the data plane program. Alternatively, some logic can be applied for deciding whether a modification is allowed or not (e.g., allow the modification only if the new header value is within a given range).

Trusted packet transformation. This capability ensures that a given transformation (e.g., a header insertion) will be correctly applied on a packet. For doing so, the sandbox deviates the flow of execution from the data plane program towards a supervision module, which contains the necessary logic. This logic can be essentially any packet processing task expressed in P4 or supported by the

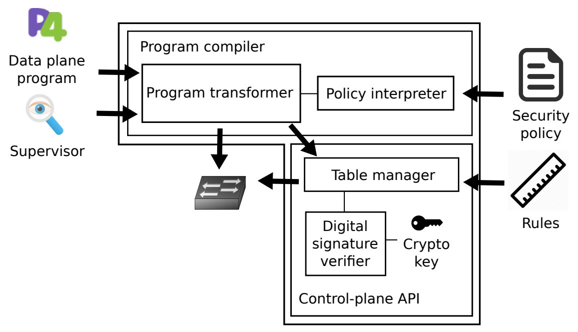

Figure 2: Prototype architecture.

target device. A security-enhanced control plane API is provided for those tasks that involve match-action tables and require some configuration from the control plane. Our API guarantees that only trusted sources can modify tables managed by our sandbox.

We are developing a prototype implementation for the proposed sandboxing mechanism. Figure 2 represents its architecture. A program compiler instruments the P4 data plane program with supervision modules responsible for enforcing a policy of interest. Policies are expressed using a simple scripting language and interpreted for configuring each supervisor module.

The program compiler generates instructions to configure both the target and our security-enhanced control plane API. This API contains a table manager responsible for parsing each forwarding rule and checking whether it can manipulate the destination table or not. It uses a digital signature verifier for this purpose, which guarantees only signed rules can manipulate tables involved in the sandboxing logic (i.e., used by the supervisor). This architecture allows our sandbox to be completely target agnostic, as we do not involve any low-level element from the programming pipeline. We are extending the P4-16 Reference Compiler [1] and the P4Runtime tool [2] provided by the P4 Language Consortium for its implementation. [1] [2]

3 USE CASE: SECURITY KERNEL

To demonstrate the general applicability of our sandboxing mechanism, we describe how it can be used for enforcing isolation properties in a security kernel for programmable networks. Ensuring properties such as end host isolation is a hard task. Recently, network verification techniques have appeared as a promising way of guaranteeing that a configuration results in a correct network behavior. A programmable data plane, however, makes this task much more challenging since current techniques either assume a fixed data plane logic or encompass some basic flexibility to the detriment of performance [3].

In Figure 3, we show how the capabilities provided by our sandboxing mechanism can be used for enabling the correct enforcement of isolation properties in large scale networks. It is based on the *stag* application proposed in [3], which isolates hosts based on security classes identified using special tags (similar to VPN switching) and represents the instrumentation of a sandboxed P4 program. Red code indicates our sandboxing logic. In summary, protected copies of relevant headers are created when the packet arrives in the forwarding device – steps 1 and 2, guaranteeing its state can be kept safe. Saved state can be restored at the end of the packet processing pipeline upon request – step 5. This is important for guaranteeing that security tags will not be erroneously modified by faulty programs. During the packet processing inside the pipeline, trusted packet transformations can be applied (with the guarantee that they cannot be circumvented) – step 4 for inserting/removing tags, and any attempt to access restricted resources in the device (e.g., a built-in function that drops the packet) can be mediated – step 3.

```
//target function imported from model description
extern void mark_to_drop();

struct headers {
  ethernet_t ethernet;
  stag_t stag;
  ethernet_t _PROTECTED_ethernet;        1
  stag_t _PROTECTED_stag;
}

parser ParserImpl(packet_in packet, out headers hdr) {
  state parse_ethernet {
    packet.extract(hdr.ethernet);
    //copy protected packet state when it enters in the device
    hdr._PROTECTED_ethernet = hdr.ethernet;
    transition accept;                          2
}}

control hook_1(inout headers hdr){
  /* may contain some control logic */     3
  apply { mark_to_drop(); }}

control controlFlow(inout headers hdr) {
  hook_1() process_hook_1;
  apply{
    //cannot circumvent trusted transformation
    //(i.e., stag insertion/removal)
    process_trusted_transformation.apply(hdr);   4
    process_switching.apply(hdr);
    process_hook_1.apply(hdr); //check method call policy
}}

control hook_2(inout headers hdr){              5
  /* may contain some control logic */
  apply { hdr.ethernet = hdr._PROTECTED_ethernet; }}

control DeparserImpl(packet_out packet, inout headers hdr){
  hook_2() process_hook_2;
  apply {
    process_hook_2.apply(hdr); //apply header protection policy
    packet.emit(hdr.ethernet);
}}
```

Figure 3: P4 program instrumentation for the *stag* example. Only most relevant parts are shown.

Acknowledgements. This work was supported in part by CNPq (460322/2014-1) and a Microsoft Azure for Research Grant.

REFERENCES

[1] 2017. P4 Language Consortium. *P4 Reference Compiler.* (2017). https://github.com/p4lang/p4c.
[2] 2017. P4 Language Consortium. *P4Runtime.* (2017). https://github.com/p4lang/PI.
[3] Nuno P Lopes, Nikolaj Bjørner, Nick McKeown, Andrey Rybalchenko, Dan Talayco, and George Varghese. 2016. *Automatically verifying reachability and well-formedness in P4 Networks.* Technical Report.

[1] https://github.com/mcnevesinf/p4c
[2] https://github.com/mcnevesinf/PI