PfComp: A Verified Compiler for Packet Filtering Leveraging Binary Decision Diagrams

Clément Chavanon
Inria, Univ Rennes
Rennes, France
clement.chavanon@inria.fr

Frédéric Besson
Inria, Univ Rennes
Rennes, France
frederic.besson@inria.fr

Tristan Ninet
Thales Group
Gennevilliers, France
tristan.ninet@thalesgroup.com

Abstract

We present PfComp, a verified compiler for stateless firewall policies. The policy is first compiled into an intermediate representation taking the form of a binary decision diagram that is optimised in terms of decision nodes. The decision diagram is then compiled into a Clight program. The compiler is proved correct using the Coq proof assistant and extracted into OCaml code. Our preliminary experiments show promising results. The compiler generates code for relatively large firewall policies and the generated code outperforms a sequential evaluation of the policy rules.

CCS Concepts: • **Theory of computation → Logic and verification**; • **Security and privacy → Firewalls**.

Keywords: packet filtering, verified compilation, binary decision diagrams

ACM Reference Format:
Clément Chavanon, Frédéric Besson, and Tristan Ninet. 2024. PfComp: A Verified Compiler for Packet Filtering Leveraging Binary Decision Diagrams. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24), January 15–16, 2024, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3636501.3636954

1 Introduction

Firewalls are the first line of defense to protect networks against malicious traffic. To prevent denial of service attacks, it is important to drop illegitimate traffic quickly. It is also important that legitimate traffic goes through with minimal overhead. Yet, a naive sequential evaluation of the firewall rules does not scale because the (worst-case) processing time increases linearly with the size of the policy. In real scenarios, the number of rules can be large. Manually optimizing the policies is not satisfactory because it is error-prone and any error may introduce a critical vulnerability. A natural solution is to design a dedicated compiler. However, the design of a compiler able to handle large policies and generate efficient code is not straightforward and requires using sophisticated data-structures and algorithms.

In this work, we advocate for applying aggressive compilation techniques to turn automatically a firewall policy into an efficient filtering program. To reduce the trusted computing base (TCB), the compiler is formally proved inside the Coq proof assistant and generates Clight code, an intermediate language of the CompCert compiler [14]. Our basic idea is to leverage Binary Decision Diagrams (BDDs), more precisely Reduced Ordered BDDs (ROBBD) [7], as an intermediate representation of the compiler. The motivation for using BDDs is that it ensures an efficient canonical strategy for performing all the tests needed to either accept or reject a network packet. Moreover, this strategy enjoys the good property that there are no redundant tests and the processing time only depend on the number of distinct tests rather than the number of rules.

Unfortunately, a direct use of ROBDDs is not enough to get a scalable compiler and existing verified BDD libraries have limitations. A major issue is the explosion of the number of BDD nodes. To tackle this issue, we propose several optimisations that enable our compiler to generate compact and efficient code for firewall policies made of hundreds of rules. We also extend the BDD library of Braibant *et al.*, [6] which only features state-of-the-art functional implementation of the basic Boolean BDD operations *i.e.*, $\wedge$, $\vee$, $\neg$. A first missing feature is a Garbage Collector for dead BDD nodes. Another, less standard, missing feature that is of utmost importance for our application is a `restrict` primitive [9] which allows to filter out BDD paths that are not feasible due to incompatible arithmetic constraints.

Our contributions can be phrased as follows:

- a verified compiler which compiles a firewall policy into an efficient filtering C program using BDDs as an intermediate representation;
- additions to the library of Braibant *et al.*, [6] to support a copy Garbage Collection scheme and a `restrict` primitive due to Coudert and Madre [9];
- scalability experiments of the compiler; and;
- performance evaluation of the generated code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPP '24, January 15–16, 2024, London, UK
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0488-8/24/01
https://doi.org/10.1145/3636501.3636954

The formal development and our experiments are available [1].

The rest of the paper is organised as follows. Section 2 gives some background information about BDDs and the library of Braibant *et al.*, [6]. Section 3 presents the compiler front-end which generates a BDD intermediate representation from a firewall policy; and Section 4 presents the compiler back-end which transforms a BDD into a CLIGHT program. Section 5 shows how to combine the front-end and back-end to get the full compiler. Section 6 details optimisations that are needed to get a scalable compiler. Our experimental results are shown in Section 7. Related work can be found in Section 8 and Section 9 concludes.

2 Background on ROBDDs

Binary Decision Diagrams (BDDs) are a well-known data-structure. They have proved very successful, in particular ROBBDs [7], at providing compact encoding of Boolean functions. In this section, we give a reminder of the key properties of (RO)BDDs (Section 2.1) and presents details of their implementation in the library of Braibant *et al.*, [6] (Section 2.2).

2.1 Binary Decision Diagrams

A BDD for a Boolean function f over the variables $\mathbb{V} = \{x_1, x_2, \cdots, x_k\}$ is a Directed Acyclic Graph (DAG) with two kinds of vertices. A *non-terminal* vertex (or *node*) n is labelled by a variable $vr(n) \in \mathbb{V}$, and a *terminal* vertex (or *leaf*) l is labelled by a Boolean value $b(l) \in \mathbb{B}$. Each node n has exactly two children: the *high* child ($hi(n)$) and the *low* child ($lo(n)$). Given a valuation $V : \mathbb{V} \to \mathbb{B}$, each vertex n can be assigned a Boolean $V(n)$ as follows:

$$V(n) = \begin{cases} b(n) \text{ if } n \in \mathit{leaf} \\ V(vr(n)) \text{ if } n \in \mathit{node} \end{cases}$$

The evaluation of the BDD consists in following the unique path $n_0 \dots n_m$ of the DAG such that:

- n_0 is the root of the DAG;
- n_m is a terminal node;
- the other nodes n_i ($i < m$) are non-terminal nodes;
- Each consecutive nodes n_i and n_{i+i} are such that n_{i+1} is one of the child of n_i. More precisely, we have $n_{i+1} =$ **if** $V(n_i)$ **then** $hi(n_i)$ **else** $lo(n_i)$.

For the valuation V, the value of the encoded Boolean function f is given by the value of the terminal vertex n_m.

We are using a restricted class of BDDs that are Reduced and Ordered (ROBDD). A Reduced BDD satisfies the following structural constraints:

- there are only two leaves: F such that $b(\mathsf{F}) = \bot$ and T such that $b(\mathsf{T}) = \top$;
- the children of a node n are distinct ($hi(n) \neq lo(n)$).
- the DAG does not contain isomorphic subgraphs.

In other words, the construction detects and merges subgraphs that are identical modulo renaming of the nodes. A

[1] see https://gitlab.inria.fr/cchavano/pfcomp

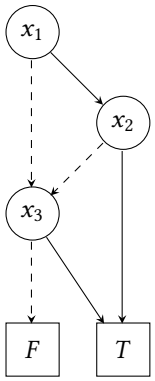


Figure 1. ROBDD for the formula $(x_1 \wedge x_2) \vee x_3$

Reduced Ordered BDD has the additional constraint that the Boolean variables are equipped with a total ordering such that, for any node n, the children vertices are labelled by lower variables *i.e.*, $vr(lo(n)) < vr(n)$ and $vr(hi(n)) < vr(n)$.

Example 2.1. Consider the ROBDD of Figure 1 which encodes the Boolean function $(x_1 \wedge x_2) \vee x_3$ for the ordering of variables $x_3 < x_2 < x_1$. The high child of node is linked by a plain arrow $\to$ and the low child by a dashed arrow $\dashrightarrow$. Consider the valuation $[x_1 \mapsto \top; x_2 \mapsto \top; x_3 \mapsto \bot]$. The corresponding path of the BDD is $x_1 x_2 \top$. Therefore, the function returns $\top$.

Note that there is a unique (up to isomorphism) ROBDD representing a function f for a given order [7, §2].

The size of ROBDDs is very sensitive to the ordering of variables. Unfortunately, for a given formula, finding the optimum variable ordering is NP-complete [5]. In practice, the ordering is improved using various heuristics [19, §6]. Those heuristics may be costly and require operations (such as projections of variables) that are not present in the Coq library described in the next section.

2.2 ROBDDs in the Coq library

The library of Braibant *et al.*, [6] features several implementations of ROBDDs. We use their deep embedding implementation. The library leverages *hash-consing* and *memoization* [10] to efficiently construct ROBDDs.

Representation of BDDs. A BDD is either a leaf or a node identifier. This is encoded by the following bdd type where $\mathbb{I}$ is the set of node identifiers.

```
Inductive bdd := F | T | N : 𝕀 -> expr.
```

A node is a triple $(l, v, h) \in \mathsf{bdd} \times \mathbb{V} \times \mathsf{bdd}$ where $v \in \mathbb{V}$ is a Boolean variable; l is the low child and h is the high child. The DAG representation of the BDD is given by the following hashcons record. In the code below, we abuse notations and write $\alpha \hookrightarrow \beta$ for the type of finite maps with keys of type α and elements of type β.

$$\begin{array}{lcl} 1 & \mapsto & (N\ 3,\ x_1,\ N\ 2) \\ 2 & \mapsto & (N\ 3,\ x_2,\ T) \\ 3 & \mapsto & (F,\ x_3,\ T) \end{array}$$

Figure 2. ROBBD graph of Figure 1

```
Record hashcons := mk_hcons {
  graph : 𝕀 ↪ node; (* BDD graph *)
  hmap : node ↪ 𝕀; (* Reverse mapping *)
  next : 𝕀 (* Fresh counter for identifiers *)
}.
```

The graph mapping encodes the DAG of the BDD and allows to retrieve the node (l, v, h) corresponding to a node identifier n. The reverse mapping hmap is used for hash-consing: each time a novel node is constructed, the algorithms check whether it already exists in the hmap. If does not, it is assigned a fresh identifier; the graph and hmap are updated with the new binding.

Example 2.2. Consider the ROBDD of Figure 1. Its corresponding graph is given by Figure 2. We can see that node identifiers make the sharing explicit. For instance, the node identifier 3 is referenced by both the left child of node identifier 1 and the left child of node identifier 2.

Semantics of BDDs. The semantics of BDDs is defined by the inductive predicate value given below.

```
Inductive value V st : bdd -> bool -> Prop :=
| value_T : value V st T ⊤
| value_F : value V st F ⊥
| value_N : ∀ n l v h,
 find n (graph st) = Some (l, v, h) ->
 ∀ b, value V st (if (env v) then h else l) b ->
 value V st (N n) b.
```

The value predicate is parameterised by a valuation V mapping variables to their Boolean value and a hash-consing state st. The BDD T (resp. F) evaluates to $\top$ (resp. $\bot$). For case value_N, suppose that the node identifier n maps to the node (l, v, h) in the state st. Depending on the valuation of the variable v, the value of the BDD (N n) is either the value of the BDD h (if V v = $\top$) or the BDD l (if V v = $\bot$). An advantage of this formulation of the value predicate is that it makes no assumption about the hash-consing state st. Yet, the library states well-formedness properties which ensure that the hash-cons state represents a ROBDD. The recursive calls of the value predicate follow the unique path leading to a leaf according to the valuation V. As a consequence, there is a unique boolean b such that value V st bdd b.

Library API. The library provides functions for computing Boolean operations over BDDs:

- mk_node builds a novel node if it doesn't already exist;
- mk_var returns the BDD for a Boolean variable;
- mk_not returns the negation of a BDD;

$$\begin{array}{lcl} \texttt{Policy} & = & \texttt{Rule}^* \\ \texttt{Rule} & = & \texttt{Cond} \times \texttt{Action} \\ \texttt{Action} \ni a & ::= & \texttt{Accept} \mid \texttt{Reject} \\ \texttt{Cond} \ni c & ::= & \lfloor\top\rfloor \mid \lfloor\bot\rfloor \mid p \mid c_1 \lfloor\wedge\rfloor c_2 \mid c_1 \lfloor\vee\rfloor c_2 \mid \lfloor\neg\rfloor c \\ \texttt{Pred} \ni p & ::= & f \bowtie i \end{array}$$

where $f \in \texttt{Field}$, $i \in \texttt{Int}$ and $\bowtie \in \{=, \neq, <, >, \leq, \geq\}$

Figure 3. Policy language

- mk_and computes the conjunction of two BBDs;
- mk_or computes the disjunction of two BBDs;

All the operations comes with a correctness statement. It takes as preconditions well-formedness conditions of the input BDD state and input BDD; and ensures that the result is also well-formed and that the BDD operation implements the right Boolean connective.

3 Compiler Front-End

3.1 Firewall Policy Language

A network packet is modelled by a mapping from fields to (unsigned 32bit) integers.

$$\texttt{Packet} = \texttt{Field} \longrightarrow \texttt{Int}$$

Though our conditions are arbitrary Boolean combinations, our language is strongly inspired from the existing Linux firewall Netfilter [3] and its associated user-space tool *iptables*. We also borrow the typical feature that firewall policies stipulate an ordering on the evaluation of rules. The abstract syntax of our firewall policies is given in Figure 3. A policy is made of a list of rules. Each rule is a pair (c, a) where c is a Boolean formula stating numeric constraints over the field of the packet and $a \in \{\texttt{Accept}, \texttt{Reject}\}$ is the action performed by the firewall. If the action is Reject, the packet is dropped by the firewall. Otherwise, it is accepted. For a given packet, the firewall sequentially evaluates the rules of the policy. As a result, the order of the rules matters and the first matching condition triggers the corresponding action. The abstract syntax does not consider wildcards that are typically used to specify ranges of addresses. This convenience feature is implemented by the parsing phase that is part of the TCB.

Example 3.1. Using our concrete syntax, a firewall policy which accepts SSH packets with source IP address **10.0.0.1** or HTTPS packets with source IP address **10.0.0.2** can be written by the following policy rules.

```
(saddr = 10.0.0.1 and dport = 22): accept;
(saddr = 10.0.0.2 and dport = 443): accept;
```

It assumes that SSH connections are using the standard port **22** and that HTTPS traffic is mapped to the port **443**. For convenience, the language supports IPv4 addresses written

using the classic notation *i.e.*, dot separating decimal numbers. A packet which does not match the rules given above is dropped by the firewall.

The semantics of a policy (see Figure 4) is given by the function `exec : Packet → Policy → Action`. It takes a policy `pol` and a packet `pk` as inputs, and returns the action to be performed on this packet. In the code, we write

```
Fixpoint exec pk pol : action :=
  match pol with
  | [] => Reject
  | (c,a) :: pol' =>
      if ⟦c⟧(|pk|) then a else exec pk pol'
  end.
```

Figure 4. Executable semantics of a packet filtering policy

$(|pk|)$: `Pred` $\to \mathbb{B}$ for an environment mapping predicates to their value for the packet pk and $[\![c]\!]_e$ for the evaluation of a Boolean condition $c \in$ `Cond` in a environment e. Therefore, $[\![c]\!]_{(|pk|)}$ corresponds to the evaluation of the condition c for the packet pk.

This `exec` function translates the fact that rules are evaluated in order with respect to the incoming packet `pk`. `exec` recursively looks for the first rule (c, a) for which $[\![c]\!]_{(|pk|)}$ holds and returns a. If no rule matches the current packet, the default `Reject` action is returned.

3.2 Logic Formulation of Policies

The first step consists in compiling a policy *i.e.*, a list of rules, into a pure logic formula. This is done by the function `pol2form` given below.

```
Fixpoint pol2form pf =
  match pf with
  | [] => ⌊⊥⌋
  | (c, a) :: pf₁ =>
    match a with
    | Accept => c ⌊∨⌋ (pol2form pf₁)
    | Reject => (⌊¬⌋ c) ⌊∧⌋ (pol2form pf₁)
    end
  end.
```

If there are no rules, the packet is rejected and therefore `pol2form` returns the formula $\lfloor\bot\rfloor$. If the first action a is `Accept`, the packet is accepted if either the current condition `c` holds or the packet is accepted by the remaining policy $\texttt{pf}_1$. If the first action is `Reject`, the packet is accepted if the current condition `c` does not holds and the packet is accepted by the remaining policy $\texttt{pf}_1$. This logic formulation accounts for the order of the rules. The correctness of the transformation is stated by Theorem 3.2.

Theorem 3.2 (`pol2form-ok`)**.**

$$\forall pf, pk.\ [\![\texttt{pol2form}\ pf]\!]_{(|pk|)} = \texttt{exec}\ pk\ pf$$

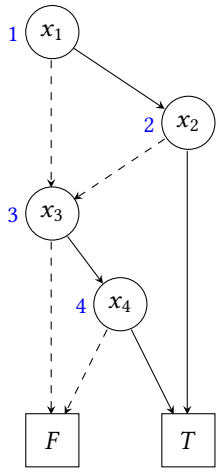
Theorem 3.2 abuses notations and assumes an implicit coercion between a Boolean $b \in \mathbb{B}$ and an action $a \in Action$ such that $b = true \leftrightarrow a = Accept$. It ensures that if a filter policy pf returns an action $a \in \{\texttt{Accept}, \texttt{Reject}\}$ for a packet pk, the condition `pol2form` pf evaluates to `true` if $a =$ `Accept` and `false` if $a =$ `Reject`.

3.3 Pure BDD Approach

Our Coq library does not provide primitives to reorder BDDs. It is therefore crucial to get the ordering right from the start. To reduce the implementation burden without sacrificing efficiency, we delegate the task of finding an optimised variable ordering to the state-of-the-art BDD library cudd [19, 20]. Given a formula, the library returns a list of predicates $[p_1; \ldots; p_n]$ with the intention that the predicate at position i *i.e.*, p_i will be assigned the variable at level i in the BDD. As a result, from the list of predicates, we construct a mapping `p2v : Pred → var` and a reverse mapping `v2p : var → Pred`. As a first step, we apply recursively the mapping `p2v` over the formula to obtain a tokenised formula which atomic formulas are just variables. This transformation of a predicate formula to a Boolean formula is proved correct for any predicate list (i.e. any predicate ordering) returned by the external cudd library. If a predicate is missing in this list, the compiler fails and does not produce code. Constructing the BDD is done by recursively applying the BDD operators corresponding to the formula constructor.

```
Fixpoint bf2bdd_rec depth bf st :=
  match bf with
  | ⌊⊤⌋ => Some (T, st)
  | ⌊⊤⌋ => Some (F, st)
  | ⌊¬⌋ϕ =>
    do (e, st₁) <- bf2bdd_rec depth ϕ st;
    (mk_not depth e st₁)
  | ϕ₁⌊∧⌋ϕ₂ =>
    do ep, stp <- bf2bdd_rec depth ϕ₁ st;
    do eq, stq <- bf2bdd_rec depth ϕ₂ stp;
    (mk_and depth ep eq stq)
  | ϕ₁⌊∨⌋ϕ₂ =>
    do ep, stp <- bf2bdd_rec depth ϕ₁ st;
    do eq, stq <- bf2bdd_rec depth ϕ₂ stp;
    (mk_or depth ep eq stq)
  | v => Some (mk_var v st)
  end.
```

The BDD library requires a `depth` parameter to ensure termination. If it is set to the maximum number of variables, the function `bf2bdd_rec` necessarily succeeds. The construction of the BDD also threads along a BDD state which is used both to store the BDD nodes but also to memoise the BDD operations. Theorem 3.3 states the correctness of the construction *i.e.*, the evaluation of the BDD is the evaluation of the formula.



(a) BDD graph

$$
\begin{array}{lcl}
1 & \mapsto & (N\ 3,\ x_1,\ N\ 2) \\
2 & \mapsto & (N\ 3,\ x_2,\ T) \\
3 & \mapsto & (F,\ x_3,\ N\ 4) \\
4 & \mapsto & (F,\ x_4,\ T)
\end{array}
$$

$$
\begin{array}{lcl}
x_1 & \mapsto & \texttt{(saddr = 10.0.0.1)} \\
x_2 & \mapsto & \texttt{(dport = 22)} \\
x_3 & \mapsto & \texttt{(saddr = 10.0.0.2)} \\
x_4 & \mapsto & \texttt{(dport = 443)}
\end{array}
$$

(b) BDD hash-consed map and v2p mapping of variables to predicates

Figure 5. BDD for the filter of Example 3.1

Theorem 3.3 (`bf2bdd_rec_correct`).

$$\forall d, \phi, n, st.\ \texttt{bf2bdd_rec}\ d\ \phi\ \varnothing = \texttt{Some}(n, st) \Longrightarrow \forall env.\ \texttt{value}\ env\ st_2\ n\ [\![\phi]\!]_{env}.$$

Proof. The proof is carried by induction over the depth d using the correctness theorems of the BDD operators. The initial empty state $\varnothing$ being generalised to a well-formed state. □

Example 3.4. Figure 5 gives the BDD associated with the policy of Example 3.1. We draw the BDD graph in Figure 5a as well as its in-memory representation in Figure 5b, composed of the hash-consed map and the mapping of variables to predicates v2p. The graph nodes are numbered.

4 Compiler Back-End

4.1 Generation of Clight

The role of the compiler back-end is to generate executable code for our BDD. We could have dumped the BDD state into a C data-structure ans prove correct a BDD interpreter written in C. Instead, to avoid the interpretation overhead, we compile each BDD node into a labelled `ifthenelse`. Our target language is Clight. Lower level language, such as the CFG-based RTL, may be an alternative. Yet, Clight expressions are convenient to express predicates. Moreover, though Clight is a structured language, it features nonetheless arbitrary `goto` statements which are essential to encode the DAG structure of the BDD.

Example 4.1. Consider the policy of Example 3.1. The graphical representation of the BDD is given in Figure 5 where each node is assigned a unique name. The produced Clight code is given in Figure 6.

The generated Clight code contains a structure `_Packet` and the firewall function `filter`. The C struct `_Packet` is

```
struct _Packet {
  unsigned int saddr;
  unsigned int dport;
};

_Bool filter(struct _Packet *packet)
{
  goto $1;
  $1: if ((*packet).saddr == 167772161U) {
    goto $2; } else { goto $3; }
  $2: if ((*packet).dport == 22U) {
    return 1; } else { goto $3; }
  $3: if ((*packet).saddr == 167772162U) {
    goto $4; } else { return 0; }
  $4: if ((*packet).dport == 443U) {
    return 1; } else { return 0; }
  return 0;
}
```

Figure 6. C code generated by `pfcomp` from the BDD of Figure 5

user-provided and may specify a precise layout of the fields using bitfield notations. The function `filter` is the compilation result of the BDD. It takes a pointer `packet` and returns `0` if the pointed packet is rejected or `1` if it is accepted. Each node $n \mapsto (l, v, h)$ contained in the BDD map is compiled into a labelled `ifthenelse` statement. There is a one-one (identity) mapping between the node identifiers and the program labels: the identifier n being compiled into the label `$n`. The `if` condition corresponds to the predicate associated with variable v in the environment v2p. The `then` statement is the compilation result of the right child h. The `else` statement is the compilation result of the left child l. Each child is compiled as follows. If it is the leaf `F`, we generate the statement `return 0`. Similarly, if it is the leaf `T`, we generate the statement `return 1`. If it is a node identifier `N n`, we generate a goto statement `goto $n`. At the entry point of the function, we also insert a goto statement to the label of the root of the BDD. We also insert as last statement a `return 0`.

Example 4.2. As an illustration, the node $3 \mapsto (F,\ x_3,\ N\ 4)$ of the BDD drawn in Figure 5 is compiled into the following statement:

```
$3: if ((*packet).saddr == 167772162U) {
      goto $4; } else { return 0; }
```

From Figure 5, the predicate mapped to variable x_3 in v2p is indeed `saddr = 10.0.0.2`. The IPv4 address `10.0.0.2` is transformed into its decimal value to become `167772162U` where the suffix `U` indicates an unsigned integer.

4.2 Back-End Correctness Proof

Theorem 4.3 states the correctness of the back-end compiler. In a nutshell, this theorem states that if the compilation function `bdd2clight` succeeds in producing a Clight program containing the function *filter*, and that the BDD evaluates from root n to a Boolean value b, then executing *filter* returns the integer value associated with b (`int_of_bool`(b) returns `1` if b = `true` and `0` if b = `false`).

Theorem 4.3 (`bdd2clight_correct`).

$$\begin{aligned}&\forall pk, v2p, b, st, n, pstruct, filter, ptr, m.\\ &\quad \texttt{packet_repr}\ pk\ pstruct\ ptr\ m \Rightarrow\\ &\quad \texttt{bdd2clight}\ v2p\ n\ st\ pstruct = \texttt{Some}(filter) \Rightarrow\\ &\quad \texttt{value}\ ((\!|pk|\!) \circ v2p)\ st\ n\ b \Rightarrow\\ &\quad (\texttt{Callstate}\ filter\ ptr\ k\ m) \rightarrow_{*}^{Clight}\\ &\qquad (\texttt{Returnstate}\ (\texttt{int_of_bool}\ b)\ (\texttt{call_cont}\ k)\ m)\end{aligned}$$

The statement of Theorem 4.3 requires some explanations. The first precondition `packet_repr` *pk pstruct ptr m* states that, in the memory m, the pointer *ptr* points to a concrete representation of the packet *pk* encoded in the C structure *pstruct*. In C-like syntax, this could be written as follows:

$$\forall f \in pstruct, ptr{-}{>}f = pk\ f.$$

Note that this precondition can be neither discharged nor dynamically enforced by the `filter` function. It is therefore the responsibility of the client code to ensure that the packet is correctly parsed into a *pstruct* and pass a valid pointer *ptr*.

`bdd2clight` compiles a BDD with root n contained in the hash-consing state *st* into the C function *filter*. There are cases where the compiler may fail, for example, if some packet fields referenced in the predicates contained in *v2p* are not present in *pstruct*. As no code is generated, this poses no practical security risk.

The `value` predicate is applied with the environment $(\!|pk|\!) \circ v2p$ which composes the evaluation of predicates $(\!|pk|\!)$ for the packet *pk* with the environment *v2p* mapping boolean BDD variables to predicates.

The conclusion of the theorem states that by executing the *filter* function using the small-step semantics of Clight, written $\rightarrow^{Clight}$, we obtain a return value `int_of_bool` b. According to the Clight semantics, the execution of a function starts from a `Callstate`. Here, the function to execute is *filter*; *ptr* is the list of arguments; k represents the code to executed after the function return *i.e.*, its continuation and m is the current memory. The `Returnstate` contains the return value that is the integer corresponding to the evaluation of the BDD, the continuation after the call and the memory updated by the function call. In our case, the memory at the function return is still the exact same memory m present at the function entry. Intuitively, this is expected because our function is side-effect free. However, in the formal model, things are more complicated and this only holds because our function does not allocate local variables and our Clight semantics for argument passing does not allocate memory.

Proof. The proof of Theorem 4.3 is carried using a classic Star Simulation [14]. To reuse the usual reasoning principle, we give an alternative definition of the evaluation of BDD (equivalent to `value`) but using a small-step presentation where a single step consists in moving from a node to its immediate child (left or right depending on the value of the variable).

The key insight of the proof is to build a simulation relation as follows. If the current explored BDD node is $n \mapsto (l, v, h)$, the C program simulation should be in a state executing a `goto $n` statement. If we reach a BDD leaf `T` (resp. `F`), the C program should execute `return 1` (resp. `return 0`). In the BDD semantics, exploring child l or h is done by finding the predicate associated to Boolean variable v, and evaluating it in function of the packet. We prove that this is equivalent to first, jump to the corresponding `ifthenelse` statement of the C code via the `goto` $n statement, then executing the `if` condition and taking either the `then` branch if we are exploring child h of the BDD node, or the `else` branch if we are exploring child l. For the whole simulation, the BDD is explored from its root to a leaf, and the compiled C program executes the corresponding `ifthenelse` statements to finally return a value encoding the leaf reached in the BDD. □

5 End-to-End Compiler

By composing the compiler front-end (see Section 3) and the compiler back-end (see Section 4.1), we get the whole compiler function `pol2clight` given below.

```
Definition pol2clight ord pf pstruct :=
  if wf_members_dec pstruct then
    do (r,st,v2p) <- formula2bdd ord pf;
    bdd2clight v2p r st pstruct
  else None
```

The compiler takes as input an oracle `ord` which returns an ordered list of predicates. The list is used as a hint for ordering the BDD variables. The oracle is implemented using an untrusted BDD library which applies reordering heuristics. The argument `pf` is the policy to compile. The remaining argument `pstruct` is a specification of the C structure used to represent the packet.

The function `wf_members_dec` checks that the C structure `pstruct` is well-formed. This amounts to verifying that there are no duplicate fields but also that each field has a meaningful type; for instance, that the width of a bitfield fits a 32 bits integer. The function `formula2bdd` takes the policy `pf` and construct a BDD using the ordering that is obtained by the oracle `ord`. Upon success, it returns the BDD root `r` but also the corresponding `hashcons` state `st` and a mapping `v2p` from BDD variables to predicates. The function `formula2bdd` is partial and the reasons for failure are the following. The oracle `ord` needs to contain all the predicates mentioned in the policy `pf` and forbid duplicates. If this is

not the case, the mapping `v2p` cannot be constructed and `formula2bdd` fails. The other reason for failure is due to function `bf2bdd_rec` (see Section 3.3) which requires as *fuel* the depth of the BDD which is given by the number of the predicates appearing in the policy. As a result, as soon as the oracle `ord` lists, only once, all the predicates of the formula, the construction of the BDD succeeds. The second stage of the compiler `bdd2clight` may also fail for essentially the same reasons *i.e.*, if the mapping `v2p` does not list all the BDD variables or if the C structure is not valid. Eventually, we get a C light function implementing the policy.

Theorem 5.1 states the correctness of the compiler.

Theorem 5.1 (`pol2clight_correct`).

$$\begin{aligned}&\forall pk,\ pf,\ pstruct,\ ptr,\ k,\ m,\ b.\\&\quad \texttt{packet_repr}\ pk\ pstruct\ ptr\ m \Rightarrow\\&\quad \texttt{pol2clight}\ pf\ pstruct = \texttt{Some}(filter) \Rightarrow\\&\quad \texttt{exec}\ pk\ pf = b \Rightarrow\\&\quad (\texttt{Callstate}\ filter\ ptr\ k\ m) \rightarrow_{*}^{Clight}\\&\qquad (\texttt{Returnstate}\ (\texttt{int_of_bool}\ b)\ (\texttt{call_cont}\ k)\ m)\end{aligned}$$

The only precondition is that the pointer *ptr* passed to the compiled function *filter* points to a valid in-memory representation of the packet *pk*. If this is the case, Theorem 5.1 ensures that the execution of the policy *pf* is the result of the execution of the Clight function *filter* produced by a successful compilation.

The proof follows from the correctness of both the front-end (see Theorem 3.3) and back-end (see Theorem 4.3) passes.

6 Optimisations

The compiler presented above show several severe inefficiencies and scalability issues. Concrete examples can be found in our experiments (Section 7).

The first problem is that the generated C code may contain a huge amount of dead code. The reason is that the back-end dumps the whole BDD state including all the BDD nodes due to intermediate BDDs. In practice, most of them are not needed by the final result. Our solution is to recursively copy the final BDD into a novel BDD state (see Section 6.1).

There are also cases where the filter policy is poorly written and contain explicit `Reject` rules which are redundant. We show that they may be removed efficiently by a preprocessing pass (see Section 6.2).

Last but not least, the number of BDD nodes could grow exponentially. To our surprise, this occurs for policies with a few hundred rules and predicates even when aggressive re-ordering is enabled. The reason is that the pure Boolean reasoning performed by BDDs does not filter out paths that are not feasible due to incompatible numeric constraints. We show how to efficiently encode some of these numeric constraints and adapt the BDD construction to exploit them (see Section 6.3).

6.1 Garbage Collection

Though it was already present in the early work of Verma *et al.*, [21], garbage collection is not always implemented in verified BDDs libraries. In particular, this feature is not supported by the library of Braibant *et al.*, [6] that we are currently using. Verma *et al.*, implement a mark and sweep collector [21, Section 5]. More ambitious is the collector of Giorgino and Strecker [11] that is based on reference counting.

For simplicity, we implement a stop-and-copy collector. In terms of BDD, this consists in a recursive copy of a BDD inside a novel empty state. The result of the copy is equal to the original BDD modulo renaming of the nodes. To preserve sharing and avoid duplicating nodes, the recursive calls of the copy are memoised.

```
Fixpoint xcopy st d e st' m :=
  match d with
  | O => None
  | S d' =>
    match e with
    | T | F => Some (e, st',m)
    | N p   =>
      match find p m with
      | Some e' => Some(e, st, m)
      | None =>
        do l, v, r <- find p (graph st) ;
        do l', st', m <- xcopy st d' l st' m;
        do r', st', m <- xcopy st d' r st' m;
        let (e, st') := mk_node l' v r' st' in
        Some (e, st', add p e m)
      end
    end
  end.
```

The function takes as input a BDD state `st` containing spurious nodes and a BDD `e` that we wish to copy into the state `st'`. The argument `d` is used for termination and the argument `m` is used to memoise the recursive calls. The result is a triple `(er, str, mr)` where `er` is the root of the copied BDD, `str` is obtained but augmenting the state `st'` with the nodes of the input BDD `e` and `mr` is obtained by augmenting the table `m` with the result of the recursive calls.

The copy works as follows. If the BDD `e` is a leaf *i.e.*, `T` or `F`, the copy is finished. Otherwise, if the BDD is a node `N p`, we look up `p` in the table `m`. If this is a hit, the copy of `p` has already been done. If this is a miss, we copy recursively the left and right child of the node. We build a new node and update the table with the result of the copy.

Theorem 6.1 states the correctness of the copy.

Theorem 6.1 (xcopy_ok)**.**

$$\begin{aligned}&\forall d, st, \phi, v_0, st', \phi', st'', m, m'.\\ &\quad wf_st\ st \wedge wf_st\ st' \wedge wf_bdd\ v_0\ \phi \wedge\\ &\quad \texttt{xcopy}\ st\ d\ \phi\ st\ st' = Some(e', st'', m')) \Rightarrow\\ &\quad wf_st\ st'' \wedge wf_bdd\ st''\ \phi \wedge\\ &\quad incr\ st'\ st'' \wedge \texttt{wf_memo_copy}\ st\ st''\ m' \wedge\\ &\quad \forall env, b.\texttt{value}\ env\ st\ e\ b \Rightarrow \texttt{value}\ env\ st''\ e'\ b.\end{aligned}$$

Compared to the mk_not BDD primitive, the difference is that we manipulate a pair of BDD states but the proof structure is otherwise similar. The premises require both states to be well-formed and that v_0 is an upper-bound on the variables of the BDD ϕ. The predicate wf_memo_copy ensures the consistency of the cache m: the BDD are well-formed and have the same evaluation in both states. For the sake of the induction, the conclusion also mentions well-formedness conditions; that the BDD state is augmented; and that the evaluation of the BDD in the copied state is the same as the original BDD in the initial state.

For our compiler, we run xcopy over the final BDD from a novel empty state. Therefore, we generate a Clight program containing only the necessary nodes.

6.2 Preprocessing

The semantics of filter policies is sequential. Moreover, when no rule applies, the packet is rejected. As a result, if the last rule is a Reject rule, it is redundant. We propose a preprocessing that goes beyond this syntactic criterion and expunge redundant Reject rules from the policy. The key insight is that a Reject rule and an Accept rule may be swapped, without affecting the semantics of the policy, if the conditions of the rules are disjoint *e.g.*, the conjunction is unsatisfiable. For predicates of the form $f \bowtie i$, we have implemented a procedure to decide whether the conjunction of two rules is unsatisfiable. This is done by computing a disjunctive normal form and checking that there at least two conflicting atoms for any conjunction. The preprocessing can therefore be implemented by the following reduce function.

```
Fixpoint is_redundant (c: cond) (pf: pfilter) :=
  match pf with
  | [] => true
  | (c',a) :: l =>
    match a with
    | Reject => is_redundant c l
    | Accept =>
      is_unsat (c⌊∧⌋c') && is_redundant c l
  end end.

Fixpoint reduce (pf: pfilter) :=
  match pf with
  | [] => []
  | (c,a) :: pf =>
    match a with
    | Accept => (c,a) :: reduce pf
    | Reject =>
      if is_redundant c pf then reduce pf
      else (c,a) :: reduce pf
  end end.
```

The reduce function iterates over the filter policy. If this is a redundant Reject rule, it is removed from the policy. The is_redundant rule check whether a (Reject) condition can be swapped with all the later rules. For Reject rules, this can be done unconditionally. For Accept rules, we check that the conjunction of the conditions is unsatisfiable.

Theorem 6.2 states the filter policies with and without removal of redundant Reject rules have the same semantic.

Theorem 6.2 (reduce_ok)**.**

$$\forall pk, pf.\ \texttt{exec}\ pk\ (\texttt{reduce}\ pf) = \texttt{exec}\ pk\ pf$$

This preprocessing has the advantage of performing only local reasoning and is therefore efficient. If all the Reject rules can be removed, a side-effect is that the order of the rules is irrelevant. Though the compiler does not exploit this property, this might be a desirable property and ease the maintenance of complex sets of rules.

6.3 Theory Reasoning

Essentially, our compiler (see Section 3.3) performs a Boolean abstraction and therefore ignores the semantics of predicates. As a result, the BDD may contain unfeasible paths that are translated into *dead code* in Clight.

Example 6.3. Consider the BDD of Figure 5b. The path $x_1x_2x_3x_4$ is not feasible because it contains the unsatisfiable conjunction (ip = 10.0.0.1) ∧ (ip = 10.0.0.2).

For policies with hundreds of rules, keeping unfeasible paths leads to a blow-up of the number of BDD nodes despite aggressive reordering heuristics. Eliminating such dead code is outside the scope of CompCert optimisations. Even if it were, the generated code would be of poor quality because the reordering of BDD would not account for arithmetic relations between Boolean variables.

6.3.1 Restriction of a BDD.

Suppose that we have a BDD ϕ which encodes the logic of the firewall policy and a BDD ψ which encodes arithmetic tautologies. We would like to get a smaller BDD ϕ' such that ϕ and ϕ' have the same evaluation when ψ holds. Note that when ψ does not hold, we do not care about the evaluation of ϕ' because this corresponds to an unfeasible path.

Definition 6.4 (restriction)**.** Given two BDDs ϕ and ψ, a restriction of ϕ by ψ (written $\phi \mid \psi$) is such that

$$\begin{aligned}&\forall env, st, r.\ \texttt{value}\ env\ st\ \psi\ true \wedge \texttt{value}\ env\ st\ \phi\ r\\ &\Rightarrow \texttt{value}\ env\ st'\ (\phi \mid \psi)\ r\end{aligned}$$

We have augmented the library of Braibant *et al.*, [6] with a restrict primitive, due to Coudert and Madre [9, Figure 1] which satisfies Definition 6.4. The implementation closely

follows the specification of Coudert and Madre based on the Shannon expansion of Boolean formulas. Alike other BDD algorithms, the `restrict` algorithms performs a case analysis over the top BDD variable of both arguments and the recursive calls are memoised.

6.3.2 Optimised Generation of BDD.

We can revise the algorithm `bf2bdd_rec` (see Section 3.3) and insert `restrict` operations to reduce the size of the BDD. We get the following `bf2bdd_res` algorithm to turn a Boolean formula into a BDD.

```
Fixpoint bf2bdd_res t d bf st m :=
  match bf with
  | ⌊⊤⌋ => Some (T, st,m)
  | ⌊⊥⌋ => Some (F, st,m)
  | ⌊¬⌋ p =>
    do e, st, m <- bf2bdd_res t d p st m;
    do ne, st <- (mk_not d e st);
    restrict d ne t st m
  | ϕ1⌊∧⌋ϕ2 =>
    do e1, st, m <- bf2bdd_res t d ϕ1 st m;
    do e2, st, m <- bf2bdd_res t d ϕ2 st m;
    do e, st <- mk_and d e1 e2 st;
    restrict d e t st m
   | ϕ1⌊∨⌋ϕ2 =>
      do e1, st, m <- bf2bdd_res t d ϕ1 st m;
      do e2, st, m <- bf2bdd_res t d ϕ2 st m;
      do e, st <- mk_or d e1 e2 st;
    restrict d e t st m
  | v => Some (mk_var v st,m)
  end.
```

The function `bf2bdd_res` takes as argument a BDD `t` that contain tautologies, a depth `d` for the termination of BDD operations, the Boolean formula `bf`, a BDD state `st` and a memoisation table `m` for tabulating the calls to `restrict`. The function `bf2bdd_res` returns a triple made of a BDD, an augmented BDD state and an augmented memoisation table.

In practice, because all the BDD are restricted, `bf2bdd_res` computes a much smaller BDD. As our experiments show (see Section 7), this significantly improves the scalability of the compiler.

6.3.3 Generation of Restriction.

For soundness, a restriction ψ is the Boolean abstraction of any tautology. For completeness, we would like the restriction to encode all the necessary arithmetic reasoning to maximally simplify all the paths of the BDD. To do that, it would be necessary to encode in a BDD all the tautological clauses. Given the number of predicates, this is not feasible. However, because our predicates are restricted to relations of the form $f \bowtie i$ for $f \in$ `Field` a field name and $i \in$ `Int` a constant integer, we conjecture that the minimal clauses can be expressed using only a pair of predicates (with the exception of an equality $x = i$ which can be encoded by the conjunction $x \geq i \wedge \neg x > i$).

Given n predicates, there are possible $4{\times}n{\times}(n{-}1)$ possible binary clauses from which we need to extract the tautological ones. For thousand of clauses, we get millions of clauses and we found that turning the million of clauses into a BDD does not scale: this is slow and the BDD is quite large. Yet, when the process completes, and BDD reordering is performed, we notice that, for certain classes of constraints, the size of the BDD is eventually linear in the number of predicates. As a result, we are optimistic that, using a predetermined order, and exploiting the structure of the clauses, the complete set of tautological clauses can be efficiently constructed. For the time being, we encode the following set of clauses

$$\begin{aligned}
&\mathit{AllDiff}(f, I) = \{\neg(f = i) \vee \neg(f = j) \mid I \ni i \neq j \in I\}\\
&\mathit{Trans}_{\geq}(f, I) = \{\neg(f \geq i) \vee (f \geq j) \mid I \ni i \geq j \in I\}\\
&\mathit{Clause}_{\geq,\leq}(f, I, J) = \{\neg(f \geq i) \vee \neg(f \leq j) \mid I \ni i > j \in J\}
\end{aligned}$$

A clause $\neg(f = i) \vee \neg(f = j) \in \mathit{AllDiff}(f, I)$ means that a field f cannot be equal to two distinct constants i and j. The conjunction of all the clauses in the sets $\mathit{AllDiff}(f, I)$, $\mathit{Trans}_{\geq}(f, I)$ and $\mathit{Clause}_{\geq,\leq}(f, I, J)$ can be efficiently built recursively assuming the set I and J are represented by lists with elements in decreasing order. A key insight of the previous algorithms is that the integers are dealt with in decreasing order. Moreover, this order corresponds to the ordering of the BDD variables representing the predicates. This is particularly visible for `AllDiff` that is given below.

```
Fixpoint AllDiff f I =
  match I with
  | [] => ⌊⊤⌋
  | i :: I =>
    (f = i) ? ⌊∧⌋_{j∈ I} ⌊¬⌋ (f = j) : AllDiff f I
  end.
```

It follows that the predicate $(f = i)$ is the top variable and therefore that the conditional expression `(f = i) ? ... : ...` consists in constructing a single BDD node. For $\mathit{Trans}_{\geq}$ and $\mathit{Clause}_{\geq,\leq}$, we also exploit the ordering to optimise the BDD operations but also generate a formula that is linear in I and J whereas there is a quadratic number of clauses. Here, we exploit that the remaining clauses can be deduced by resolution that is a purely Boolean reasoning rule.

$$\frac{\neg(f \geq i) \vee (f \geq j) \qquad \neg(f \geq j) \vee (f \geq k)}{\neg(f \geq i) \vee (f \geq k)}$$

Using those formulations and ordering the BDD variables accordingly, we can efficiently construct a tautology BDD that can be used by the `restrict` primitive during the construction of the BDD.

7 Experiments

In this section, we report our experimental evaluation of `pfcomp`. We evaluate the efficiency of our optimisations both

sport	dport	nat_sport	nat_dport	action
49900	443	28761	443	allow
43931	53	36161	53	allow
51048	445	0	0	drop

Table 1. Excerpt of the firewall log

in terms of the compiler running time but also in terms of the size of the code. We also evaluate the efficiency of the generated code. In particular, we compare `pfcomp` to `nftables`, a Linux configuration tool for Netfilter which is the modern successor for `iptables`.

7.1 Scalability of `pfcomp`

The goal of our first experiments is to assess the scalability of `pfcomp` and the benefit of the optimisations presented in Section 6. The host machine is powered by a 13th gen Intel Core i71365U CPU, 16GiB of RAM and Linux 6.6. For these experiments, we use the Internet Firewall Data Set [1] obtained from the UCI machine learning repository. The description says that "this data set was collected from the internet traffic records on a university's firewall".

Dataset Format. The log file consists in 65532 records. Each record provides information about a connection between a client machine and the Internet that is captured by a firewall. A very interesting feature is that the traffic is annotated. The log file takes the form of a CSV file and contains the following fields: i) The source and destination ports of packets sent by the client (resp. `sport` and `dport`); ii) The Network Address Translation (NAT) source and destination ports if NAT is triggered (resp. `nat_sport` and `nat_dport`); iii) The `action` performed by the firewall on the connection (either `accept`, `drop`, `deny` or `reset-both`); iv) Other data such as the number of packets sent and received, the quantity of exchanged data and the elapsed time of the connection. We give an excerpt of this log in Table 1, with the most relevant fields for our benchmarks. If no port translation occurred, fields `nat_sport` and `nat_dport` are set to 0.

Construction of Policies. For a given packet p annotated with a verdict a, we can construct a rule which matches only the packet p such that the action is `accept` if the verdict is `allow`. For instance, for the first log entry of Table 1, we get the following rule:

```
(sport = 49900 and dport = 443) and
  (nat_sport = 28761 and nat_dport = 443)
    : accept;
```

Using this simple approach, we construct policies of increasing size such that P_i is the policy which accepts the first i packets using i rules. The policies do not contain rules with the `reject` action because, as each rule matches exactly a single packet, they are proved redundant by our preprocessing

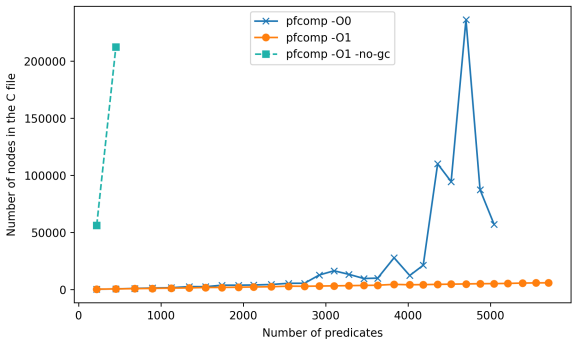


Figure 7. `pfcomp`-O1 vs `pfcomp`-O0 : number of BDD nodes w.r.t the number of predicates.

(see Section 6.2) at a negligible cost. Moreover, when building policies, we make such that the same rule only appears once.

***Evaluation of* `pfcomp`.** Each policy file is compiled with several set of options:

- `-O0` is using dynamic re-ordering of BDD variables but works on a Boolean abstraction of the predicates (see Section 3.3);
- `-O1` is performing theory reasoning and removes unfeasible paths (see Section 6.3);
- `-no-gc` disables the Garbage Collection of intermediate BDD nodes (see Section 6.1).

To get a baseline, we also compile the firewall policies using a so-called *naive compiler* which performs a sequential evaluation of the rules.

Figure 7 gives the evolution of the number of BDD nodes w.r.t the number of predicates. Given the way policies are built, the number of predicates and the number of rules are highly correlated. However, we believe that the number of predicates is more relevant as the worst-case depth of the BDDs is given by the number of predicates. For `pfcomp` without Garbage Collection (`-no-gc`), the number of BDD nodes skyrockets and we get a stack-overflow for a few hundreds of rules. This shows that this optimisation is mandatory. With re-ordering but without theory reasoning, `pfcomp` is less scalable and the number of nodes is less predictable. For instance, for 4704 predicates, we have a very high peak in the number of nodes for `pfcomp -O0` (more than 200K nodes) whereas `pfcomp -O1` generates a BDD with less than 5K nodes. Figure 8 only shows the number of BDD nodes for `pfcomp -O1`. Except for two peaks, the evolution is fairly predictable and shows a linear slope with a growth factor close to one.

Figure 9 gives the compiler running time for `pfcomp -O0`, `pfcomp -O1` and the naive compiler. Not surprisingly, the naive compiler has a negligible execution time (below 0.1s).

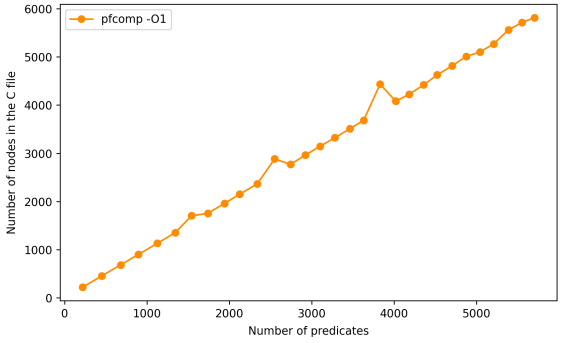


Figure 8. `pfcomp`-O1 : number of BDD nodes w.r.t the number of predicates.

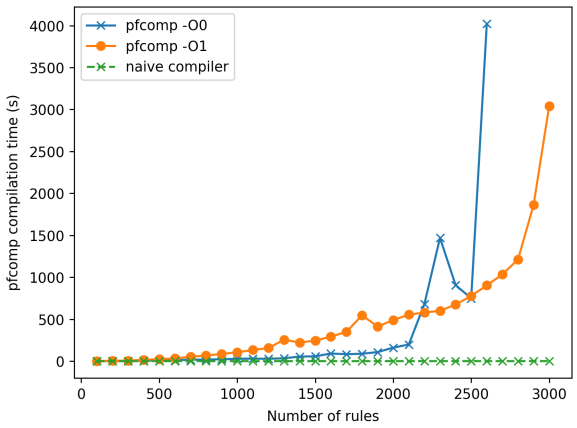


Figure 9. `pfcomp` compilation time w.r.t policy size.

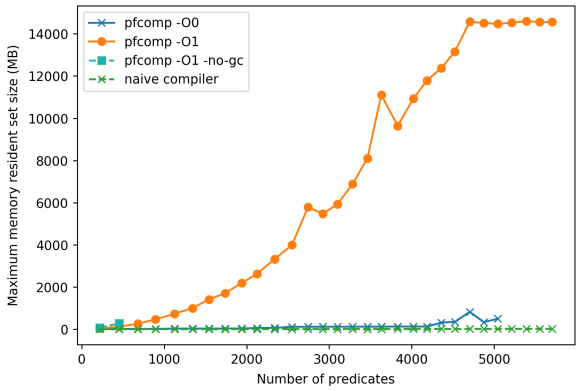


Figure 10. `pfcomp` memory usage w.r.t policy size

However, our efficiency experiments show that the generated code is not competitive (see Section 7.2). For up to 2000 rules, `pfcomp -O0` is very competitive in terms of execution time. However, for policies with more than 2700 rules, the compilation does not finish after more than an hour of computation. With `pfcomp -O1`, the compiler is somewhat slower but the compilation time is more predictable and more scalable. We do not have clear idea of the expected theoretical complexity. Yet, by extrapolating the available data, we fear that the compilation time might grow exponentially. For a policy containing 3000 rules and 5704 predicates, the compilation time of `pfcomp -O1` is around 50 minutes.

Figure 10 shows the maximum memory resident set size of the compilation with `pfcomp` with respect to the number of predicates. The version of the compiler with the `restrict` primitive (`pfcomp -O1`) reaches the maximum memory that can be allocated on the host machine from 4704 predicates. This memory consumption is significantly higher than with `pfcomp -O0`. We believe the construction of tautologies and the call to the `restrict` primitive is memory-intensive. As we only garbage collect the necessary nodes at the end of the BDD construction, the garbage collector has no influence on the memory consumption during the BDD generation. Unsurprisingly, the maximum memory resident set size is negligible for the naive compiler (it does not exceed 22 MB). The one of `pfcomp -O0` has a peak at 830 MB for 4704 predicates, which is directly related to the very high number of nodes contains in the BDD for this policy (see Figure 7). Having a Coq implementation of BDDs using less memory would also benefit the compile time for `pfcomp -O1` because accessing data in the swap memory – when the RAM is saturated – is very time consuming compared to a direct RAM access.

Note that using a verified BDD library incurs a substantial overhead for the compilation time. For instance, for the policy with 5389 predicates, the untrusted computation using the cudd library takes only 90 seconds. It takes more than 25 minutes for the verified extracted BDD code to rebuild the same BDD using the cudd-provided variable ordering. In an early stage of the project, we picked the deep embedding approach of Braibant *et al.*, [6]. At some engineering cost, we could switch to their so-called shallow embedding and expect a 30% efficiency gain.

In summary, we get a policy compiler that is scalable and predictable for most of our benchmarks. Nonetheless, there are a few cases for which we are currently unable to explain a peak in the compiler execution time. One hypothesis is that those firewall policies interact badly with ordering heuristics. Our current benchmarks may not be representative of real-life firewall policies. Yet, we believe that pushing the policy size to its limits provides insights about the scalability of our BDD-based approach.

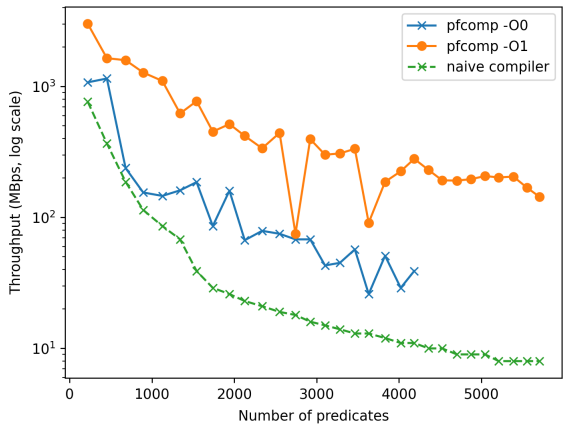


Figure 11. Throughput w.r.t. policy size.

7.2 Throughput of `pfcomp` Compiled Firewall Policies

In this Section, we evaluate the efficiency of the code generated by `pfcomp`. We first run synthetic benchmarks with the dataset described in 7.1. We then compared `pfcomp` to the state-of-the-art implemented in the Linux kernel, namely `nftables`.

Synthetic Benchmarks. From the Internet Firewall Data Set, we forge packets with four fields: source, NAT source, destination and NAT destination ports. These packets are synthetic because NAT source and destination ports are not real network packet fields. We use them to build more complex filtering rules. The packet data are written in a binary file. Each packet is 16 bytes (4 fields of 32-bit unsigned integers). We transformed the entire log file into a packet array, which contains the 65532 packets.

To make the generated `filter` function executable, we write a `main` C program that casts the binary packet data file into an array of C `struct` representing a packet. This program then iterates the `filter` function over the 65532 packets and measures the filtering time. We compute the average filtering time of 1000 executions. The Clight programs generated by `pfcomp` and this main program are given to CompCert in order to generate a binary executable.

Figure 11 gives the filtering throughput in Mega-byte per second (MBps) as a function of the number of predicates. Note that the throughput is given using a log scale. We can see that the optimised BDD-based compiler outperforms the naive evaluation of rules, as well as the result of `pfcomp -O0`. As the number of predicates grows, the performance gap widens. This can be explained by the increasing probability of predicate redundancy in the policy as the number of rules increases. The `restrict` primitive is therefore able to remove more and more nodes.

Comparison to nftables. To compare `pfcomp` to `nftables`, we generate two types of policies. The first type only contains `accept` rules with one test on the source IP address:

```
saddr = 10.B1.B2.B3: accept;
```

The second type contains rules being a conjunction of tests on the source and destination IP addresses, as written below:

```
(saddr = 10.B1.B2.B3
  and daddr = 10.B4.B5.B6): accept;
```

The "BX" bytes of IP addresses are chosen randomly when generating rules. Like for the synthetic benchmarks, we progressively increase the number of rules in the policy. The second type of policy contains predicate redundancy. We give below the pseudo-code that generates this second type of rules:

```
policy = [];
for i in 1 .. block_size:
 saddr_val = gen_addr()
 for j in 1 .. block_size:
   daddr_val = gen_addr()
   policy = (saddr = saddr_val and
      daddr = daddr_val: accept) :: policy
```

Hence every second type policy contains `block_size` blocks of rules. Each block contains `block_size` rules and rules of a same block share the same source IP address.

To compare `pfcomp` to `nftables`, we set up, on the same host machine, two network namespaces called client and server. The host machine is a quad-core 1.60Ghz CPU with 8GiB RAM running Linux 6.6. The namespaces are connected by a virtual ethernet cable. We then load the filter we want to benchmark into the `INPUT` Netfilter hook of the server namespace. This hook is triggered for incoming IP packets in server. Loading `nftables` rules amounts to using the nft tool. To load `pfcomp` rules, we compile the policy with the optimised version of `pfcomp` and compile the result with CompCert. We then link the resulting object file with a small Linux kernel module that we write and whose task is to register the filtering code into Netfilter. After the filter (either `nftables` or `pfcomp`) is loaded, we run a throughput test from client to server namespace using the `iperf3` tool. This tool generates network traffic and makes performance measurements. The addresses of client and server are set so that packets flowing from client to server hit the worst-case performance of the filter (i.e. the longest BDD path for the `pfcomp`-compiled policies).

Comparison results. The results of our benchmarks are presented in Figure 12. We draw the graph of the throughput (expressed in Giga-bit per second, Gbps) in function of the number of distinct predicates (*i.e.*, without counting duplicates) in the policy. These benchmarks compare the optimised version of `pfcomp` to nftables, either using classic rules (`nft`) or using a named set [2] (`nftset`). `nft` simply goes through each rule sequentially. When using a named

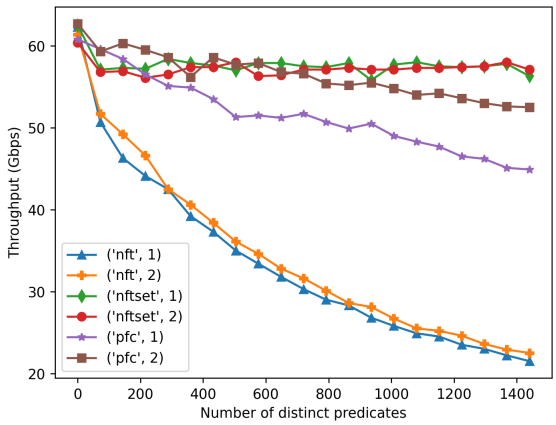


Figure 12. Throughput (Gbps) w.r.t the number of distinct predicates.

set, we first create an empty set whose elements will be IP addresses in the case of a first type policy, or couples of IP addresses in the case of a second type policy. Loading a rule then amounts to adding an element to this set. Each flowing packet will have its IP address(es) checked against the set to determine if it is accepted or not. Named sets are implemented using hash tables, so lookup is in $O(1)$. The graph legend of Figure 12 reports the name of the evaluated compiler and the type of policy (identifier 1 for rules with one predicate, identifier 2 for rules with two predicates). The policy with 0 rules can be seen as a control test, yielding the maximum available throughput.

Let p be the number of distinct predicates, b the block size, c the number of comparisons performed by the filter, f the filtering time and t the throughput. We have $t = 1/f = O(1/c)$. `nftset` achieves constant throughput thanks to its use of hash tables. For a type 1 policy, `nft` and `pfc` both make p comparisons in the worst case, hence have $\mathtt{t} = O(1/\mathtt{b})$. We see however on the graph that throughput drops much faster for `nft` than for `pfc`. We believe that this performance gap is a direct result of the implementation differences. For a type 2 policy, we have $\mathtt{p} = \mathtt{b}^2 + \mathtt{b}$. `nft` makes b^2 comparisons, hence has $\mathtt{t} = O(1/\mathtt{p})$. `pfc` makes $2 \times b$ comparisons, hence has $\mathtt{t} = O(1/\sqrt{(\mathtt{p})})$.

This benchmark shows how `pfcomp` outperforms `nft`, especially when redundancy occurs in the policy. However, `pfcomp` still has higher time complexity than `nftset`.

8 Related Work

Though not verified, several compiler strategies are using variant of Binary Decision Diagrams (BDD). Hazelhurst *et al.*, [12] compile firewall policies into Reduced Ordered Binary Decision Diagrams. A packet is seen as a bitfield of some finite width n *i.e.*, $Packet = \mathbb{B}^n$. With this representation, a policy is a Boolean function $Pol = \mathbb{B}^n \to \mathbb{B}$ that can therefore be encoded as a BDD. The conclusion of the experiments is that the efficient evaluation of the BDD requires dedicated hardware to access the packet at the bit-level and that general CPUs perform rather poorly. Our Boolean variables have a different interpretation and represent predicates of the policy (not a bit of the packet). To get compact filters, Christianson and Fleury [8] compile policies into Multi-Terminal Interval Decision Diagram (MTIDD). MTIDD have native support for interval predicates. A drawback is that more general predicates cannot be encoded. It would be tempting to build a policy compiler based on MTIDD but this would require to start from scratch as there is no verified MTIDD library available. As MTIDD are based on algorithmic principles similar to BDD (variable ordering, sharing and memoisation), we could reuse reasoning principles of the BDD library.

In the context of Software Defined Network, Smolka *et al.*, [18] introduce Forwarding Decision Diagrams (FDD), an extension of ROBDD where the variables are interpreted as equality constraints between a packet field and a constant. They revise all the BDD algorithms to account for equality constraints. Using the `restrict` primitive, we get a more general and less invasive approach where (arithmetic) tautologies are encoded into a separate BDD that is used to filter out redundant paths. A drawback of using `restrict` is that redundant paths are not eliminated on the fly but after the fact. An alternative would be to weave the `restrict` logic into the existing BDD primitives.

There has been several formalisations of BDD operations in proof-assistants [6, 11, 13, 17, 21]. ROBDD are imperative data-structures. Yet, they leverage two programming techniques that are familiar to functional programmers: *hash-consing* and *memoisation* [10]. To our knowledge, Verma *et al.*, [21] present the first purely functional implementation of BDDs in a proof assistant. A notable feature is support for garbage collection of unused BDD nodes. Braibant *et al.*, [6] have proposed several verified implementation of BDDs varying in the modelling of hash-consing. We are using their deep embedding implementation which is using a state monad for memoization and is free of axioms. In Isabelle/HOL, there are also several formalisation of BDDs but with a focus on imperative code [11, 13, 17]. The most comprehensive verified BDD library is due Giorgino and Strecker [11]. It is imperative and features memoisation and garbage collection. Imperative BDDs are more efficient. Yet, our purely functional implementation has a reasonable efficiency.

Berkely Packet Filter (BPF) [15] is a minimal specialised in-kernel virtual machine geared toward efficient packet filtering. Begel, MacCanne and Graham [4] propose a comprehensive tool chain to compile a high-level filter policy language into an efficient BPF program. They consider traditional SSA-based compiler optimisations such as Partial Redundancy Elimination (PRE) [16, 23] that is able to factorise redundant

conditions. Our BDD intermediate representation is easier to reason about and the `restrict` primitive performs optimisations that are similar to PRE. On the verification side, Jitk [22] is a verified BPF compiler that is using the CompCert [14] compiler as back-end. To ease the writing of policies, they also propose a DSL. Their rules do not feature arbitrary logic connectives and are therefore simpler than ours. In their approach, all the optimisation are performed by CompCert. This contrasts with our compiler front-end which performs aggressive optimisation leveraging BDD construction and transformation.

9 Conclusion

We propose a compiler for firewall policies leveraging a BDD based intermediate representation. The optimisation exploit the strength of the BDD that provide a compact encoding of Boolean formulas providing the existence of a good variable ordering. We also exploit the semantics of predicates to remove redundant conditions and paths in the BDD. The compiler is fully proved and target the Clight language of CompCert.

As future work, several extensions are worth being investigated. On the policy language side, we could extend the set of available predicates by featuring richer expressions beyond fields and integers. The challenge would be to extend the set of tautology clauses in order to still filter out redundant paths. On the BDD side, we worked at optimising the size of the BDD. Another relevant metric would be to minimise the depths of the paths taking into account a distribution of the possible packets. This would allow to optimise the average filtering time at the cost of increasing the code size.

As conclusion, our work shows that fully verified implementations of security critical code can be competitive with state-of-the-art implementation. In general, verified code trades efficiency for simplicity. We are doing the same and our compiler speed could probably be optimised further. In our setting, we are lucky enough that we avoid costly heuristic search, *e.g.*, variable ordering, can be outsourced to untrusted libraries. Moreover, the compiler speed is not the critical factor as our firewall are fixed and compiled once and for all.

References

[1] 2019. Internet Firewall Data. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5131M.

[2] 2023. *nftables sets.* https://wiki.nftables.org/wiki-nftables/index.php/Sets

[3] NetFilter authors. 2023. *NetFilter.* https://www.netfilter.org/

[4] Andrew Begel, Steven McCanne, and Susan L. Graham. 1999. BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In *SIGCOMM*. ACM, 123–134.

[5] Beate Bollig and Ingo Wegener. 1996. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Computers* 45, 9 (1996), 993–1002.

[6] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and Reasoning About Hash-consed Data Structures in Coq. *J. Autom. Reason.* 53, 3 (2014), 271–304.

[7] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.

[8] Mikkel Christiansen and Emmanuel Fleury. 2004. An MTIDD Based Firewall. *Telecommun. Syst.* 27, 2-4 (2004), 297–319.

[9] Olivier Coudert and Jean Christophe Madre. 1990. A Unified Framework for the Formal Verification of Sequential Circuits. In *ICCAD*. IEEE Computer Society, 126–129.

[10] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *ML*. ACM, 12–19.

[11] Mathieu Giorgino and Martin Strecker. 2012. Correctness of Pointer Manipulating Algorithms Illustrated by a Verified BDD Construction. In *FM (LNCS, Vol. 7436)*. Springer, 202–216.

[12] Scott Hazelhurst, Adi Attar, and Raymond Sinnappan. 2000. Algorithms for Improving the Dependability of ll ll and Filter Rule Lists. In *DSN*. IEEE Computer Society, 576–585.

[13] Sava Krstic and John Matthews. 2002. Verifying BDD Algorithms through Monadic Interpretation. In *VMCAI (LNCS, Vol. 2294)*. Springer, 182–195.

[14] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[15] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*. USENIX Association, 259–270.

[16] Frank Mueller and David B. Whalley. 1992. Avoiding Unconditional Jumps by Code Replication. In *PLDI*. ACM, 322–330.

[17] Veronika Ortner and Norbert Schirmer. 2005. Verification of BDD Normalization. In *TPHOLs (LNCS, Vol. 3603)*. Springer, 261–277.

[18] Steffen Smolka, Spiridon Aristides Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. In *ICFP*. ACM, 328–341.

[19] Fabio Somenzi. 1999. Binary decision diagrams. In *Calculational system design (NATO Science Series F: Computer and systems sciences, Vol. 173)*. IOS Press, 303–366.

[20] Fabio Somenzi. 2016. CUDD: CU Decision Diagram Package 3.0.0. online. https://framagit.org/nberth/mlcuddidl/-/archive/3.0.7/mlcuddidl-3.0.7.tar.gz

[21] Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. 2000. Reflecting BDDs in Coq. In *ASIAN (LNCS, Vol. 1961)*. Springer, 162–181.

[22] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*. USENIX Association, 33–47.

[23] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210.

Received 2023-09-19; accepted 2023-11-25