Programmable In-Network Security for Context-aware BYOD Policies

Qiao Kang
Rice University

Lei Xue
The Hong Kong Polytechnic University

Adam Morrison
Rice University

Yuxin Tang
Rice University

Ang Chen
Rice University

Xiapu Luo
The Hong Kong Polytechnic University

Abstract

Bring Your Own Device (BYOD) has become the new norm for enterprise networks, but BYOD security remains a top concern. Context-aware security, which enforces access control based on dynamic runtime context, is a promising approach. Recent work has developed SDN solutions to collect device contexts and enforce access control at a central controller. However, the central controller could become a bottleneck and attack target. Processing context signals at the remote controller is also too slow for real-time decision change.

We present a new paradigm, *programmable in-network security* (Poise), which is enabled by the emergence of programmable switches. At the heart of Poise is a novel security primitive, which can be programmed to support a wide range of context-aware policies in hardware. Users of Poise specify concise policies, and Poise compiles them into different configurations of the primitive in P4. Compared with traditional SDN defenses, Poise is resilient to control plane saturation attacks, and it dramatically increases defense agility.

1 Introduction

BYOD refers to the practice where enterprise employees could use privately owned tablets, phones, and laptops at work [32]. This practice has become the new norm in many organizations [7, 13, 14, 17, 23, 29], and its market is projected to grow and exceed $73 billion by 2021 [17].

One of the top concerns, however, is BYOD security. As BYOD devices are generally less well-managed than their enterprise counterparts, they are easier targets to compromise [6, 8, 47, 101]. This is further exacerbated by the fact that such devices are used to access sensitive enterprise resources as well as untrustworthy services in the wild [4, 22]. At its core, BYOD security represents a concrete instance of a fundamental challenge, sometimes known as the "end node problem" [11, 12]. The "end nodes" are not subject to the same level of centralized control, management, and protection as the enterprise infrastructure. We can easily update the access control lists on the gateway router, or patch newly discovered vulnerabilities on a server, but ensuring that the individual end points are properly patched is much more difficult. As such, insecure end devices tend to become the weakest link in the security chain [25].

One promising approach to BYOD security is to use *context-aware* policies, which enforce access control based on devices' runtime contexts [58]. For instance, a policy may deny access from devices whose TLS libraries have not been updated [98], or grant access to devices that are physically located in the enterprise boundary [88], or allow the use of a sensitive service only if administrators are online [56, 87]. In each of these scenarios, the policy makes security decisions based on additional "threat signals", such as the device location, library version, or even the status of other devices in the network. Context-aware policies are in a class of their own—they are much more *dynamic*, as contexts can change frequently (e.g., GPS location), and they require *global visibility* of the entire network (e.g., administrators online).

Supporting context-aware policies in enterprise networks presents interesting research challenges. Some traditional systems operate at the server side [89, 94], which enables easier management and update of security policies; others operate at the client side [91], making it easier to access device context. A common limitation, however, is that the individual nodes—clients or servers—only have local visibility. Such a "tunnel vision" hinders the ability to make synchronized security decisions network-wide [86]. Latest proposals address this using OpenFlow-based SDN, where a software controller collects context signals from all devices and enforces network-wide access control [58]. However, the central controller is vulnerable to control plane saturation attacks [82], and processing threat signals in a remote software controller incurs delay and decreases agility.

Our contribution. We present a novel design called Poise, or programmable in-network security, whose goal is to address the limitations of OpenFlow-based SDN defense. Poise has a new security primitive that runs in switch hardware, and it can change defense decisions at hardware speeds. Clients embed context signals in network traffic, and Poise parses these signals and enforces security policies without involving a remote software controller. This primitive is also reprogrammable in a declarative language to support a wide range of context-aware policies. These declarative policies are

Kang, Xue, and Morrison contributed to this work equally; Chen and Luo are the corresponding authors.

compiled by Poise into different configurations of the security primitive as P4 programs. Compared with traditional SDN defenses [58, 75, 82], this new paradigm results in defenses that are highly efficient, agile, and resilient to control plane saturation attacks [82].

The key enabler for Poise is the emerging *programmable data planes* developed by the latest networking technology. New switches, such as Intel FlexPipe [16], Cavium XPliant [9], and Barefoot Tofino [3], can be programmed in P4 [19] to support user-defined network protocols, custom header processing, and sophisticated state in hardware. P4-programmable networks represent a major step beyond OpenFlow-based SDN. OpenFlow switches have fixed-function hardware, and they can only support programmable forwarding by occasionally invoking remote software controllers. P4 switches, on the other hand, offer hardware-based programmability, which can be applied to every single packet without performance slowdown. The novelty of Poise lies in leveraging these new hardware features for context-aware security—we encode context signals with user-defined protocols, compute access control decisions using programmable packet processing, and support stateful, network-wide policies by designing hardware data structures.

After motivating our problem further in §2, we present:

- The concept of programmable in-network security (§3);
- A language and compiler for context-aware policies (§4);
- A novel in-network security primitive (§5);
- The Poise orchestration service and device module (§6);
- Discussions and limitations of Poise (§7);
- Prototype and evaluation of Poise that demonstrate its practicality, as well as its higher resilience to control plane saturation attacks and increased defense agility compared with OpenFlow-based SDN defense (§8);

We then describe related work in §9, and conclude in §10.

2 Background and Motivation

Context-aware security (CAS) stands in stark contrast to conventional security mechanisms—existing mechanisms can only support *static* policies, but CAS uses *dynamic* policies based on runtime contexts. For instance, NAC (network access control) mechanisms such as IEEE 802.1x [33] and Cisco Port/VLAN/IOS ACLs [10, 26] statically configure access control policies, whether for a device, an IP prefix, or a VLAN ID. Role- or attribute-based access control mechanisms [52, 53, 76] also perform access control based on statically-defined roles or attributes.

CAS, on the other hand, uses the runtime contexts of a request as threat signals (e.g., location/time of access, status of the network); whenever the signals change, the security decisions would adapt accordingly. The theoretical underpinnings of CAS have been studied more than a decade back [41], but it recently found an array of new applications in securing IoT and mobile devices [39, 51, 58, 60, 91]. These devices, just like the BYOD clients in our scenario, suffer from the "end node problem" [11, 12]. CAS has proven to be effective for such scenarios, because it can enable a more precise protection based on threat signals collected from the end nodes.

2.1 Design space

The concept of CAS by itself does not necessitate a client-, server-, or network-based design; rather, these design points have different tradeoffs. First off, purely server-side solutions are often ineffective, as we desire to collect context signals from client devices at runtime. Therefore, typical CAS systems [58, 91] need to install a *context collection* module at the clients. In terms of *policy enforcement*, one could co-locate enforcement with context collection, resulting in a purely client-based solution [91]. The main drawbacks, however, are that a) individual devices only have local views, making network-wide decisions hard to come by, and that b) policy management is much harder, as policies are distributed to each device; this might raise additional concerns if some policies are themselves sensitive data. Another option is to enforce the policies inside the network. The network has a global view for holistic protection, and it enables centralized policy management and update. Poise adopts this design option.

2.2 Traditional networks are not enough

However, traditional network devices (i.e., switches and middleboxes) are not up to the task, because they are built with fixed-function hardware that is customized for specific purposes. For instance, traditional switch hardware is optimized for a fixed set of protocols (e.g., TCP/IP), but it does not understand context information, such as GPS location, time of access, or library versions. Similarly, hardware middleboxes also come with fixed functions, e.g., firewalls or deep packet inspection (DPI); function updates are typically constrained by the speed of hardware upgrades, which is much slower than the need for defense adaptation. As a result, traditional in-network security mechanisms merely provide *fixed-function security*, such as static access control lists, firewalls, and traffic filters. There is a fundamental gap between the dynamic nature of CAS and the static nature of the network devices.

2.3 How about OpenFlow-based SDN?

Software-defined networking (SDN) [67] can partially address this by the use of a software controller for *control plane programmability*. Although the OpenFlow switch hardware remains fixed in function, switches can send `PacketIn` messages to the central controller for programmable decisions. This paradigm underlies many recent developments in network security [58, 75, 77, 80, 81, 82]. In particular, a recent work PBS [58] supports context-aware security by running

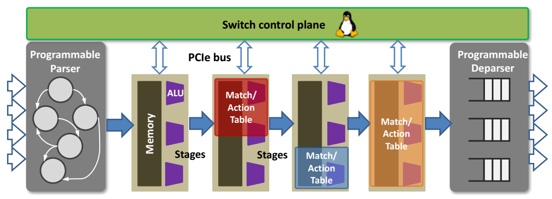


Figure 1: P4 switches are programmable in hardware. Packets first go through a programmable *parser*, which supports user-defined protocols. Packet headers are then streamed through a number of hardware *stages*, each of which contains stateful *registers*, arithmetic logic units (*ALUs*), and *match/action tables*. Packets can be *recirculated* to go through the stages multiple times to trigger different programmable elements.

the policy enforcement module as an "SDN app" in a centralized controller. This app can collect context signals from all devices and enforce access control in a global manner.

However, in traditional SDN, programmability comes at a great cost, as it resides in a centralized software controller. First, `PacketIn` messages incur a round-trip time delay between the switch and the remote controller, whereas packets in the data plane are processed at hardware speeds. As such, we can only programmatically process a small set of packets—typically one packet per flow (e.g., the first packet). Second, traditional SDNs are vulnerable to control plane saturation attacks [82], where an adversary can cause high-volume traffic to be sent to the software controller. A recent work OFX [84] has further highlighted that, for security applications that require dynamic, fine-grained decisions, centralized SDN controllers would pose a severe bottleneck. The key goal of Poise is to address the limitations of traditional SDN defenses by enforcing CAS in switch hardware.

2.4 Opportunity: Programmable data planes

Data plane programmability represents the latest step in the networking technology. In contrast to OpenFlow-based SDN, P4-programmable networks provide new features that can be reconfigured in hardware (Figure 1). The key novelty of Poise is to leverage them for in-network policy enforcement.

1. Customized header support for CAS. P4 switches can recognize customized protocols and headers beyond TCP/IP via the use of a programmable parser, without the need for hardware upgrades. Our observation is that this allows us to programmatically define context signals as special header fields, and embed them in network traffic. P4 switches can directly parse context signals from client traffic.

2. Security decision changes at hardware speeds. Each hardware stage is integrated with ALUs (Arithmetic Logic Units) that can perform computation over header fields at linespeed. The implication for security is that, without involving a remote software controller, switches can evaluate context values (e.g., GPS locations) and make security decisions (e.g., location-based access control) directly in hardware.

3. Cross-packet state for network-wide security. Last but not least, the hardware stages also have persistent memory in read/write registers, and they can process packets based on persistent state. We observe that this enables the network to make coordinated security decisions in a network-wide manner—decisions for one client could depend on past network behaviors, or activities from other parts of the network.

These hardware features are programmable in P4 [19, 43]. Switch programs can be compiled and installed from the switch control plane (Figure 1), which typically runs a customized version of Linux and has general-purpose CPUs. The P4 compiler maps a switch program to the available hardware resources [64]. Programs that successfully compile on a target are guaranteed to run at linespeed, due to the pipelined nature of the hardware. Programs that exceed available hardware resources would be rejected by the P4 compiler.

2.5 Trust model

Poise shares the same trust model as existing CAS solutions [58, 91]—the context collection module at the clients and policy enforcement module at the switch are both trusted. As a network-based design, Poise also trusts the network infrastructure (switches and access points). The context collection module can be installed as a pre-positioned Android kernel module with OEM support; this is common in Enterprise Mobility Management solutions [5, 27, 31, 91]. It only collects and propagates context signals when devices are connected to the enterprise network; standard BYOD frameworks such as Android for Work [2] or Samsung Knox [24] can support this. Users can install unmodified Android apps. CAS specifically protects against malicious apps, and following existing work [58, 91], we assume that malicious apps cannot compromise the kernel or obtain root privileges. It is possible to further relax these assumptions by directly establishing the root of trust in hardware [28, 30, 74]. In the case where untrusted devices may connect to the network, Poise needs to perform authentication on context signals before using them for decision making, e.g., by adding support for cryptography in P4 switches. We discuss this in more detail in §7.

3 Programmable In-Network Security

We call this new paradigm *programmable in-network security*.

Scenario. Consider the enterprise network shown in Figure 2, which hosts several types of private data, such as employee records and sales records, and also provides connectivity to the Internet. The operator wants to enforce dynamic access control of sensitive enterprise data in the presence of BYOD clients. For instance, the policy might specify that a) sales records should only be accessed by devices belonging to the sales department; b) during regular work hours; c) from devices that are properly patched to address some recently discovered vulnerability; and, d) a device can only access the sales records if the sales manager is online. Poise is designed for context-aware security policies such as these.

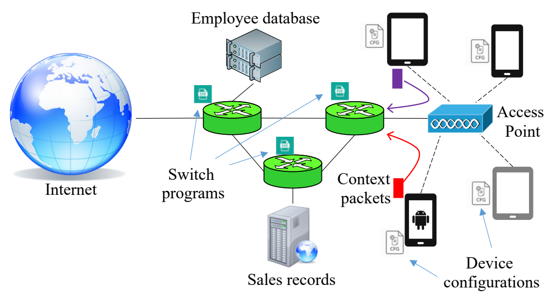


Figure 2: Poise compiles high-level policies into a) switch programs, and b) device configurations. The clients send periodic context packets to the network, and Poise enforces the policy in the switches.

The Poise system. At the heart of Poise is a novel *switch primitive* that can enforce CAS policies in hardware. The design of this primitive also tackles a practical challenge. Since P4 programs specify low-level packet processing behaviors, they are akin to "assembly-level" programs, and one often needs to hand-optimize P4 programs to reduce resource usage. Therefore, we allow network operators to specify CAS policies in a *declarative language* that is much higher-level than P4. Our *compiler* can then generate optimized P4 programs automatically, which are different versions of the security primitive. The Poise compiler also generates configurations for the context collection module at the clients. It collects context signals based on the configuration, and sends out periodic context packets to the network. Policy changes can be easily supported by a recompilation. Client configurations need not be affected by policy updates, unless the new policies require new types of context signals to be collected.

Next, we first describe the Poise language and compiler, then the switch primitive, and finally, the client module and how these components work together.

4 The Poise Language and Compiler

The policy language in Poise is inspired by the Frenetic family of SDN programming languages [38, 54, 68, 69, 79], but we adapt them a) from an OpenFlow setting to P4, which supports richer header operations and state, and b) from a network management setting to security, by supporting security contexts. Specifically, we have designed the Poise language based on Pyretic NetCore [69], where network policies are written as a series of match/action statements. In terms of the semantics of the language, a policy represents a function that maps an incoming packet to zero (i.e., drop), one (i.e., unicast), or more (i.e., multicast) outgoing packets. A policy could be as simple as `drop`, which drops all packets, although practically, the policy would make a decision based on the context a packet carries, such as `if match(dip==66.220.144.0) then drop`, which blacklists a block of destination IP addresses, or `if match(0800<=time<=1800) then drop`, which denies access depending on the time of day. Figure 3 summarizes the language syntax, and the highlighted portions show the differences from NetCore, which we explain more below.

Primitive Actions

$A \quad ::= \quad \mathsf{drop} \mid \mathsf{fwd(port)} \mid \mathsf{flood} \mid \mathsf{log}$

Expressions

$E \quad ::= \quad \mathsf{v} \mid \mathsf{e_1 + e_2} \mid \mathsf{e_1 - e_2} \mid \mathsf{e_1 * e_2} \mid M$

Constant Lists

$L \quad ::= \quad \mathsf{nil} \mid \mathsf{v, L}$

Predicates

$P \quad ::= \quad \mathsf{match(e_1 \circ e_2)} \mid \mathsf{match(h \circ e)} \mid \mathsf{match(h\ in\ l)} \mid \mathsf{P\&P} \mid \mathsf{(P|P)} \mid \mathsf{!P}$

Monitors

$M \quad ::= \quad \mathsf{count(P)}$

Policies

$C \quad ::= \quad \mathsf{A} \mid \mathsf{if\ P\ then\ C\ else\ C} \mid \mathsf{(C|C)}$

Figure 3: The language syntax for Poise policies. Context fields are represented as h. Expressions are represented as e, or v (constants). The ∘ operator indicates comparisons.

4.1 Key language constructs

Security contexts. Poise encodes context fields in customized headers, such as `time` or `dev`. When a policy refers to multiple context fields, Poise structures the context headers in the order in which they appear in the policy program.

Context operations. Poise also supports sophisticated operations over context headers, as indicated in the expressions and predicates in Figure 3. An expression could be a constant, an arithmetic operation over header fields, or a complex expression over subexpressions. Security decisions are made based on predicates over expressions, where the ∘ operator indicates comparisons such as $>$, $<$, and so on. Contexts can also be tested against constant lists, which are pre-defined in the policy to encode membership relations. For instance, one could define a list of devices with administrative roles as `def adminlst = ["dev1", "dev2"]`. Then, the policy could refer to the lists as part of the decision-making process, such as `if match(!dev in adminlst) then fwd(mbox)`, which forwards traffic from non-admin devices to a middlebox for traffic scrubbing. We note that the original NetCore does not support the use of contexts or sophisticated context operations; rather, Poise adds such extensions based on the extra processing power in P4 for security support.

Stateful monitors. Unlike NetCore, Poise supports stateful policies which make security decisions based on network-wide state. This is done via monitor expressions, which monitor activities of interest in persistent state. A monitor expression is written as `count(pred)`, which counts the number of packets that satisfy the predicate `pred` in the current time window; for instance, `count(match(is_admin))` counts the number of packets generated from a device with an administrative role. The counters are periodically reset to zero when a new time window begins. These monitors enable programmers to write network-wide policies. This is different from stateless NetCore policies, where monitors passively collect traffic statistics, but do not affect forwarding decisions.

Actions. The decision of a Poise policy is represented by its action field. Currently, Poise supports four types of actions. The `drop` decision denies access. The `fwd` decision allows access, and can be further parameterized by an outgoing switch port, so that it can actuate further processing—e.g., sending packets through an DPI device that can be reached via a particular port. The `flood` decision broadcasts a packet. The `log` decision sends a packet to a logger that detects potentially suspicious activity; this is achieved by aliasing the `fwd` decision and specifying a special port for the local switch CPU. Packets sent for logging will be pumped to the control plane of the switch, which runs a logging component. This can be easily generalized to enable remote logging, e.g., by wrapping the packet inside another IP header, where the destination IP represents a network activity logger.

Composing policies. Similar as NetCore, Poise can compose multiple policies `P1|P2|...|Pn` and compile them into a single switch program. This is useful, e.g., when `Pi` and `Pj` check different context signals and the enterprise wants to apply them in combination. The Poise compiler rejects the composition of conflicting policies at compilation time.

4.2 Example policies

The Poise language is expressive enough to capture a wide range of existing and new policies, and it is much more concise than low-level languages such as P4. Next, we describe seven practical BYOD policies, where the first two are adapted from existing work [58] and the rest are new policies supported by Poise. Variables `dev`, `time`, `lat`, `lon`, and `usr` are customized header fields.

P1: Block certain services in work hours [58]: A common BYOD policy is to block access from certain devices to entertainment websites during work hours:

```
def businesslst = ["dev1", "dev2"]
if match(dip==66.220.144.0 &
   dev in businesslst &
   (time>=0800)&(time<=1800))
then drop
```

P2: Direct traffic from guest devices through a middlebox [58]: Another useful policy is to distinguish traffic from authorized devices and guest devices, and direct guest traffic through a middlebox for traffic scrubbing:

```
def authlst = ["dev1", "dev2"]
if match(dev in authlst)
then  fwd(server)
else  fwd(mbox)
```

New policies. There are also useful policies in Poise that cannot be easily supported in traditional networks; they are implementable in Poise due to the use of programmable data planes, which can perform arithmetic operations over context headers, and maintain network-wide state to make coordinated security decisions. We give an example of each below.

P3: Distance-based access control: This policy grants access to a service only if the user is within a certain distance from a physical location (e.g., the server room); this requires performing arithmetic operations over GPS coordinates embedded in the packet header:

```
if ((lat-x)*(lat-x)+(lon-y)*(lon-y) < D)
then  fwd(server)
else  drop
```

P4: Allow access only if admin is online: Poise can support coordinated, network-wide policies by monitoring security events of interest and making decisions based on the result. For instance, a policy might grant access to a service only if the admin is online:

```
def adminlst = ["Bob", "Alice"]
c = count(match(usr in adminlst))
if match(c>0) then fwd(server)
```

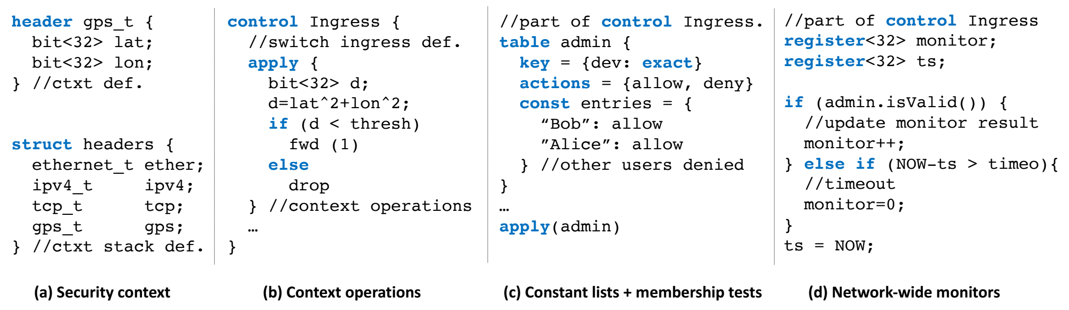
Advanced policies. Inspired by the literature of "continuous authentication" [37, 49, 50, 92], we propose a set of advanced policies that use device context to detect subtle but important indicators of potential attacks. Due to space constraints, we only describe the high-level policies, but not the programs. *P5: Block requests without explicit user interaction*, which denies access to a sensitive service if all apps are running in the background and there is no user interaction with the touchscreen to trigger the request; such requests are likely generated by malware. *P6: Scrub traffic if UIs are overlapping*, which forwards traffic through a middlebox if the context information shows that app UIs are overlapping—a potential sign for UI hijacking [55]. *P7: Conduct deep packet inspection if camera/recorder is on*, which detects if sensitive information is being leaked through an active camera/recorder app [36].

4.3 Compilation

Next, we discuss how the Poise compiler processes the key language constructs and generates P4 implementations.

Compiling security contexts. The Poise compiler generates P4 headers for each context. Context packets have special IP protocol numbers (143 for TCP, 144 for UDP), and they have no payload. Context headers follow the TCP/UDP headers (e.g., Eth|IP|TCP|Ctxt). Poise switches recognize the context headers by the IP protocol number, whereas legacy switches forward these packets based on destination IPs. User traffic is not modified by Poise in any way. (See Figure 13 in Appendix.) As a concrete example, Figure 4(a) shows the P4 headers for the `gps` signals: latitude and longitude.

Compiling context operations. The Poise compiler distinguishes between five classes of context operations: arithmetic operations, bitwise operations, comparisons, context matches, and membership tests. The first three classes are simpler to handle, as they can be directly translated into their P4 counterparts; the latter two require the compiler to generate additional code components in P4. First off, all context fields are compiled into header definitions and references to these



```
header gps_t {
  bit<32> lat;
  bit<32> lon;
} //ctxt def.


struct headers {
  ethernet_t ether;
  ipv4_t     ipv4;
  tcp_t      tcp;
  gps_t      gps;
} //ctxt stack def.
```

(a) Security context

```
control Ingress {
  //switch ingress def.
  apply {
    bit<32> d;
    d=lat^2+lon^2;
    if (d < thresh)
      fwd (1)
    else
      drop
  } //context operations
  …
}
```

(b) Context operations

```
//part of control Ingress.
table admin {
  key = {dev: exact}
  actions = {allow, deny}
  const entries = {
    "Bob": allow
    "Alice": allow
  } //other users denied
}
…
apply(admin)
```

(c) Constant lists + membership tests

```
//part of control Ingress
register<32> monitor;
register<32> ts;

if (admin.isValid()) {
  //update monitor result
  monitor++;
} else if (NOW-ts > timeo){
  //timeout
  monitor=0;
}
ts = NOW;
```

(d) Network-wide monitors

Figure 4: The Poise compiler processes the key language constructs and generates P4 implementations. The P4 snippets shown are simplified for clarity of presentation. For instance, in (b), the instantiation of the `thresh` register is not shown; in (d), the timestamp of a packet is obtained via the `ingress_global_timestamp` field instead of a variable called `NOW`.

headers, as discussed above. Then, for arithmetic, bitwise, or comparison operations over header fields, such as `lat*lat`, `sensors&0x01`, or `time<10`, our compiler forms expressions using the corresponding P4 operations over the headers. For arithmetic operations, the current P4 specification supports addition, subtraction, and multiplication, which are all supported by the Poise compiler. Notably missing from the list are division and modulo operations, which tend to be expensive to implement in switch hardware (although sometimes they can be approximated by bit shifts if the divisor is a power of two). If a Poise program involves operations unimplementable in P4, our compiler would reject the policy during compilation.

As an example, Figure 4(b) shows simplified P4 snippets that our compiler generates for computing the distance between a pair of GPS coordinates to a pre-defined center (assumed to be `(0,0)`). Our compiler also generates conditional statements based on the policy, e.g., `if-else` branches to test if the distance exceeds a threshold. Context operations are performed within an `apply` block at `control Ingress`, which means the switch ingress pipeline.

Context matches, on the other hand, are compiled into match/action tables in P4. A match can be an `exact` match, which requires matching a context field against a list of keys bit by bit. It could be a `range` match, which compares a context field against a range of values in TCAM (Ternary Content Addressable Memory). By default, Poise uses 4-byte headers for exact matches, and 2-byte headers for range matches. Context matches can also be performed against a user-specified constant list that defines membership, e.g., a set of devices owned by the sales department. For a list with k items $[a_1, a_2, \cdots, a_k]$, our compiler will construct a match/action table with k entries, where each entry corresponds to an item in the list. The actions associated with the entries depend on the mode of access defined in the policy program.

For instance, consider the P4 snippet in Figure 4(c), which shows a match/action table generated from a constant list of two entries: Bob and Alice. The table implements an `exact` match on the device ID field. If the context match is successful, then the device will be granted access; unsuccessful matches indicate that the context fails the membership test, and these requests will be denied access.

Compiling stateful monitors. The Poise compiler generates a read/write register for each stateful monitor in the policy, as well as code components for detecting monitored events and updating the monitor values. Such monitors are implemented as a number of registers in P4, which are supported in switch SRAM. Updates to the registers are linespeed, so they can be performed on a per-packet basis. Specifically, for each incoming packet, the generated code checks whether this corresponds to an event of interest, using either a context match, or a match over a membership list. If this event should be monitored, the code additionally updates the monitor register and records the event timestamp. If a long time has elapsed after the previous event took place, then this register is cleared to indicate that the monitored event is absent. As discussed before, monitors enable network-wide policies that make coordinated security decisions—a policy can test if a monitored event is detected, and make decisions accordingly.

Concretely, the snippet in Figure 4(d) shows an example. It instantiates a 32-bit register to hold the monitor value, and updates the register when the `admin` context is active in a packet. The code associates a timestamp to this monitor, and resets the monitor upon timeout.

Compiling actions. An action will be taken on each packet to represent the final decision made on its context. In P4, decisions are represented by attaching special metadata fields to a packet, which will be recognized and processed by a *traffic manager*, which schedules packets to be sent on the correct outgoing port(s) or dropped. Logging a packet is achieved by setting the outgoing port to be the switch CPU.

Compiler optimizations. Programmable data planes have three types of notable constraints. *Stages:* There is a fixed number of hardware stages, and a packet can only match against one single context table per stage. *Tables:* A single stage can only hold a fixed number of tables. *Memory:* Each stage has a limited amount of memory.

The Poise compiler performs two types of optimizations, which are particularly useful when Poise needs to compose many policies together. (a) If multiple policies check against the same context signal, our compiler will perform *table dedu-*

plication to eliminate redundant context tables and save memory. (b) If a policy performs more context checks than the number of available stages, Poise will *collapse* the policy by *recirculating* context packets to traverse the stages multiple times, triggering different tables at each recirculation. This addresses the switch constraint that a packet can only trigger a single table per stage. Our optimization creates the illusion of a larger number of stages with the cost of slightly increased latency for recirculated packets. We refer interested readers to Appendix A.2 for more details.

Summary. So far, we have described the basic compilation algorithm as if each packet is tagged with context information. This makes it easy for a switch to access a packet's context without keeping state, but it results in high traffic overhead. Next, we will relax this assumption by the design of a stateful, efficient, programmable in-network security primitive.

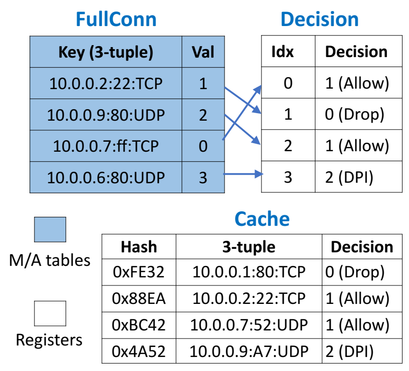
5 The In-Network Security Primitive

Poise has a security primitive that runs in a programmable switch, which is dynamic, efficient, and programmable.

Goal: A dynamic and efficient security primitive. The in-network primitive should ideally allow the level of protection to be adjusted between per-packet and per-flow granularities, by supporting a tunable frequency of context packets for each connection. At one end of the spectrum, per-flow granularity of protection degenerates into a static security mechanism that does not support context changes within a connection. Thus the protection is very coarse-grained, especially for long-lived connections that persist for an extended period of time (e.g., push-based mobile services, such as email [93]). At the other end, per-packet granularity is extremely fine-grained, but it may incur unnecessary resource waste unless context changes from packet to packet. As a concrete example, if there are 20 context fields across policies, then each client needs to send $20 \times 4/500 = 16\%$ extra traffic, assuming typical 500-byte packets and 4-byte context fields. The Poise primitive supports a property that we call *subflow-level security*, which achieves a tunable tradeoff between security granularity and overhead when enforcing context-aware security.

Property: Subflow-level security. We state this property more formally below. Consider a sequence of packets in the same flow $c_i, p_{i_1}, \cdots, p_{i_k}, c_{i+1}$, where c represents a context packet and p a data packet. Subflow-level security requires that decisions made on the context packet c_i should be applied to subsequent data packets $p_{i_j}, i_j \in [i_1, i_k]$, but fresh decisions should be made for data packets that follow c_{i+1}. The decision granularity can be tuned by f, the frequency of context packets. This results in an overhead of $s \cdot f$, where s is the size of context packets. For instance, assuming 80-byte context packets and a frequency of one context packet per ten seconds, the overhead would be as low as 8 bytes per second.

Challenges. Designing a primitive that supports subflow-level security, however, requires tackling three key challenges.



Key (3-tuple)	Val
10.0.0.2:22:TCP	1
10.0.0.9:80:UDP	2
10.0.0.7:ff:TCP	0
10.0.0.6:80:UDP	3

Idx	Decision
0	1 (Allow)
1	0 (Drop)
2	1 (Allow)
3	2 (DPI)

Hash	3-tuple	Decision
0xFE32	10.0.0.1:80:TCP	0 (Drop)
0x88EA	10.0.0.2:22:TCP	1 (Allow)
0xBC42	10.0.0.7:52:UDP	1 (Allow)
0x4A52	10.0.0.9:A7:UDP	2 (DPI)

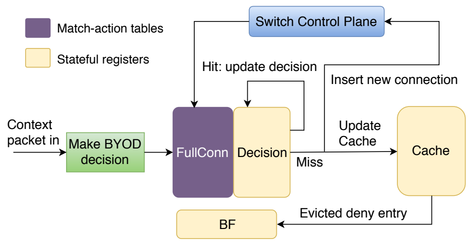
Figure 5: The key/value store with example entries.

(a) Keeping per-flow state requires a prohibitive amount of memory, but modern switches only have O(10MB) SRAM. Poise addresses this by approximating per-flow state using a on-chip key/value store. *(b) Buffering control plane updates* is necessary for handling new flows. Although context changes can be entirely handled by the data plane, new flows require installing match/action entries from the switch CPU, which takes time. Before updates are fully populated, Poise uses another hardware data structure akin to a cache to make conservative decisions for buffered flows. *(c) Mitigating DoS attacks* that could arise due to the interaction between data and control planes. This defends against malicious clients that craft special context packets to degrade the performance of selected clients, or even the entire network. In the next three subsections, we detail each of these techniques.

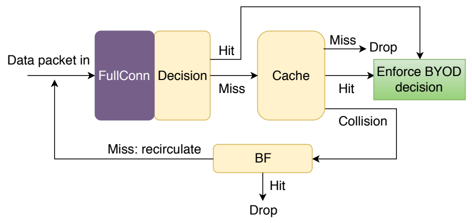
5.1 Approximating per-flow state

The key problem in the first challenge stems from the fact that the switch needs to process data packets without contexts attached to them. Therefore, when a switch processes a context packet, it needs to remember the decision and apply it to subsequent data packets in the same connection, until the next context packet refreshes the decision. A naïve design would require keeping per-flow state on the switch, which leads to high memory overhead.

To address this, Poise approximates per-flow state using a key/value store consisting of two data structures, `FullConn` and `Decision`, as shown in Figure 5. The `FullConn` schema is `[sip, sport, proto]`$\rightarrow$`idx`. The match key is the source IP/port and protocol for the client, and the value is an index to a register array `R`. The indexed register `R[idx]` holds the decision made on the latest context packet within this connection, and it can be refreshed entirely in the data plane. Insertions to this key/value store require control plane involvement, but they are relatively infrequent and only needed for new connections. Since the match key does not include the destination IP/port, this introduces some inaccuracy when a client reuses a source port across connections. Therefore, for short-lived connections, data packets may see slightly outdated decisions. To ensure that such inaccuracy does not



(a) The logic for processing context packets



(b) The logic for processing data packets

Figure 6: Poise uses a combination of match/action tables and stateful registers to process context and data packets.

misclassify a "deny" as an "allow", we blacklist the source IP addresses that have recently violated the enterprise policy: all connections from these clients would be blocked temporarily.

5.2 Buffering control plane updates

Insertions to `FullConn` requires control plane involvement, so they take much longer than updating policy decisions for an existing connection. As a result, when data packets in a new connection arrive at the switch, the `FullConn` match/action table may not have been populated with the corresponding entry yet. To address this, Poise uses a *level of indirection* by creating a small hardware `Cache` to buffer decisions for pending table updates, which resides on the data plane and can be updated at linespeed. All decisions in `Cache` are up-to-date, since writes to this cache are immediately effective; but this table has a smaller capacity. The `FullConn` table takes more time to update, but it holds more connections.

The cache design. As shown in Figure 5, `Cache` has a fixed number of entries. Our implementation uses 2^{16} entries, which corresponds to the output size of a CRC-16 hash function. Each entry is of the form `h`$\rightarrow$`[sip,sport,proto,dec]`, where `h` is the CRC hash of the flow's three tuple, i.e., `h`=CRC(`sip,sport,proto`), and `dec` is the decision made based on the context packet. The size of `Cache` is $2^{16} \times (7+1)$=0.38 MB memory. When Poise receives a context packet from a new connection (Figure 6a), it immediately adds the entry to `Cache`, and then invokes the control plane API to insert the match/action entry in `FullConn`. Since CRC functions are not collision resistant, different connections may be mapped to the same entry; hence, we evict old entries upon collision. When a data packet comes in (Figure 6b), Poise first matches it against the `FullConn` table and applies the decision upon success. If there is no entry for this packet, then Poise indexes the `Cache` table instead. Upon a cache hit, the corresponding decision is applied to the data packet. Upon a cache miss, one of two situations has happened: a) the switch has not seen a context packet from this client, or b) the entry for this client has been evicted due to collision. Poise distinguishes between these cases using the following cache eviction algorithm.

Handling cache evictions. Upon collision, we always replace the existing entry with the new one. This is because Poise has already invoked the control plane to install the corresponding entry in `FullConn`, which will complete in time. Therefore, if a packet does not match any entry in `FullConn` and experiences a collision in `Cache`, we use a special instruction to recirculate the packet inside the data plane to delay its processing. Recirculated packets are sent back to the switch ingress to be matched against the `FullConn` table one more time. This recirculation is repeated up to k times, where the latency is chosen to be larger than the expected time for the control plane to populate an entry. If a packet has reached this threshold, and the `FullConn` table still has not been populated, then we consider this to be case a) above and drop the packet.

Early denies. To reduce the amount of recirculated packets, we make early decisions to drop a packet if its context is evaluated to a "deny". Specifically, when evicting an entry from `Cache`, we add its source IP address into a blacklist Bloom filter (`BF` in Figure 6) if the decision is to drop. Source addresses in `BF` represent devices that have violated the policy recently and need to be blacklisted for a period of time. If a packet cannot find an entry in either `Cache` or `FullConn`, but hits `BF`, we drop it without recirculation. Since Bloom filters can only produce false positives, but never false negatives, we will always correctly reject an illegal connection. However, we might err on the conservative side and reject legal connections as well, if the `BF` produces a false positive. This is a rare case, however, as this will only happen during the window in which `FullConn` has not been populated, the `Cache` entry has been evicted, and the `BF` happens to produce a false positive. Nevertheless, Poise periodically clears this Bloom filter to reduce false positive rates, which grow with the number of contained elements. When the `BF` is being cleared, packets will be recirculated until the operation completes.

5.3 Handling denial-of-service attacks

Since Poise requires extra processing inside the network, we need to ensure that it does not introduce new attack vectors. Specifically, we have identified two potential denial-of-service attack vectors and hardened the primitive against them.

Total residency attacks. Different from stateless, IP-based routing, Poise keeps state in the `FullConn` table. Therefore, an attacker could initiate many new connections and try to a) overwhelm the `FullConn` table and b) constantly involve the switch CPU to install new entries. A defense, for instance, could rate limit the number of active connections and to control the growth of the `FullConn` table. In addition, the Poise control plane periodically scans through the `FullConn` table and expires inactive entries (using hardware support) to make room for new connections.

Cache eviction attacks. The above algorithm defends against a malicious attacker that generates many connections to overwhelm the `FullConn` table. However, an attacker can also launch targeted DoS attacks without initiating a suspiciously large number of connections. Specifically, she could send context packets more frequently than usual, and try to evict cache entries from `Cache` that are mapped to the same bucket. Although the attacker may not know the hash seed, therefore cannot predict who would be the victim of the attack, she could degrade the performance of the connection that shares the same hash entry, if one exists. To prevent such attacks, we enhance the cache eviction strategy. When replacing an old entry e_o with a new entry e_n, we check whether these two entries are from the same source IP. If so, we immediately replace the entries. If not, we opportunistically perform the replacement. By doing so, we limit the amount of damage an attack can cause by sending frequent context packets.

6 Orchestrating Poise

Next, we explain how we orchestrate the Poise in-network primitive using a software *controller*, and describe the *client module* that runs on the mobile devices for context collection.

The Poise controller. Poise has a controller that hosts the compiler and distributes the generated data plane programs to the switches. Unlike an OpenFlow-based SDN controller, which actively makes decisions on behalf of the data plane, the Poise controller is *not involved* in packet processing, so it does not create any software bottleneck. The main controller runs in a remote server, and uses well-defined RPC calls to communicate with programmable switches' local control planes. Each switch has a local control plane running on the switch CPUs, and it configures the switch data plane by installing match/action table entries, loads new switch programs, and serves as the primary logging component.

The Poise client module. Our client module PoiseDroid is installed at BYOD devices to collect context signals and embed them into packets. PoiseDroid does not require modification of existing Android apps, but rather acts as a pre-positioned kernel module. When the device connects to the enterprise network, it needs to go through an authentication phase (e.g., using WPA3 [95], or additionally using two-factor authentication [71]). The module stops propagating context signals when the device leaves the network. Figure 7 shows the architecture of PoiseDroid with three submodules.

The context submodule. It collects context information from the Android system services [97] using usermode-helper APIs [34, 63], and it registers a virtual device to redirect the context data to our kernel module. The information to be

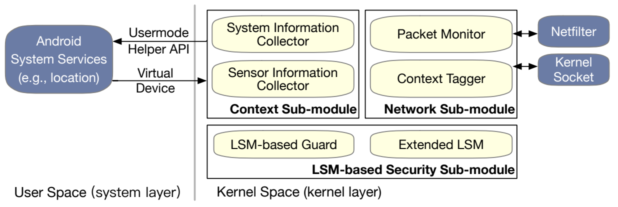

Android System Services (e.g., location)
Usermode Helper API
Virtual Device
System Information Collector
Sensor Information Collector
Context Sub-module
Packet Monitor
Context Tagger
Network Sub-module
Netfilter
Kernel Socket
LSM-based Guard
Extended LSM
LSM-based Security Sub-module
User Space (system layer)
Kernel Space (kernel layer)

Figure 7: The architecture of the PoiseDroid client module.

collected is specified by a BYOD client configuration, which includes a) app information, such as UIDs of active apps, b) system information, such as screen light status, and c) device status, such as accelerometer and gyroscope readings.

The protection submodule. It protects the registered virtual device, the system tools (e.g., `dumpsys`), and the system services using LSM hooks in Android kernel [40, 72]. It monitors invocations of selected system calls, such as *ptrace()*, *open()*, *mprotect()* and *chown()*, and prevents any other processes to write false data to these protected components.

The network submodule. It crafts and sends special context packets with signals needed for the enterprise policies, using a frequency specified in the configuration. When an app opens a new socket, or when an existing socket sends packets after being dormant for a while, it also generates a context packet.

7 Limitations and Discussions

Authentication. As an access control mechanism, Poise focuses on resource *authorization* and should be used with an *authentication* method, e.g., the SAE (simultaneous authentication of equals) protocol [57] in WPA3 [95], or two-factor authentication with TOTP [71]. Only authenticated users can further access enterprise resources in Poise.

Context integrity and privacy. One limitation of the current Poise prototype is that it relies on external cryptographic mechanisms to secure context packets. This is because today's P4 switches do not have built-in support for cryptography. Adding cryptography support in P4 switches can be achieved in two ways. First, the P4 standard allows cryptographic modules to be added as "externs". The main Poise program can invoke such an extern module to encrypt, decrypt, and authenticate context packets. Second, a recent project SPINE [48] shows that the current P4 language is expressive enough to implement a keyed hash function. SPINE further leverages this to generate one-time pads to encrypt/decrypt IP and TCP headers at linespeed. Poise could use a similar design, where clients encrypt context packets and the switch decrypts them using shared keys. To protect integrity, Poise can additionally use the keyed hash function to generate a MAC (message authentication code) of the context fields at the clients, and verify the MAC at the switch. To protect against replay attacks, the context packets also need to include timestamps or sequence numbers. Either way, the Poise switch or the "extern" module needs to be configured with key pairs with each enterprise client.

Existing security mechanisms in enterprise networks can also offer some support. Typically, client devices connect to the network via wireless access points (APs), and then to the wired network. Communication between clients and APs can be protected by WPA3 [95], and communication between the APs and the wired network by MACsec [15]; both can protect the integrity and confidentiality of packets and are secure against replay attacks [15, 95]. Under these protections, context packets are always encrypted on (wired and wireless) network links, therefore secure against network reconnaissance attacks. However, supporting cryptography in P4 switches would provide stronger, end-to-end guarantees.

8 Evaluation

In this section, we describe the experimental results obtained using our Poise prototype. Our experiments are designed to answer five research questions: a) How well does the Poise compiler work? b) How efficiently can Poise process the security contexts inside the network? c) How well does Poise scale to complex policies? d) How much overhead does the Poise client incur on mobile devices? and e) How does Poise compare with traditional SDN-based security?

8.1 Prototype implementation

We have implemented the Poise prototype using 5918 lines of code in C/C++ and Python [20]. The Poise *compiler* is implemented in C++, using Bison 2.3 as the syntax parser, and Flex 2.5.35 as the lexer. It can generate switch programs in P4 for the Tofino hardware. The PoiseDroid *client module* is implemented in C as a pre-positioned kernel module on Linux 3.18.31. It extends the default LSM framework, SEAndroid, to implement the protection submodule. For evaluation, PoiseDroid runs on a Pixel smartphone with a Qualcomm Snapdragon 821 MSM8996 Pro CPU (4 cores) and Android v7.1.2. The Poise *control plane* is implemented in Python, and runs as part of the control plane software suite for the Tofino switch. It manages the match/action table entries and reconfigures the data plane programs. It can also be configured to invoke the hardware-based packet generator on the switch to send traffic at linespeed (100 Gbps per port), which we have used to test the latency and throughput of Poise.

8.2 Experimental setup

We set up a testbed with one Wedge 100BF Tofino switch and two servers. The Tofino switch has a linespeed of 100 Gbps per port, and 32 ports overall, achieving an aggregate throughput of 3.2 Tbps when all ports are active. It also has a 200 Gbps pipeline—separate from the 32 regular ports—for handling packet recirculation. Each server is equipped with six Intel Xeon E5-2643 Quad-core CPUs, 128 GB RAM, 1 TB hard disk, and four 25 Gbps Ethernet ports, which collectively can emulate eight forwarding decisions (one per server port). The servers are connected to the Tofino switch using breakout

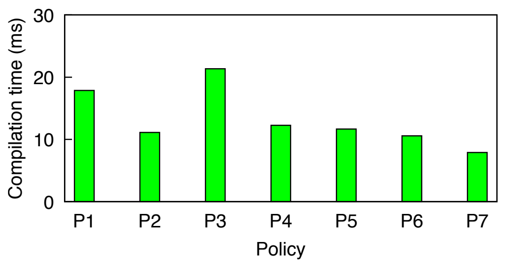


Figure 8: Poise compiles the policies efficiently.

cables from the 100 Gbps switch ports to the 25 Gbps server Ethernet ports. At linespeed, the testbed should achieve full 100 Gbps bandwidth per switch port.

On the first server, one of its ports is configured to be an enterprise server, and other ports are configured to emulate a DPI device, a traffic scrubber, and a logger, respectively. The other server functions as an enterprise client. The mobile traces are first collected from our Pixel smartphone, and then "stretched" to higher speeds to be replayed. The replay can be initiated from a) the enterprise client, or b) the hardware generator for Poise at linespeed.

8.3 Compiler

We start by evaluating the performance of the Poise compiler and its generated programs. All programs support one million connections in the `FullConn` table.

Compilation speed. In order to understand the performance of our compiler, we measured the time it took to generate switch programs for each of the seven policies. We found that compilation finished within one second across all policies. P1 and P3 took slightly more time than the rest, because they involve more context fields and our compiler needs to generate more logic for header processing. Figure 8 shows the results.

Generated P4 programs. The generated P4 programs have 855-975 lines of code, which are significantly more complex than the original policy programs that only contain a few lines of code. For one million connections across policies, the utilization of Poise for SRAM (used for exact match) is roughly 43%, for TCAM (used for longest-prefix match) is below 1.1%, and for VLIWs (Very Long Instruction Words, used for header modifications) is below 7%.

8.4 In-network processing overhead

Next, we turn to evaluate the overhead of Poise in terms of packet processing latency and switch throughput.

Packet processing latency. Poise increases the overhead of packet processing, because it needs to process context headers and approximate per-flow state. To quantify this overhead, we have tested the latency for Poise to process a) a context packet, b) a data packet, and compared them with c) the latency for directly forwarding a packet without any processing. Figure 9 shows that for all tested policies, the extra latency on average is 72 nanoseconds for processing data packets,

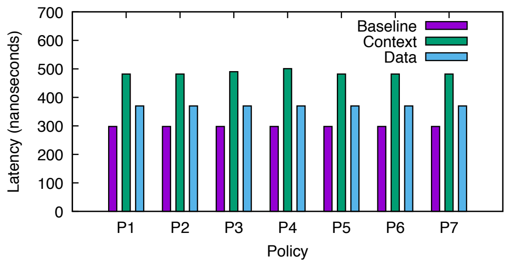


Figure 9: The amount of processing latency of Poise is small.

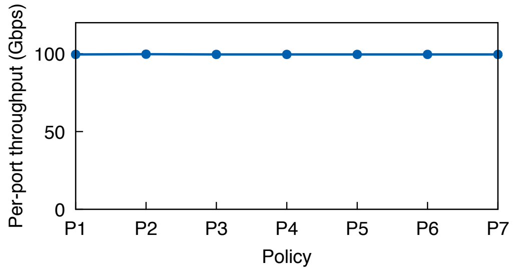


Figure 10: Poise achieves full linespeed programmability.

and 189 nanoseconds for processing context packets. In an enterprise network where the round-trip times are on the order of milliseconds, such a small extra latency is negligible.

Switch throughput. Next, we measured the throughput per switch port using the hardware packet generator for stress testing. The generator ingested mobile traces collected from our phone, and stretched the trace to be 100 Gbps. Figure 10 shows the per-port throughput for all policies. As we can see, although there is additional processing delay in Poise, the pipelined nature of the switch hardware makes it achieve full bandwidth nevertheless. In other words, Poise leverages programmable data planes to enforce context-aware security at linespeed, a key goal that we have designed for.

8.5 Scalability

Next, we evaluate how well Poise scales to complex policies. As policies may perform different numbers of checks on different numbers of contexts, we define a "unit policy" to be one that performs a single check on a single context. We then create many unit policies, and use the Poise compiler to compose them together. We characterize the complexity of the composed policy in two dimensions: a) the number of checks per context, and b) the number of contexts. For a), we further distinguish between exact vs. range checks, and for b), we distinguish between regular (i.e., non-monitor) vs. monitor contexts. For instance, consider the following unit policies:

```
if match (usr==Bob) then fwd(mbox)
if match (lib==1.0.2) then fwd(server)
```

We say that the composed policy has two regular contexts and performs two exact checks—one check per context.

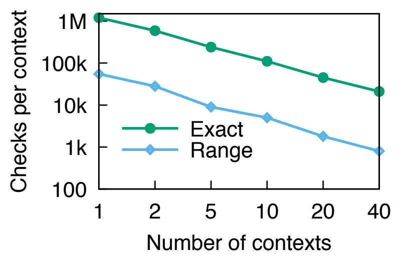
Number of checks. Poise compiles each check into a match/action entry, so the number of checks a switch can support depends on its available memory (SRAM and TCAM). Exact checks (e.g., X==1) are supported by SRAM and range checks (e.g., 10<X<20) by TCAM, so they are bottlenecked by the SRAM and TCAM sizes, respectively. We first measured the maximum number of *exact* checks Poise can perform on a single context, by asking the compiler to compose more and more unit policies until the compilation failed. We found that our switch can support 1.2 million checks, which are spread across 5 hardware stages. We then modified all unit policies to perform *range* checks, and found that Poise can perform 55 k checks, as the TCAM size is smaller.

Number of contexts. Poise compiles each regular context into a match/action table, so the number of contexts is bottlenecked by the number of tables a switch can support. We increased the number of contexts (e.g., time, library version) from one to the maximum until compilation failed, and found that Poise can support a maximum of 40 contexts—each of the 5 stages can support 8 context tables.

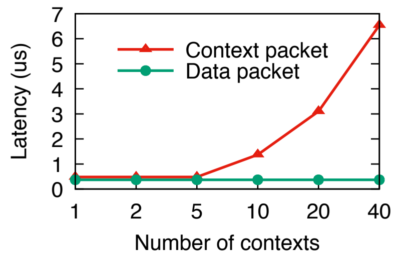
For each data point, we also measured the number of checks Poise can perform per context. We found that the number of checks per context decreases as we add more contexts, as the context tables need to multiplex switch memory. With 40 contexts, Poise can perform 21 k exact checks or 0.8 k range checks per context (Figure 11a). In other words, Poise can support at least 21 k distinct context values (e.g., user IDs for per-user policies) or 0.8 k distinct context intervals (e.g., time intervals for time-based access control).

We then modified all unit policies to check against network-wide monitors. A monitor is compiled into two tables—one for monitor updates, and another for monitor checks. Poise supports a maximum of 20 monitors in 40 tables. Policies can also use a mix of monitors and regular contexts. The constraint on the number of monitors m and the number of regular contexts c is $2\times m+c \leq 40$, as they are all compiled into tables under the hood. In terms of the number of checks per monitor, the results for a policy with m monitors are similar as those for a policy with $2\times m$ regular contexts (Appendix A.3).

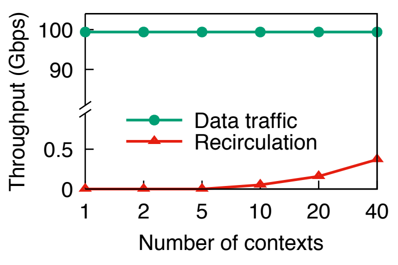
Overhead. We define a "baseline" to be the latency and throughput for a unit policy, where a context packet traverses the hardware stages exactly once without recirculation. A packet with k contexts would be recirculated to traverse the stages $\lceil \frac{k}{5} \rceil$ times, every time matching against 5 tables, one in each stage. At the maximum, Poise supports 7 recirculations for 40 contexts at a latency of 6.5μs (Figure 11b), which is still orders of magnitude lower than typical enterprise RTTs (ms). Recirculation also causes extra traffic overhead. We measured the overhead using 1 million connections and one context packet per second per connection. As Figure 11c shows, the maximum recirculation overhead is 0.37 Gbps per port. A monitor policy with m monitors has similar results as a policy with $2\times m$ regular contexts (Appendix A.3). Exact and range checks have similar results, as the types of checks do not affect the number of recirculations.



(a) Num. of contexts vs. num. of checks



(b) Num. of contexts vs. latency



(c) Num. of contexts vs. traffic overhead

Figure 11: Poise can perform 1.2 million exact checks for a single context, or 21k exact checks for a maximum of 40 contexts. Context packets with more than 5 contexts need to be recirculated multiple times; Poise supports a maximum of 7 recirculations, which leads to a latency of 6.5μs and an additional 0.37 Gbps traffic per port in a dedicated recirculation pipeline. Poise supports fewer range checks (55 k for one context, 0.8 k for 40 contexts) than exact checks, as the former are supported in TCAM, which is smaller than SRAM; but the latency and bandwidth overheads are similar, as they do not depend on the types of checks. Data packets are not affected by policy complexity, as they simply look up the decisions from the connection table.

We note that recirculation traffic is contained in a dedicated 200 Gbps switch pipeline—it does not compete with normal user traffic. Also, recirculation only incurs latency on context packets, as data packets simply look up previous decisions in a single stage traversal. Therefore, even when recirculating context packets, Poise still processes data packets at baseline latency and full linespeed (Figures 11b-11c).

Discussion: Per-user policies. Poise supports per-user policies by including the user ID as a context. Therefore, per-user policies merely reduce the number of total contexts by one, from 40 to 39. The number of user IDs Poise can check against is 21k, assuming the policy has 39 contexts. As another dimension of constraint, assuming each user may launch 1k concurrent connections, then Poise would support a maximum of 1M/1k=1k users. To put this into perspective, Microsoft headquarter reports 80 k employees in 125 buildings [18]; assuming that each building has its own access control switch, then every switch needs to support 0.64 k users.

8.6 Client overhead

We now evaluate the overhead of the client module, using vanilla Android without PoiseDroid as the baseline system.

CPU overhead. We tuned the frequency at which the client module sends context packets, and measured the CPU overhead for each frequency. In a naïve design where PoiseDroid tags every packet with context information, the CPU overhead is as much as 11%. With an optimized design where the client module sends one context packet per second, the CPU overhead is drastically reduced to 1.3%. Figure 15a in Appendix A.4 shows the results.

Traffic overhead. Next, we measured the traffic overhead due to the context packets. This experiment assumes four context fields (16 bytes). We found that, at one context packet per second, the traffic overhead is less than 0.01%, a negligible amount. Figure 15b in Appendix A.4 shows the results.

Battery overhead. We used PCMark [35], a battery life benchmark tool to test smartphones and tablets, to quantify the amount of battery overhead. Table 1 in Appendix A.4 shows the results. The overall overhead across benchmarked activities introduced by PoiseDroid is only 1.02%, and even for the activities that introduce the highest overhead (i.e., writing), the overhead is only 2.87%.

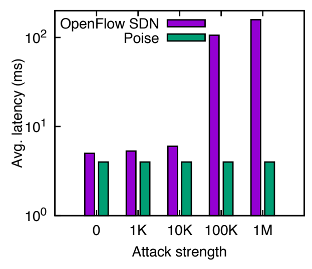
Overall benchmark. Next, we used CF-Bench, a comprehensive benchmark tool designed for multicore mobile devices, to quantify the overall overheads of PoiseDroid. This tool can further measure the overheads introduced by native code, Java code, and an overall benchmark score, where higher scores mean better performance. Figure 16 in Appendix A.4 shows that PoiseDroid only introduces 5%, 4%, and 5% additional overhead for the native, Java, and overall scores.

8.7 Poise vs. OpenFlow-based SDN

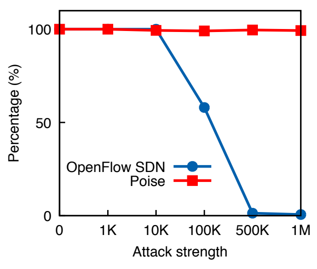
Last but not least, we compare the paradigm of programmable in-network security, as embodied in Poise, against the paradigm of OpenFlow-based SDN security, in terms of a) the speed for security decision change, and b) resilience to control plane saturation attacks [82]. As we motivated before, one key advantage of Poise over traditional SDN security solutions is the avoidance of software-based packet processing on a remote controller, because Poise uses programmable data planes to directly process context signals in hardware.

Setup. We set up a Floodlight `v1.2` SDN controller on a separate server, and configured other servers to use the controller via OpenFlow as implemented in OpenvSwitch `v2.9.2`. We implemented our example policies (P1-P7) as software "SDN apps" in the controller. These apps listen for client context updates, and push OpenFlow rules to the clients for access control. This closely mirrors the setup in state-of-the-art security solutions based on OpenFlow-based SDN [58, 75, 82].

Defense agility. We quantify the *defense agility* of a security system by measuring δ, the time it takes to change its access control decision after seeing a new context packet. For OpenFlow-based SDN, this includes the round-trip time delay



(a) New connection latency



(b) Successful connections

Figure 12: Poise is resilient to control plane saturation attacks. Attack strength is measured by the number of context changes per second that the attacker generates. In the OpenFlow-based solution, new connections and context changes would generate `PacketIn` and `FlowMod` events between the OpenFlow switch and the central controller.

for the context packet to reach the controller and for the controller to push new OpenFlow rules back to the OpenvSwitch. (We did not include the additional latency of OpenvSwitch because a hardware OpenFlow switch can reduce this significantly.) We found that, depending on the network load, the agility of the baseline system is $\delta = 5$ ms–2.47 s. In comparison, Poise directly processes context changes on the fast path, achieving $\delta < 500$ ns in all cases, which is *three to seven orders of magnitude* faster than the baseline.

Control plane saturation attacks [82]. An attacker can also create high loads on the channel between the data plane and the control plane by generating a large number of context changes. This effectively degrades the performance of legitimate users for establishing new connections, as the `PacketIn` messages go through the same channel. As Figure 12 shows, the central controller struggles to keep up with the amount of context changes that it needs to process. At an attack strength of 1M context changes per second, legitimate clients clients were not able to establish new connections (99%+ connection requests from legitimate clients were dropped; the rest experienced a latency 30× higher than normal on average). Poise, on the other hand, processes context changes entirely in the data plane at hardware speeds. The performance for legitimate clients stays *almost constant* during the attacks.

9 Related Work

SDN/NFV security. SDN/NFV-based solutions for enterprise security started with SANE [46] and Ethane [45]. Recent work also includes PSI [98], FortNox [77], PBS [58], PivotWall [75], OFX [84], and CloudWatcher [80]. Existing work has also considered new attack vectors in SDNs [59, 82, 83, 96], such as control plane saturation attacks [82]. Poise leverages the recent development of programmable data planes, and develops defenses that are resilient to control plane saturation attacks with much higher agility.

Context-aware security. Security researchers have recognized the need for context-aware security to support fine-grained, dynamic policies. Barth et al. [41] propose a logic framework for contextual integrity. Recent work has developed various applications leveraging this concept. ContexIoT [60] analyzes UI activities, app information, and control/data flow information, and prompts users for runtime permissions. FlowFence [51] runs applications in sandboxes and enforces information flow control across IoT applications. PBS [58] uses OpenFlow-based SDN for BYOD security. Yu et al. [99] sketch a vision for using network function virtualization for context-aware IoT security. DeepDroid [91] traces IPC and system calls to achieve fine-grained security. Compared to existing work, Poise designs a network primitive for security enforcement, and has an end-to-end framework for specifying, compiling, and enforcing declarative policies.

Policy languages. Most domain-specific languages for networking [38, 42, 69, 78, 79, 90, 98, 100] are not targeted at security. Policy languages for network security also exist, but we are not aware of an existing language that can support context-aware policies on programmable data planes. For instance, PSI [98] uses finite state machines to specify security policies, but it assumes that the policies are implemented by general-purpose software; PBS [58] assumes a traditional SDN environment. Poise builds upon an existing SDN language (NetCore [69]), but adapts it for enforcing context-aware security on programmable data planes.

Programmable data planes. Poise builds upon the emerging trend of using data plane programmability [43, 44, 85] for in-network computation, e.g., load balancing [65], network monitoring [73], key-value cache [62, 66], and coordination [61], but it focuses on a very different goal: security. The closest to our work is a recent workshop paper [70], but it neither has a full system implementation nor evaluation.

10 Conclusion

We have described Poise, a system that can enforce context-aware security using a programmable, efficient, in-network primitive. In Poise, administrators can express a rich set of policies in a high-level language. Our compiler then compiles the policies down to switch programs written in P4. These programs run inside modern switches with programmable data planes, and can enforce security decisions at linespeed. Our evaluation shows that Poise has reasonable overheads, and that compared to OpenFlow-based defense, it is highly agile and resilient to control plane saturation attacks.

Acknowledgments: We thank our shepherd Adwait Nadkarni, the anonymous reviewers, Vladimir Gurevich, Kuo-Feng Hsu, Dingming Wu, and Jiarong Xing for their insightful comments and suggestions. This work was supported in part by a Hong Kong RGC Project (No. PolyU 152279/16E, CityU C1008-16G) and an NSF grant CNS-1801884.

References

[1] AndFTP. http://www.lysesoft.com/products/andftp.

[2] Android for Work. https://www.android.com/enterprise/employees/.

[3] Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino.

[4] The benefits and risks of BYOD. https://goo.gl/ym9ATg.

[5] Blackberry EMM. https://www.blackberry.com/us/en/solutions/enterprise-mobility-management-emm.

[6] Bring your own risk with BYOD. https://goo.gl/bn1rN4.

[7] BYOD: A global perspective. https://goo.gl/BTrSm4.

[8] BYOD: Mobile devices threats and vulnerabilities. https://goo.gl/phTav6.

[9] Cavium XPliant. https://www.cavium.com/xpliant-ethernet-switch-product-family.html.

[10] Cisco Port ACLs (PACLs) and VLAN ACLs (VACLs). https://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst6500/ios/12-2SX/configuration/guide/book/vacl.html.

[11] End node. https://goo.gl/D99C39.

[12] How to solve the end node problem. https://goo.gl/9wWqJr.

[13] IBM Mobile: BYOD. https://goo.gl/zafGxN.

[14] IBM opens up smartphone, tablet support for its workers. https://goo.gl/WBn3vP.

[15] IEEE 802.1AE: MAC security. https://1.ieee802.org/security/802-1ae/.

[16] Intel FlexPipe. https://www.intel.com/content/www/us/en/products/network-io/ethernet/switches.html.

[17] Market reports. https://goo.gl/25SX7K.

[18] Microsoft headquarters. https://www.builtinseattle.com/2018/11/12/microsoft-redmond-campus-headquarters.

[19] P4 language repositories. https://github.com/p4lang.

[20] The Poise code repository. https://github.com/qiaokang92/poise.

[21] Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml.

[22] The rise and risk of BYOD. https://www.druva.com/blog/the-rise-and-risk-of-byod/.

[23] Samsung BYOD solutions. https://goo.gl/GmZ1io.

[24] Samsung Knox. http://www.samsung.com/global/business/mobile/solution/security/samsung-knox.

[25] Securing your weakest link: Your mobile devices. https://goo.gl/Z769MG.

[26] Security configuration guide: Access control lists, Cisco IOS XE Release 3S. https://goo.gl/zTJaUL.

[27] Symantec EMM. https://www.symantec.com/content/dam/symantec/docs/data-sheets/endpoint-protection-mobile-for-emm-en.pdf.

[28] A technical report on TEE and ARM TrustZone. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-technical-report-on-tee-and-arm-trustzone.

[29] Top 21 companies in the BYOD market. https://goo.gl/MuRr66.

[30] Verified Boot: Android Open Source Project. https://source.android.com/security/verifiedboot/.

[31] VMware Airwatch. https://www.air-watch.com/capabilities/enterprise-mobility-management/.

[32] What is BYOD and why is it important? https://goo.gl/H71Nji.

[33] IEEE 802.1x remote authentication dial in user service (RADIUS) usage guidelines, RFC 3580. 2003. https://www.rfc-editor.org/info/rfc3580.

[34] dumpsys. https://developer.android.com/studio/command-line/dumpsys, 2018.

[35] Pcmark for android benchmark. https://play.google.com/store/apps/details?id=com.futuremark.pcmark.android.benchmark, 2018.

[36] P. Aditya, R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee, and T. T. Wu. I-pic: A platform for privacy-compliant image capture. In *Proc. MobiSys*, 2016.

[37] A. Alzubaidi and J. Kalita. Authentication of smartphone users using behavioral biometrics. *IEEE Communications Surveys& Tutorials,*, 18:1998–2026, 2016.

[38] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[39] N. Apthorpe, Y. Shvartzshnaider, A. Mathur, D. Reisman, and N. Feamster. Discovering smart home Internet of Things privacy norms using contextual integrity. *Proc. IMWUT*, 2018.

[40] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android security framework: Extensible multi-layered access control on android. In *Proc. ACSAC*, 2014.

[41] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proc. IEEE S&P*, 2006.

[42] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. SIGCOMM*, 2016.

[43] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[44] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. SIGCOMM*, 2013.

[45] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[46] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. USENIX Security*, 2006.

[47] D. Dang-Pham and S. Pittayachawan. Comparing intention to avoid malware across contexts in a BYOD-enabled Australian university: A protection motivation theory approach. *Computers & Security*, 48:281–297, 2015.

[48] T. Datta, N. Feamster, J. Rexford, and L. Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

[49] S. Eberz, K. B. Rasmussen, V. Lenders, and I. Martinovic. Evaluating behavioral biometrics for continuous authentication: Challenges and metrics. In *Proc. AsiaCCS*, 2017.

[50] M. Ehatisham-ul-Haqa, M. A. Azama, U. Naeemb, Y. Amina, and J. Looc. Continuous authentication of smartphone users based on activity pattern recognition using passive mobile sensing. *Journal of Network and Computer Applications*, 109:24–35, 2018.

[51] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *Proc. USENIX Security*, 2016.

[52] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2007.

[53] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[54] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proc. ICFP*, 2011.

[55] Y. Fratantonio, C. Qian, P. Chung, and W. Lee. Cloak and dagger: From two permissions to complete control of the UI feedback loop. In *Proc. IEEE S&P*, 2017.

[56] C. K. Georgiadis, I. Mavridis, G. Pangalos, and R. K. Thomas. Flexible team-based access control using contexts. In *Proc. SACMAT*, 2001.

[57] D. Harkins. Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks. In *Proc. SensorComm*, 2008.

[58] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu. Towards SDN-defined programmable BYOD (bring your own device) security. In *Proc. NDSS*, 2016.

[59] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In *Proc. NDSS*, 2015.

[60] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *Proc. NDSS*, 2016.

[61] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soule, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[62] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[63] M. T. Jones. Invoking user-space applications from the kernel. `https://www.ibm.com/developerworks/library/l-user-space-apps/index.html`, 2018.

[64] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling packet programs to reconfigurable switches. In *Proc. NSDI*, 2015.

[65] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[66] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *Proc. ASPLOS*, 2017.

[67] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[68] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proc. POPL*, 2012.

[69] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NDSI*, 2013.

[70] A. Morrison, L. Xue, A. Chen, and X. Luo. Enforcing context-aware BYOD policies with in-network security. In *Proc. HotCloud*, July 2018.

[71] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. Time-based one-time password algorithm. RFC 6238.

[72] A. Nadkarni and W. Enck. ASM: A programmable interface for extending Android security. In *Proc. USENIX Security*, 2014.

[73] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[74] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In *Proc. TRUST*, 2010.

[75] T. OConnor, W. Enck, W. M. Petullo, and A. Verma. PivotWall: SDN-based information flow control. In *Proc. SOSR*, 2018.

[76] B. Parducci. eXtensible Access Control Markup Language (XACML) specification. 2005.

[77] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Proc. HotSDN*, 2012.

[78] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proc. HotSDN*, 2013.

[79] C. Schlesinger, M. Greenberg, and D. Walker. Concurrent NetCore: From policies to pipelines. In *Proc. ICFP*, 2014.

[80] S. Shin and G. Gu. Cloudwatcher: Network security monitoring using OpenFlow in dynamic cloud networks. In *Proc. ICNP*, 2012.

[81] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *Proc. NDSS*, 2013.

[82] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks. In *Proc. CCS*, 2013.

[83] R. Skowyra, L. Xu, G. Gu, T. Hobson, V. Dedhia, J. Landry, and H. Okhravi. Effective topology tampering attacks and defenses in software-defined networks. In *Proc. DSN*, 2018.

[84] J. Sonchack, A. Aviv, E. Keller, and J. Smith. Enabling practical software-defined networking security applications with OFX. In *Proc. NDSS*, 2016.

[85] H. Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[86] Sophos. Synchronized security: Best-of-breed defense that's more coordinated than attacks. `https://www.sophos.com/en-us/medialibrary/gated-assets/white-papers/sophos-security-heartbeat-wpna.pdf`.

[87] W. Tolone, G.-J. Ahn, and T. Pai. Access control in collaborative systems. *ACM Computing Surveys*, 37:29–41, 2005.

[88] N. Ulltveit-Moe and V. Oleshchuk. Enforcing mobile security with location-aware role-based access control. *Security and Communication Networks*, 9:429–439, 2016.

[89] VMware. Next generation security with VMware NSX and Palo Alto Networks VM-series. In *White Paper*, 2013.

[90] A. Voellmy, A. Agarwal, P. Hudak, N. Feamster, S. Burnett, and J. Launchbury. Don't configure the network, program it! Domain-specific programming languages for network systems. Technical report, Yale University, 2010.

[91] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on Android devices. In *Proc. NDSS*, 2015.

[92] X. Wang, T. Yu, O. Mengshoel, and P. Tague. Towards continuous and passive authentication across mobile devices: an empirical study. In *Proc. WiSec*, 2017.

[93] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. SIGCOMM*, 2011.

[94] R. Ward and B. Beyer. BeyondCorp: A new approach to enterprise security. *USENIX ;login:*, 39:6–11, 2014.

[95] Wi-Fi Alliance introduces Wi-Fi Certified WPA3 security. https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-wi-fi-certified-wpa3-security.

[96] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu. Attacking the brain: Races in the SDN control plane. In *Proc. USENIX Security*, 2017.

[97] R. Ye. *Android System Programming: Porting, customizing, and debugging Android HAL*. Packt Publishing, 2017.

[98] T. Yu, S. K. Fayaz, M. Collins, V. Sekar, and S. Seshan. PSI: Precise security instrumentation for enterprise networks. In *Proc. NDSS*, 2017.

[99] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *Proc. HotNets*, 2016.

[100] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for SDN policies. In *Proc. CoNEXT*, 2015.

[101] N. Zahadat, P. Blessner, T. Blackburn, and B. Olson. BYOD security engineering: A framework and its analysis. *Computers & Security*, 55:81–99, 2015.

A Appendix

In this appendix, we include more discussions and results.

A.1 Poise protocol format

In this subsection, we extend the discussion in §4.3 and describe the Poise protocol format in more detail. The Poise client module periodically sends context packets for each active connection. Context packets have the same flow tuples (source IP, destination IP, source port, destination port) with data packets from the same TCP/UDP flows. The only differences are that a) context packets have a special IP protocol number (IPProto=143 for TCP, IPProto=144 for UDP; both are unassigned protocol numbers [21]), b) context headers come after the transport-layer (TCP/UDP) header, and c) context packets do not have payload. Poise never propagates context packets to external networks but rather drops them at the switch, and it does not modify data packets. Figure 13 shows the format for TCP flows.

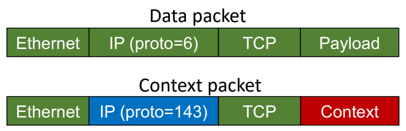


Figure 13: Context packets have a special IP protocol number. Data packets from Poise clients have unchanged headers.

A.2 Compiler optimizations

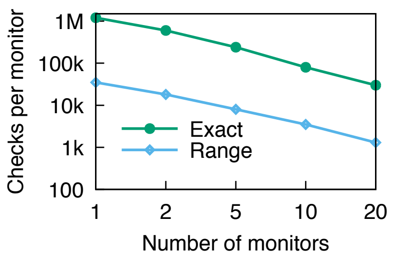
This subsection extends §4.3 and describes in more detail the compiler optimizations.

Table deduplication. Suppose that we would like to compose two policies that perform checks on the same context type. A naïve compiler would simply compile each check into a separate match/action table. With this approach, the number of policies that can be supported would be limited by the number of match/action tables in a switch. Depending on the switch model, this number is on the order of O(10), which is quite small. Our compiler can recognize that policies share the same context type, and it merges checks on the same context type by creating one table for each unique context across policies. Then, it compiles each check into a match/action table entry instead of a separate table. This optimization allows Poise to scale the number of context types to the number of table entries a switch can support, not the number of unique tables. This number is on the order of O(1M).

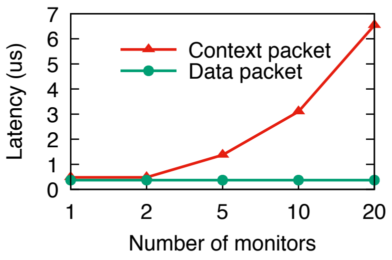
Policy collapsing. Consider now a policy that checks many context fields one by one, and only arrives at the final decision afterwards. The key challenge for handling such a policy is that these checks create "dependent tables", which due to P4 constraints must reside in separate stages. In essence, such a policy would result in a long chain of tables, which might exceed the number of available stages (O(1-10)) in a switch. Our optimization collapses a chain of tables of length k into multiple shorter chains $k_1, k_2, .., k_t$, each of which stays within the number of available stages. Due to another P4 constraint—a packet can only match against a single table per stage, matching against all subchains $k_1, k_2, .., k_t$ would require recirculating the packet t times, each for a subchain. Recirculation of context packets would cause additional latency, as such packets now need to traverse the switch multiple times before finishing processing, and also additional recirculation traffic in a dedicated switch pipeline.

A.3 Scalability

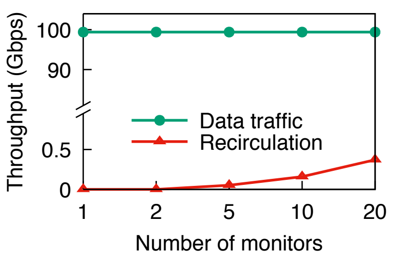
This subsection includes more results for §8.5. Figure 14 shows the scalability of Poise for monitor policies, in terms of a) the number of monitors, and the number of checks per



(a) Num. of monitors vs. num. of checks



(b) Num. of monitors vs. latency



(c) Num. of monitors vs. traffic overhead

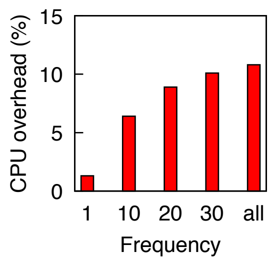
Figure 14: The scalability of Poise with monitor policies. The high-level takeaways are similar as those for regular, non-monitor policies (Figure 11 in §8.5). The only difference is that a monitor uses two tables, whereas a regular context uses one table.

monitor (Figure 14a), b) the latency of context and data packets (Figure 14b), and c) the throughput of recirculated context traffic and data traffic (Figure 14c).

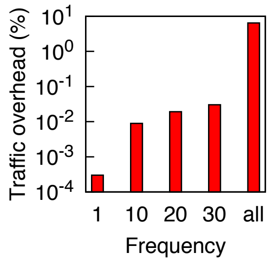
Policies could also use a mix of monitor and regular context types. At a high level, a monitor is just another type of context, except that it uses two tables instead of one. Figures 17, 18, 19, and 20 present the scalability results assuming 1, 2, 5, and 10 monitors in the policies; the rest of the available tables are used for regular contexts.

A.4 Client overhead

This subsection includes the full results for §8.6 on the client overhead due to the extra PoiseDroid module.



(a) CPU overhead



(b) Traffic overhead

Figure 15: CPU and traffic overheads of PoiseDroid under different frequencies of context packets. Baseline: Android.

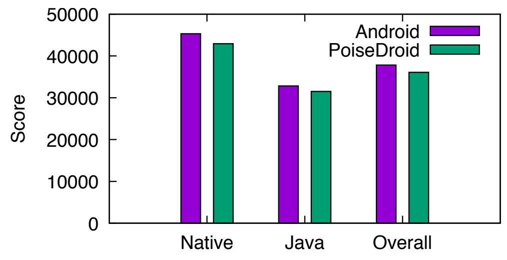


Figure 16: The overall overhead of PoiseDroid, as measured using the CF-bench benchmark tool (higher is better).

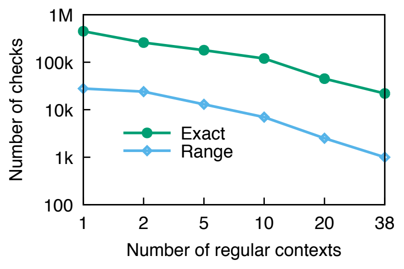
Table 1: The battery overhead of PoiseDroid (lower is better).

Attribute	Overall	Browsing	Video	Writing	Photo	Data
Android	5493	4278	5458	4530	11432	4136
PoiseDroid	5591	4303	5597	4660	11746	4145
Overhead	1.02%	0.06%	2.55%	2.87%	2.75%	0.22%

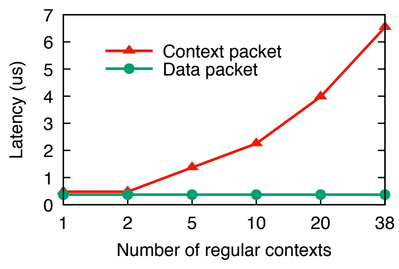
CPU and traffic overheads. Figures 15a and 15b show the CPU and traffic overheads at different frequencies of context packets. For each data point, we uploaded a video file of 1.73 GB to a remote FTP server using the mobile app `AndFTP` [1], and measured the CPU overhead as collected from the `/proc/loadavg` file. As we can see, if Poise were to tag each data packet with context information, then the CPU and traffic overheads are prohibitive ($\sim$10%). Because the in-network primitive is stateful, it can remember past decisions for each connection; this enables an optimized design where client modules can send out context packets periodically. The Poise primitive can look up its stateful data structure and apply access control decisions accordingly. For instance, at the frequency of one context packet per second, the CPU and traffic overheads are both low enough to be practical.

Battery overhead. Table 1 shows the battery overhead of the PoiseDroid client, as measured by PCMark [35]. PCMark tests capture a wide variety of activities, such as browsing, video playback, photo editing, writing, and data manipulation. In the beginning of the experiment, the phone was charged with full capacity (100%), and the tests ran until the battery dropped to less than 20%. We can see that, the highest overhead across all scenarios is only 2.87%.

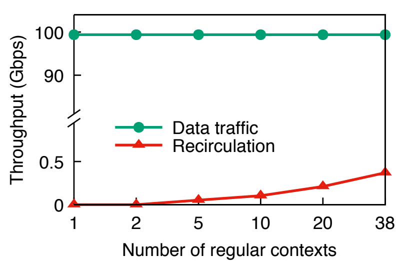
Overall benchmark. Figure 16 shows the results obtained by CF-Bench, a comprehensive benchmark tool for testing multicore mobile devices. PoiseDroid introduces 5%, 4%, and 5% additional overheads for the native code, Java code, and overall scores, compared to the baseline system of a vanilla Android system without PoiseDroid installed.



(a) Num. of contexts vs. num. of checks

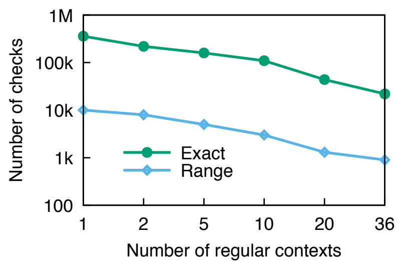


(b) Num. of contexts vs. latency

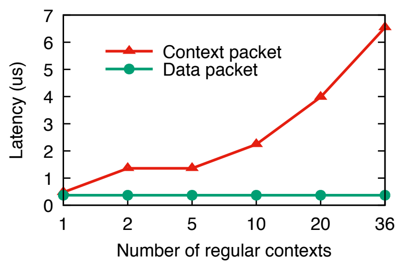


(c) Num. of contexts vs. traffic overhead

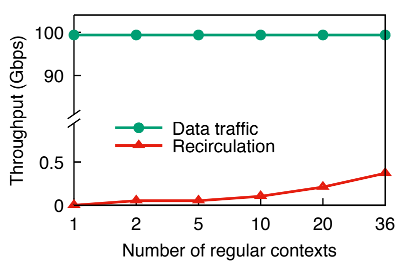
Figure 17: Scalability results for policies with one monitor and 1–38 regular contexts. The number of (exact or range) checks Poise can perform is the same for a regular or monitor context. Similarly for all figures below.



(a) Num. of contexts vs. num. of checks

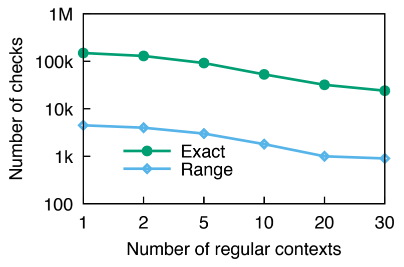


(b) Num. of contexts vs. latency

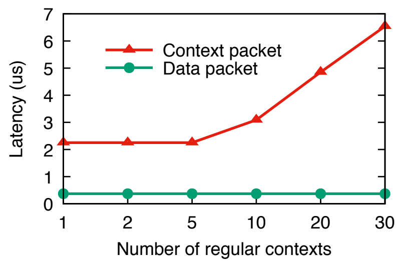


(c) Num. of contexts vs. traffic overhead

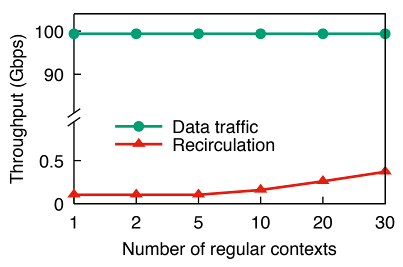
Figure 18: Scalability results for policies with two monitors and 1–36 regular contexts.



(a) Num. of contexts vs. num. of checks

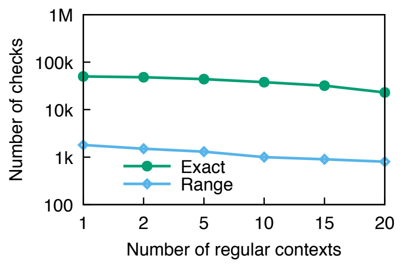


(b) Num. of contexts vs. latency

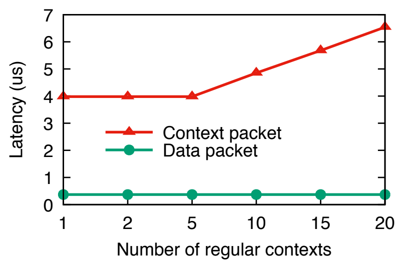


(c) Num. of contexts vs. traffic overhead

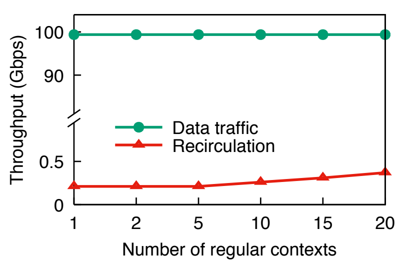
Figure 19: Scalability results for policies with five monitors and 1–30 regular contexts.



(a) Num. of contexts vs. num. of checks



(b) Num. of contexts vs. latency



(c) Num. of contexts vs. traffic overhead

Figure 20: Scalability results for policies with ten monitors and 1–20 regular contexts.