Automated Synthesis of Verified Firewalls

Abstract

We demonstrate correct-by-construction firewalls—stateful packet filters for TCP/IP packets—using the Fiat synthesis library [3]. We present a general DSL for specifying their behavior independent of algorithmic implementation. We outline the design of a verified compiler in Coq, detail a few verified efficiency optimizations, and show how the compiler can easily be extended to support custom optimizations for user-defined policies.

ACM Reference Format:
. 2020. Automated Synthesis of Verified Firewalls. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Introduction

Firewalls ("any device, software, or arrangement or equipment that limits network access" [2]) are ubiquitous in modern networked devices. They implement security policies often specified in human-readable DSLs, such as in the widespread *iptables* configuration tool for Linux [5]. This paper focuses on firewalls that access TCP and IP headers of packets to make a stateful (i.e. defined by previous packets) allow/deny decision for each incoming packet, although the possibility of payload-inspecting firewalls is not precluded. Our DSL emulates some functionality of *iptables*.

Fiat is a Coq library that enables "decompos[ing] a program into a high-level specification of its functionality and a sequence of semantics-preserving optimizations that produces an efficient, executable implementation" [3]. Crucially, the proof-producing synthesis process can be almost entirely automated, as demonstrated in the Query Structures database library supporting efficient SQL-like operations [3] and the Narcissus library that synthesizes encoders and decoders for binary formats [4]. We make use of both libraries to specify and automatically synthesize firewalls as abstract data types (ADTs) in Fiat.

Firewalls are an apt domain for this approach to synthesis. The simultaneous generation of code and proof provides an important security guarantee, and automation shifts the burden of proof from user to designer. Further, the separation of ADT specification and implementation formalizes the intuitive distinction between a firewall's policies and actions reflected in existing configuration tools.

Specifying firewalls

We model our firewall as a monolithic device, separate from the hosts on the network, that intercepts all traffic on the network. It assumes that packets are provided to it as Coq records containing decoded TCP and IP fields, payload, and a timestamp delta from the previous packet (implying a monotonic clock); this assumption is fulfilled by Narcissus.

```
Definition Firewall (fil: FilterType) :=
  rep := QueryStructure PacketHistorySchema,
  Def Constructor "Init" : rep := empty,,

  Def Method "Filter" (r: rep) (inp: packet) :=
    ts inp' <- latest <- Max (For (t in r!"History")
                              Return t!"Timestamp");
               ret (latest + (ts inp));
    res <- fil r inp';
    r <- Insert inp' into r!"History";
    ret (r, res).
```

Figure 1. Modeling firewalls as ADTs.

Given in Figure 1 is an initial specification[1] of the firewall ADT, parametric in `fil`, the actual packet filtering logic. The internal state (`rep`) is a database with schema `PacketHistorySchema` which specifies a single table, "History", with two columns, "Timestamp" and "Packet". Upon receiving a packet, the firewall computes the cumulative timestamp, the stateful allow/deny decision, and the updated state, and returns the latter two. Figure 2 is an example of how a simple stateful firewall might be specified in our DSL.

Note that we make ample use of Coq notations. Fiat provides notations to define ADTs, operate on Query Structures databases, and work within the non-determinism[2] monad with a bind (`<-`) and return (`ret`) [3]. We define notations that emulate *iptables* syntax for packet predicates and capture common specification patterns like 'count packets in last n ticks that match [...]'.

Synthesizing firewalls

Synthesizing `ExampleFirewall` from Figure 2 into an executable firewall is relatively straightforward, due to the automated proof tools included with the Query Structures library that produce verified implementations of database operations (`Max`, `Insert`, `Count`). Everything in the specification that is not a database operation is already a stateless, deterministic computation.

However, the code thus synthesized is very inefficient. Before we address the major inefficiencies, let us work through a simple optimization that illustrates our general process.

Instead of repeatedly querying the History table for the maximum timestamp, we could cache it in the internal state and update it for each incoming packet. We define a *relation* between the existing state (**`rep`**`: QueryStructure [...]`) and a new (**`rep'`**`: QueryStructure [...] * nat`), in which **`rep`** `= (fst` **`rep'`**`)` and the `Max` query over `rep` is 'refined' by returning `(snd` **`rep'`**`)`. (`B` refines `A` if the range of `B`'s behaviors is a subset of that of `A` in Fiat's computation monad.)

[1] The syntax is close to how it would be written in Coq, but simplified for presentation.
[2] The monad has a non-deterministic 'pick' operator, used for instance to choose an unspecified unique row ID when `Inserting`, but this is not relevant to this paper.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
Definition ExampleFilter (r: _) (inp: packet) :=
  c <- Count in r in last 1000 ticks (fun p =>
    (iptables --destination-port 8 --source p.src) inp);
  If (c <? 10) Then <ACCEPT> Else <DROP>.
Definition ExampleFirewall := Firewall ExampleFilter.
```

Figure 2. An example firewall specification in our DSL.

Next, we prove a theorem that if `rep` and `rep'` satisfy this relation, then we can refine the original firewall specification into one that uses and updates the cached timestamp by finite differencing, preserving the relation. And finally, with some Ltac additions to the database synthesis automation, we can apply this refinement theorem to any firewall, effecting the corresponding changes in the synthesized code.

Column reduction

Recall that the `PacketHistorySchema` specifies that the History table has two columns, "Timestamp" and "Packet", but a firewall may not need the entire contents of every packet. We can significantly improve efficiency by simply not storing packet fields that the firewall will never access. We call this optimization "column reduction". Like timestamp caching, column reduction applies to any firewall regardless of its filtering logic.

The refinement relation for column reduction states that the new History is a projection of the old one retaining only specific packet fields instead of the entire "Packet", and the refinement theorem states that this change of database schema preserves the filtering logic. This theorem is automatically proved at compile time with Ltac scripts, unlike the cached timestamp example whose proof is included in the library, because which fields the projection retains depends on the specifics of the firewall. We point out that column reduction is limited to purely *syntactic* deduction of unused fields. It is sound but not complete (e.g. does not detect semantically irrelevant fields).

Row reduction

A firewall as specified maintains an arbitrarily large history of every packet received because we want to allow the expression of policies with arbitrarily long memory. When instantiated with a particular policy, though, we can reduce space and time requirements by not storing rows in the table that the firewall will never access. This optimization—"row reduction"—does not readily admit techniques as general as for column reduction, but we present three useful refinements for common patterns.

The first is applicable when the firewall specification has an explicit bound on the history it accesses, as in `ExampleFirewall` in Figure 2. If the only query performed by the firewall accesses only the last n ticks, then the firewall will have the same behavior if, on every `Insert`, it deletes all rows older than n ticks. This refinement is a proven theorem which the compiler can automatically apply if the firewall is in the expected syntactic form.

A second opportunity for row reduction is when the firewall's only query checks for the existence of a packet `p` in History satisfying predicate `(P p inp)` where `inp` is the incoming packet. (For example, whether `p` and `inp` have the same source address.) If we have a (not necessarily injective) projection `f` such that for all `inp`, `(P p inp)` iff `(P (f p) inp)`, then the firewall will maintain its behavior if it only stores `inp` when there is no packet `p` in History such that `(f p) = (f inp)`. (In our example, an `f` that retains only the source address would work, so that only packets with unique source addresses would be stored.) Here, the refinement is a proven theorem in the automation library, but a suitable projection `f` needs to be determined at compile-time using an Ltac script very similar to column reduction.

The third heuristic generalizes the timestamp caching optimization to database queries of the form, 'return `(g p)` for the latest packet `p` that satisfies some predicate'. The refinement is to add a "Cache" table to the database where the keys are projections of packets that preserve the query predicate (as in the previous paragraph) and the values are the corresponding `g` values. The relevant proofs and automation are still in development.

Extensibility

A benefit of our approach is that it is easy to support DSL extensions with custom compiler optimizations. We present ongoing work in the case study of a *rate-limiting* firewall that does not allow more than n packets in any window of k ticks. We remark that we introduce non-determinism here by not specifying a decision if a window has fewer than n packets. Accepting packets in that case based on a `Count` is permissible, as is denying all packets.

We wish to implement rate limiting using a *token bucket filter* (TBF) (see *iptables*' TBF-based 'limit' module [1]). A TBF acquires r tokens per tick ('rate') up to a maximum of b tokens ('burst'). Upon receiving a packet, if there are any tokens in the bucket, one is removed and the packet is allowed; else it is denied. One can see that no more than $(rk + b)$ packets will ever be allowed in any window of k ticks, so a TBF with suitable values of r and b would *refine* our specification of a rate-limiting firewall. Importantly, it eliminates an expensive database in favor of a mere counter.

We expect to be able to prove the TBF refinement and automate its application in the same manner as other optimizations.

Conclusion and future work

Using the Fiat library for ADT synthesis, we have demonstrated a DSL and compiler for correct-by-construction firewalls. A range of compiler optimizations enable reasonably efficient implementation without sacrificing correctness. Additionally, our design makes it easy to support new behaviors and optimizations.

Beyond completing the development of the generalized caching and rate limiting optimizations, some work is needed to be able to synthesize firewalls for practical use. First and foremost is to build robust proof automation that ties together the various optimization components and Ltac scripts. Our goal is to produce a single user-facing tactic that performs the required sequence of refinements, applying optimizations when possible, but never failing for any firewall in the DSL. Second, our DSL can certainly be made more natural and usable for common firewall patterns.

Future work could greatly expand the practical scope of firewalls by adding new optimizations and enabling their composition. For instance, we might be able to refine certain `Count` queries into accessing a table of counters instead of a database, and compose this with rate limiting for source-specific rate limiting. Another possibility is to use column reduction to avoid decoding unnecessary packet fields at the interface level (e.g. in Narcissus).

References

[1] [n. d.]. Ubuntu Manpage: iptables — administration tool for IPv4 packet filtering and NAT. Retrieved Oct. 18, 2020 from https://manpages.ubuntu.com/manpages/precise/en/man8/iptables.8.html

[2] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. 2003. *Firewalls and Internet Security: Repelling the Wily Hacker* (2nd ed.). Addison-Wesley Professional.

[3] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *POPL'15: Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. http://adam.chlipala.net/papers/FiatPOPL15/

[4] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-By-Construction Derivation of Decoders and Encoders from Binary Formats. In *ICFP'19: Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*. http://adam.chlipala.net/papers/NarcissusICFP19/

[5] netfilter Core Team. [n. d.]. netfilter: firewalling, NAT, and packet mangling for linux. Retrieved Sep. 18, 2020 from https://netfilter.org/