APPENDIX

A. IR Grammar

This section provides technical details on the intermediate representation (IR) grammar.

```
<IR> ::= {
  <SourceEntity> ::= <String>: {
    <DestinationEntity> ::= <String>: [
      <ProtocolFlow> ::= {
            "fid": <Integer>,
            "state": <Boolean>,
            "dependency_fid": <Integer>,
            "protocol": <String>,
            "protocol-header-1": <String>,
            "protocol-header-2": <String>,
            ...,
            "protocol-header-n": <String>,
      },
      ...
    ]
  },
  <SourceEntity> ::= <String>: {
    <DestinationEntity> ::= <String>: [
     <ProtocolFlow> ::= { ... },
      ...
    ]
  },
  ...
}
```

Fig. 5. IR grammar.

Figure 5 illustrates the formal structure of the IR generated by the front end. The IR consists of multiple `<SourceEntity>` and `<DestinationEntity>` pairs, which represent unique end-system names or network addresses (such as IP or MAC addresses). Each source-destination pair is associated with an array of `<ProtocolFlow>` elements, where each element describes a distinct flow between the entities.

A `<ProtocolFlow>` element contains four key components: `<fid>`, `<state>`, `<dependency_fid>`, and `<protocol>`. The `<fid>` serves as a unique identifier for the flow across the entire IR. The `<state>` indicates whether the flow is stateful or stateless. If the flow relies on the state of another flow for packet propagation, the `<dependency_fid>` holds the `fid` of that dependent flow, which is thus stateful. The `<protocol>` defines the protocol stack relevant to the flow. Within this stack, the protocol headers contain header-value pairs that correspond to the protocol layers specified in the `<protocol>` element.

B. Policy Insertion and Lookup Time Complexity

The insertion and lookup operations iterate through the encapsulated protocol headers, defined by protocol specifications and, therefore, can be up to a length of L, where L is the total number of headers in a protocol stack. Moving from one node to the next is done through the node's hash map, which has a lookup time of $O(1)$. For the policy lookup, each protocol header is checked against at most two potential paths (exact match and wildcard). Despite the dual path check (exact match and wildcard), the traversal depth remains L. The potential extra paths do not significantly impact the overall time complexity in terms of big-O notation because each level still processes a single property value. Therefore, both policy insertion and lookup have a time complexity of $O(L)$ where L is constant. This means that both operations are not impacted by the number of security policies (which can be significant for large networks). Note that while L may vary slightly depending on the protocol stack, it remains finite and bound by protocol specifications and standards.

C. Switch Stateful Packet Handling

In this section, we provide the algorithm pseudocode for stateful packet handling, which runs on a data plane switch.

Algorithm 1 Generated Stateful Packet Processing Code

Require: isStatefulFlow, stateIndex, isDependentFlow, dependencyStateIndex

```
1:  if isDependentFlow == 1 then
2:      flowStateRegister.read(dependentStateValue, dependencyStateIndex)
3:      if dependentStateValue == 0 then
4:          dropPacket()
5:      else
6:          if isStatefulFlow == 1 then
7:              flowStateRegister.write(stateIndex, 1)
8:          end if
9:      end if
10: else
11:     flowStateRegister.write(stateIndex, 1)
12: end if
```

D. Supporting New Protocols

As discussed in the main paper, our framework already supports well-known protocols such as Ethernet, IP, and TCP. However, organizations often develop custom internal communication protocols layered on top of existing standards.

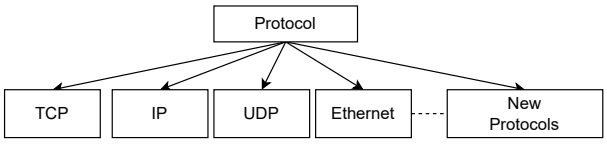

Fig. 6. Extending protocol support by subclassing the Protocol class.

To seamlessly integrate new protocols into the framework, one must extend the Protocol class (see Figure 6). The subclasses need to implement specific functions essential for policy orchestration and runtime management. These functions must define protocol header information, including header names and bit widths, as well as encapsulation details, such as how the protocol frame is identified within a protocol stack (i.e., protocol code). This information is then utilized to automatically generate the corresponding headers, parser states, and policy enforcement predicates.

Once the new protocol class is in place, the policy orchestrator will incorporate this information to fulfill the IR requirements, allowing the framework to support protocol stacks that include the newly added protocol.

E. Representations of P4 Language Components

As discussed in Section IV, we have developed object-oriented representations of P4 language components in the back end, adhering to the $P4_{16}$ language specifications [28]. These representations encapsulate the syntax, grammar, and semantics of the language, ensuring that they compile seamlessly into valid P4 programs. These representations serve as the foundation for generating the corresponding pipeline programs for the switches.

Fig. 7. Compilation of P4 representations to P4 code.

Figure 7 illustrates code snippets from the back-end processing alongside their corresponding generated components in P4. The first example demonstrates how a `Header` object, generated based on the UDP protocol frame specification (UDP is a subclass of the Protocol class, as detailed in Appendix D), is compiled into the corresponding P4 header. The second example shows how a `Parser` object is updated to parse UDP/IP packets and is subsequently compiled into P4 code. Finally, the third snippet depicts the compilation of a `Table` object, updated to meet an IR requirement, into the corresponding P4 table code.

F. Front End Compiler Performance

We assess the performance of the front-end compilation from the policy to the intermediate representation. The experiments reflect two scenarios: (1) the policy is authored using the fine-grained policy graphs; (2) the policy is authored using the conventional approach where administrators only define the CR graph, i.e., the subjects (client systems) are directly connected to their corresponding accessible services).

Fine-grained Approach. Figure 8(a) shows the combined RBAC and CR graphs used for the experiments. The three network services, namely the web service, streaming service, and the Git server, can communicate with each other using TCP and listen on port 5555. Entities assigned to the base task can access the web service (flows 1 and 2) using TCP at port 5051. Flow 2 permits the web service to communicate, given that the actor has originated the communication (flow 1). The green arrows denote flow dependencies. Flows 3 and 4 allow communication to the streaming service using UDP at port 5053 for entities connected to the media task. However, the policy requires the user to communicate with the web service first (e.g., for authentication) before being allowed to communicate with the streaming service. Flows 5 and 6 allow communication with the Git server for entities assigned to the Git task. Flows 5 and 6 are stateless flows. Purple dotted arrows denote task inheritance. Thus, actors assigned to the marketing role can access the streaming service and web service (due to permission inheritance). The engineering role can access the Git server as well as the web service. Actors are systems representing users or standalone devices (e.g., IoT).

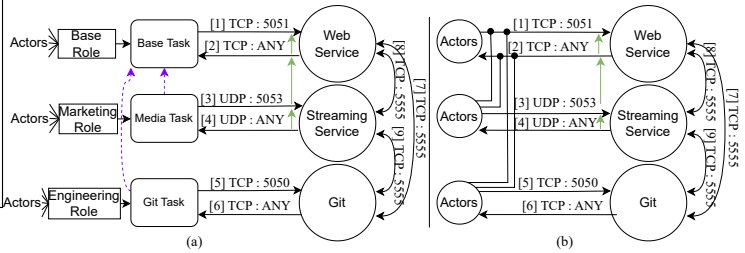

Fig. 8. (a) fine-grained AC policy graph. (b) Conventional AC policy graph.

We assigned an equal number of actors to each role, ranging from 30 to 390, resulting in a range of 90 to 1000 users. Figure 9 shows the performance measurement plots. Figure 9(a) shows the IR generation (in msec) time with respect to the number of actors, while Figure 9(b) shows the increase in the size of the intermediate representation for increasing numbers of actors. The results show that the compilation process performs reasonably well, given that generating the IR takes about 2 seconds for a network of 1K actors.

Conventional Approach. Figure 8(b) shows the policy graph when using the conventional approach for authoring the policy. The policy is equivalent to Figure 8(a); therefore, the IR generated has the same size as with the fine-grained policy. However, the IR compilation time performs better as the compiler does not have to traverse the various graphs as in the former scenario.

G. Policy Orchestration Performance

The policy orchestration evaluation focuses on four key aspects: (1) loading and processing the intermediate representation (IR), (2) inserting policies into the policy engine, (3) generating data plane programs relative to the number of

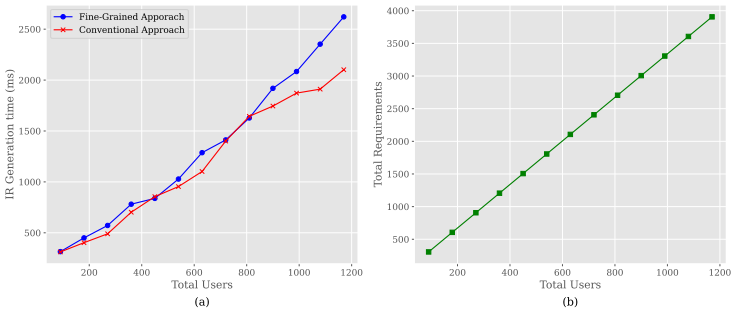

Fig. 9. (a) IR generation (in ms) time with respect to the number of actors. (b) Total number of requirements with respect to the number of actors.

TABLE IV
BACK-END EVALUATION RESULTS.

IR Size	Total Switches	IR Load/Processing (ms)	Building Switch Pipelines (ms)	Control Plane Orchestration (ms)	Average Policy Insertion Time (ms)	Total Back End Processing (ms)
1200	10	622	145	117	0.005	893
1800	20	953	135	190	0.01	1295
2400	30	1210	170	265	0.0004	1657
3000	40	1432	197	345	0.002	1987
3000	90	1157	209	726	0.003	2101

switches, and (4) orchestrating the control plane by updating it with the supported tables, actions, and policy predicates for each switch pipeline.

The primary factors influencing back-end performance are the size of the IR to be processed and the number of network switches, which determine the scale of the data plane. Table IV provides the performance results, showing that the total compilation time increases with the size of both the data plane and the IR. Notably, IR processing accounts for the majority of the back-end processing time. This is expected, given that the IR, which encapsulates the policies, is significantly larger than the data plane size, requiring more processing time. Data plane programs are updated incrementally to support the IR requirements, making the process highly efficient. For instance, processing an IR of 3K in size for a data plane comprising 90 switches took approximately 209 milliseconds to generate the pipeline programs. Policy insertion into the policy engine proves to be highly efficient, consistent with the theoretical analysis presented in Section V-1.