Reconciling multiple IPsec and firewall policies

Tuomas Aura, Moritz Becker, Michael Roe, Piotr Zieliński

Microsoft Research

Abstract. Manually configuring large firewall policies can be a hard and error-prone task. It is even harder in the case of IPsec policies that can specify IP packets not only to be accepted or discarded, but also to be cryptographically protected in various ways. However, in many cases the configuration task can be simplified by writing a set of smaller, independent policies that are then reconciled consistently. Similarly, there is often the need to reconcile policies from multiple sources into a single one. In this paper, we discuss the issues that arise in combining multiple IPsec and firewall policies and present algorithms for policy reconciliation.

1 Introduction

We would like to develop software tools that make it easier for system administrators to correctly configure IPsec:

- IPsec policies are typically configured using the same representation that is used internally by the OS kernel for the IPsec operation. This representation has not been designed for usability. It easy to make mistakes in the policy configuration and to allow accesses that one wanted to deny, and vice versa. We would like to provide some alternative means of specifying the security policy — one that is easier to understand, and harder to get wrong — and use it to automatically generate the policy that the operating system uses internally.
- Mobile devices move regularly between networks and security domains such as office, home, and cellular networks. Since the IPsec policy on the mobile device is typically configured by a single administrator, it protects only communication within one domain. For example, business laptops are usually not configured to use IPsec when communicating with the user's home PC. We would like be able to combine policies from two or more security domains, unless they are inherently in conflict with each other. (We are just concerned with the policies. Each domain must have its own means of authentication, which may also require configuration, but that is outside the scope of this paper.)
- Applications, such as a web server, and even kernel-level protocols, such as Mobile IPv6[4], may require changes to the local IPsec policy when they are installed. It is impossible for a system administrator to anticipate all such policies. Thus, it is necessary to compose policies defined by the administrator, local user, and various applications. We would like to do this

policy composition automatically and in a provably correct way, rather than manually by the administrators.

These objectives have led us to consider the problem of *reconciling* policies: given two or more security policies, how do we automatically generate a combined policy that meets the requirements of all of them? In the rest of this paper, we will describe an algorithm for reconciliation and give a proof of its correctness. Using this algorithm, complex policies can be constructed by combining simple building blocks. For example, a system administrator could write a separate policy for each service that a machine provides, and then reconcile them to form a policy for the machine. When security policies are generated by instantiating templates, as is commonly done in large systems, the reconciliation algorithm allows us to handle machines that have multiple roles: instantiate the template for each role separately, and then reconcile the results.

The reconciliation algorithm can also be used when a network administrator sets a policy for every machine on a network, but the administrator of each local machine is permitted to add additional constraints: reconcile the policies specified by the two levels of administration. If necessary, this can be extended to more than two policy sources, so that we could reconcile policies set by the network administrator, the local machine administrator, the administrator of a visited network (e.g., at home), the user, and the installed applications.

2 IPsec and firewall policies

In the IPsec architecture[5], the security policy is modelled as an ordered list of ⟨selector, action⟩ pairs. This list is known as the *security policy database* (SPD). Each packet sent or received is compared against the selectors to find the first one that matches, and then the corresponding action is taken. The possible actions are:

- BYPASS - pass the packet through without modification
- DISCARD - drop the packet
- PROTECT - apply some form of cryptographic processing, such as encryption or decryption

PROTECT is a family of actions, rather than a single action: sending packets through a secure tunnel to gateway A is different from sending them through a similar tunnel to gateway B. In the same way, encrypting for confidentiality is a different action from adding a MAC for integrity, and encrypting with DES is a different action from encrypting with AES.

The order of the SPD entries matters. Suppose that a particular packet matches the selectors in two SPD entries, one with an action of BYPASS and another with an action of DISCARD. The action that is taken depends on which of the entries appears first.

3 Extended policies

The security policy database specifies a single action to be taken for each possible packet. This is what is needed to enforce the policy at run-time: the IPsec implementation is passed a packet, it looks up the corresponding action, and carries it out. We have found that reconciliation needs extra information about the policy.

Suppose that we defined the reconciliation of polices p_A and p_B to be a policy that takes the same action as both p_A and p_B. Then it would be possible to reconcile two policies if and only if they specify the same action in all situations. This isn't very useful.

If actions could be ordered in such a way that a "higher" action always met all the requirements that were met by a "lower" one, then we could reconcile two policies by taking the least upper bound of the action specified by each.

Unfortunately, it is not possible to order actions in this way. Security policies can express both *safety* properties (packets of this form must not be accepted, because otherwise the system would be vulnerable to an attack) and *liveness* properties (packets of this form must be accepted, because otherwise the system would not be able to fulfil its function). If we are just given an action, we cannot always tell if it relates to a safety or a liveness property, and we cannot tell which alternative actions would also provide that property.

Suppose that the action specified by a policy is BYPASS. If this policy is expressing a liveness property (packets of this form must be accepted), then DISCARD is not an acceptable substitute. If it is expressing a safety property (packets of this form can be accepted without compromising security), then DISCARD is acceptable. Similarly, if a DISCARD action is expressing a safety property (packets of this form must be discarded), then BYPASS is not an acceptable substitute. But if the DISCARD action was expressing a liveness property (the system will continue to work even if packets of this form are dropped), then BYPASS is OK. This means that we cannot order DISCARD and BYPASS (or even the various PROTECT actions) in such a way that one is always an acceptable substitute for the other.

To capture the additional information, we extend the SPD format to specify a set of allowed actions, rather than a single action. The set contains every possible action that would be acceptable — i.e. would not prevent the system from working and would not make it vulnerable to an attack. Thus, we have to consider all subsets of DISCARD, BYPASS, and the various PROTECT actions.

With this extra information, we can now define the reconciliation of two policies: for all possible packets, the allowed actions under the reconciled policy must also be allowed under each component policy.

These extended policies cannot be directly used by the IPsec implementation. To make them usable, we need to choose just one of the permitted actions to be *the* action that is actually taken. This choice could be made at random, but this would not take into account the fact that there is often a strong preference between different permitted actions. In a later section, we will consider how to choose the "best" of the permitted actions.

There is a more serious problem with choosing one of the permitted actions: the intersection of two non-empty sets can be empty. For example,

$$\{\text{BYPASS}\} \cap \{\text{DISCARD}\} = \emptyset$$

Here, we cannot choose an action from the (empty) set of permitted actions, and reconciliation *fails*. There is a genuine conflict between the policies — one says that packets of a certain form must be accepted, and one says that they must be discarded. If this happens, our algorithm can output the ranges of packet headers that cause a conflict.

4 Preferences

When a policy permits more than one action, there may still be reasons for preferring one action over another. For example, BYPASS might be preferable to PROTECT because the system runs faster without encryption. Alternatively, PROTECT might be preferable to BYPASS because it gives a higher level of security, even if BYPASS meets the policy's minimum acceptable level of security. As we cannot place the actions in preference order without additional information, we need to extend the SPD format still further to include information about preferences.

Suppose that we have reconciled N policies, and now wish to choose (for each possible packet) a single action from the set of permitted actions. We would like to take into account the preferences of each of the N policies. This is like holding an election, with each of the N policies being a voter. Any existing voting scheme can be used. We do not advocate a particular scheme, because there is no one scheme that is best in all circumstances[1].

It may be the case that the user considers some of the reconciled policies to be more important than others. In this case, the important policies can be given strict priority or votes with greater weight.

If we do not include preferences, policy reconciliation is associative and commutative. We can reconcile A, B and C by reconciling A with B, treating the result as a single policy (retaining the action sets, rather than choosing a single action), and reconciling it with C. This property is useful when we wish to reconcile several administratively-imposed policies, treat the result as a single policy, and transmit it to client machines which reconcile it with one or more user-specified policies.

If we add preferences, the associative property may no longer hold (depending on which voting scheme is chosen). With some voting schemes, the result of an election with voters A, B and C cannot be calculated by merging A and B's preferences into a single hypothetical voter and then holding an election with this voter and C. This has implications for the data structures we use to represent the intermediate steps in the reconciliation of three or more policies: one possible approach is keep track of the preferences in each of the component policies, and hold a single election right at the end.

5 Basic requirements for reconciliation

In this section, we briefly summarise the informal requirements for policy reconciliation.

- Component policies may specify multiple allowed actions for a packet based on its headers. The reconciled policy may also specify multiple actions but, before installing the policy into the IPsec implementation, one of them must be selected as the unique allowed action.
- The allowed actions represent absolute requirements. When a packet is processed by the reconciled policy, the result must conform to every one of the component policies. If the component policies have conflicting requirements, the reconciliation fails.
- In addition to absolute requirements, policies may also specify preferences or priorities. These may be used to select the unique action if there are multiple possibilities. Note that most existing policy-specification mechanisms cannot express preferences.

6 Reconciliation theory

This section presents the theoretical justification for our policy reconciliation algorithm. Theorem 26 is the main result of this paper as it proves the correctness of a simple yet non-obvious algorithm. An impatient reader may want to take first a look in the example in the appendix.

An IPsec policy maps IP packets to actions based on their headers. In this section, the header and action spaces are treated as unstructured sets. The set of all IP headers is denoted by $\mathbf{H}$ and the set of all actions by $\mathbf{A}$. Our definition of policy actions differs from existing IPsec implementations in that there can be multiple allowed actions for each packet.

Definition 1 (policy). A *policy entry* is a pair $\langle s, a \rangle$ where $s \subseteq \mathbf{H}$ is a *selector* and $a \subseteq \mathbf{A}$ is the set of *actions*. A *policy* is a sequence of policy entries $p = \langle e_1, e_2, \ldots, e_n \rangle = \langle \langle s_1, a_1 \rangle, \langle s_2, a_2 \rangle, \ldots, \langle s_n, a_n \rangle \rangle$ where $\cup_{i=1}^{n} s_i = \mathbf{H}$. n is called the *length* of p. □

In order to define the refinement and equivalence of policies, we need to define how the policy maps IP packets, based on their headers, to actions. The allowed actions for the packet are determined by the first policy entry that matches the packet header.

Definition 2 (matching entry). A policy entry $\langle s, a \rangle$ matches a header $h \in \mathbf{H}$ iff $h \in s$. □

Definition 3 (catching entry). Let $p = \langle e_1, \ldots, e_n \rangle$ be a sequence of policy entries. Let $h \in \mathbf{H}$ be a header. If h matches e_i and it does not match any e_j with $j < i$, we say that the ith policy entry in p *catches* h. □

Definition 4 (allowed actions). Let $p = \langle e_1, \ldots, e_n \rangle$ be a sequence of policy entries. If $e_i = \langle s, a \rangle$ catches $h \in \mathbf{H}$, we say that the *allowed actions* for h are $\text{Allowed}(p, h) = a$. If there is no policy entry in p that catches h, then we denote $\text{Allowed}(p, h) = \perp$. □

Note that, in order to accommodate fragments of policies, the above two definitions refer to a sequence of policy entries rather than full policies.

Definition 5 (equivalence). Two policies p and p' are *equivalent* iff $\text{Allowed}(p', h) = \text{Allowed}(p, h)$ for all $h \in \mathbf{H}$. □

Definition 6 (refinement). A policy p' *refines* p iff $\text{Allowed}(p', h) \subseteq \text{Allowed}(p, h)$ for all $h \in \mathbf{H}$. □

Definition 7 (implementability). A policy is *implementable* iff $\text{Allowed}(p, h) \neq \emptyset$ for all $h \in \mathbf{H}$. □

The following lemma follows directly from the definitions of policy and catching entry.

Lemma 8. *Given a policy and $h \in \mathbf{H}$, there is a policy entry that catches h.* □

Lemma 9. *Let $p = \langle e_1, \ldots, e_n \rangle$ be a policy and let $p' = \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$ be a sequence of policy entries obtained from p by removing the ith entry. Let $h \in \mathbf{H}$. If e_i does not catch or does not match h in p, then* $\text{Allowed}(p, h) = \text{Allowed}(p', h)$. □

Proof. Let p, p' and h be as in the lemma and assume that e_i does not catch h in p. By lemma 8 there is some e_j that catches h where $j \neq i$. In both p and p', e_j is the first entry that matches h. One can see this by considering both situations where $j < i$ and $j > i$. If $j < i$, then it does not matter whether e_i matches h or not because it is not the first matching entry anyway. On the other hand, if $j > i$, then e_i cannot match h. In neither case is the first matching entry changed by the removal of e_i.

Lemma 10. *Removing one or more policy entries that that do not catch any headers produces an equivalent policy.* □

Proof. A selector that does not catch any headers has no effect on the union of selectors. Thus, the union remains equal to $\mathbf{H}$ when some such policy entries are deleted. The equivalence follows directly from lemma 9.

RFC 4301 defines the concept of decorrelation. The idea is that if the selectors in the policy are independent of each other, then the order of the policy entries does not matter.

Definition 11 (decorrelation). Let $p = \langle\langle s_1, a_1\rangle, \langle s_2, a_2\rangle, \ldots, \langle s_n, a_n\rangle\rangle$ be a policy. p is *decorrelated* iff $s_i \cap s_j = \emptyset$ for all $1 \leq i < j \leq n$. We denote by $\mathrm{Decor}(p)$ the following function:

$$\mathrm{Decor}(p) = \langle\langle s_i^*, a_i\rangle \mid s_i^* = s_i \setminus \cup_{j=1}^{i-1} s_j \text{ and } i = 1 \ldots n\rangle$$

□

$\mathrm{Decor}(p)$ is the obvious way of converting policies to equivalent decorrelated ones. This is verified by the following lemma.

Lemma 12. *If p is a policy,* $\mathrm{Decor}(p)$ *is a decorrelated policy.* □

Proof. Let $p = \langle\langle s_1, a_1\rangle, \ldots, \langle s_n, a_n\rangle\rangle$ be a policy. We show first that $\mathrm{Decor}(p)$ is a policy. $\cup_{i=1}^{n} s_i^* = \cup_{i=1}^{n}(s_i \setminus \cup_{j=1}^{i-1} s_j) = \cup_{i=1}^{n} s_i$. Since p is a policy this is equal to $\mathbf{H}$ and, thus, $\mathrm{Decor}(p)$ is a policy.

Next, we show that $\mathrm{Decor}(p)$ is decorrelated. Consider any $s_i^* = s_i \setminus \cup_{j=1}^{i-1} s_j$ and $s_l^* = s_l \setminus \cup_{j=1}^{l-1} s_j$ with $i < l$. Then, $s_i^* \subseteq s_i \subseteq \cup_{j=1}^{l-1} s_j$, which does not intersect with s_l^*.

If p is a decorrelated policy, then $\mathrm{Decor}(p) = p$. We can also prove the following two lemmas to show that the equivalence of policies is preserved by decorrelation and by arbitrary reordering of the policy entries in the decorrelated policy.

Theorem 13. *Any policy p is equivalent to* $\mathrm{Decor}(p)$. □

Proof. Let p be a policy and h a header. The actions in the ith entries of p and $\mathrm{Decor}(p)$ are the same for any i. Thus, it suffices to show that the same (ith) entry in both policies catches h. If the ith entry in p catches h, it means that $h \in s_i$ and $h \notin s_j$ for $j = 1 \ldots i-1$. This is equivalent to $h \in s_i \setminus \cup_{j=1}^{i-1} s_j$, which is the selector of the ith entry in $\mathrm{Decor}(p)$. Since $\mathrm{Decor}(p)$ is decorrelated, this can happen if and only if the ith entry in $\mathrm{Decor}(p)$ catches h.

We now define formally the main requirement for reconciliation algorithms, i.e., the fact that the reconciled policy must not violate any of the component policies.

Definition 14 (correct reconciliation). Let P be a set of policies and p a policy. p is a *correct reconciliation* of P iff p refines every $p' \in P$. □

The following lemma follows from the definitions of correct reconciliation, refinement and equivalence.

Lemma 15. *Let $P = \{p_1, \ldots, p_m\}$ and $P' = \{p'_1, \ldots, p'_m\}$ be sets of policies such that p'_k is equivalent to p_k for $k = 1 \ldots m$. If a policy p is a correct reconciliation of P, it is also a correct reconciliation of P'.* □

Lemma 16. *Let P be a set of policies and p a correct reconciliation of P. If a policy p' is equivalent to p, then p' is also a correct reconciliation of P.* □

Proof. The lemma, too, follows directly from the definitions of correct reconciliation, refinement and equivalence.

Probably the most intuitive way of reconciling policies is to decorrelate them first and then take a cross product of the component policies. The number of entries in the reconciled policy is equal to the product of the number of entries in the component policies. The selectors in the reconciled policy are computed as intersections of the component selectors and the actions as intersections of the component actions.

Definition 17 (crossproduct set). Let $P = \{p_1, \ldots p_m\}$ be a set of policies where

$$p_k = \langle e_1^k, \ldots, e_{n_k}^k \rangle = \langle\langle s_1^k, a_1^k\rangle, \ldots, \langle s_{n_k}^k, a_{n_k}^k\rangle\rangle$$

and n_k is the length of p_k for $k = 1 \ldots m$. Furthermore, denote

$$\begin{aligned} s_{(i_1,i_2,\ldots,i_m)} &= \cap_{k=1}^m s_{i_k}^k, \\ a_{(i_1,i_2,\ldots,i_m)} &= \cap_{k=1}^m a_{i_k}^k, \text{ and} \\ e_{(i_1,i_2,\ldots,i_m)} &= \langle s_{(i_1,i_2,\ldots,i_m)},\ a_{(i_1,i_2,\ldots,i_m)}\rangle. \end{aligned}$$

We call the set of policy entries $E = \{e_{(i_1,i_2,\ldots,i_m)} \mid 1 \le i_k \le n_k \text{ for } k = 1 \ldots m\}$ the *crossproduct set* of P. □

Definition 18 (policy crossproduct). Let P be a set of policies. Any policy that is obtained by ordering the crossproduct set of P linearly is a *crossproduct* of P. □

Lemma 19. *Let P be a set of policies and E its crossproduct set. Any linear ordering of E is a policy.* □

Proof. Let P be a set of decorrelated policies and E its crossproduct set. Denote the elements of P and E be as in definition 17. We observe that the following reduction holds:

$$\begin{aligned} & \cup \{s_{(i_1,i_2,\ldots,i_m)} \mid 1 \le i_k \le n_k \text{ for } k = 1 \ldots m\} \\ = \ & \cup_{i_1=1}^{n_1} \cup_{i_2=1}^{n_2} \ldots \cup_{i_{m-1}=1}^{n_{m-1}} \cup_{i_m=1}^{n_m} (\cap_{k=1}^m s_{i_k}^k) \\ = \ & \cup_{i_1=1}^{n_1} \cup_{i_2=1}^{n_2} \ldots \cup_{i_{m-1}=1}^{n_{m-1}} \cup_{i_m=1}^{n_m} ((\cap_{k=1}^{m-1} s_{i_k}^k) \cap s_{i_m}^m) \\ = \ & \cup_{i_1=1}^{n_1} \cup_{i_2=1}^{n_2} \ldots \cup_{i_{m-1}=1}^{n_{m-1}} ((\cap_{k=1}^{m-1} s_{i_k}^k) \cap (\cup_{i_m=1}^{n_m} s_{i_m}^m)) \\ = \ & \cup_{i_1=1}^{n_1} \cup_{i_2=1}^{n_2} \ldots \cup_{i_{m-1}=1}^{n_{m-1}} ((\cap_{k=1}^{m-1} s_{i_k}^k) \cap \mathbf{H}) \\ = \ & \cup_{i_1=1}^{n_1} \cup_{i_2=1}^{n_2} \ldots \cup_{i_{m-1}=1}^{n_{m-1}} (\cap_{k=1}^{m-1} s_{i_k}^k) \ = \ \ldots \ = \ \mathbf{H} \end{aligned}$$

The equivalence with $\mathbf{H}$ results from repeating the same reduction m times.

Lemma 20. *Let P be a set of decorrelated policies and E its crossproduct set. Any linear ordering of E is a decorrelated policy.* □

Proof. Let P be a set of decorrelated policies and E its crossproduct set. Denote the elements of P and E as in definition 17. By lemma 19, a linearization of E is a policy. We need to show that a linearization of E is decorrelated. Assume the contrary, i.e., that for some $e_{(i_1,i_2,\ldots,i_m)} \neq e_{(j_1,j_2,\ldots,j_m)} \in E$, there exist an $h \in \mathbf{H}$ such that $h \in s_{(i_1,i_2,\ldots,i_m)}$ and $h \in s_{(j_1,j_2,\ldots,j_m)}$. From the definition of $s_{(i_1,i_2,\ldots,i_m)}$ it follows that $h \in s^k_{i_k}$ and $h \in s^k_{j_k}$ for all $k = 1 \ldots m$. Since all p_k are decorrelated, it must be the case that $i_k = j_k$ for all $k = 1 \ldots m$. Thus, $e_{(i_1,i_2,\ldots,i_m)} = e_{(j_1,j_2,\ldots,j_m)}$. This contradicts with our assumption, which proves the claim.

Theorem 21. *Let P be a set of decorrelated policies. Every crossproduct of P is a correct reconciliation of P.* □

Proof. Let P be a set of decorrelated policies and E its crossproduct set. Denote the elements of P and E be as in definition 17. Let p be a sequence obtained by ordering linearly the elements of E. From lemma 20, we know that p is a decorrelated policy. It remains to show that p refines all policies in P. Consider an arbitrary $p_l \in P$ and $h \in \mathbf{H}$. There is a unique policy entry $e_{(i_1,i_2,\ldots,i_m)} = \langle s_{(i_1,i_2,\ldots,i_m)}, a_{(i_1,i_2,\ldots,i_m)} \rangle$ in p that matches h. $s_{(i_1,i_2,\ldots,i_m)} = \cap_{k=1}^m s^k_{i_k} \subseteq s^l_{i_l}$ where i_l is the index of the unique policy entry in p_l that matches h. The allowed actions for h in p_l are $a^l_{i_l}$. The allowed actions for h in p are $a_{(i_1,i_2,\ldots,i_m)} = \cap_{k=1}^m a^k_{i_k} \subseteq a^l_{i_l}$. This shows that, for an arbitrary h, $\text{Allowed}(p,h) \subseteq \text{Allowed}(p_l,h)$. Thus, p refines p_l, which concludes the proof.

Theorem 22. *The following algorithm computes a correct reconciliation of a set of policies:*

1. *Decorrelate each input policy by computing* $\text{Decor}(p)$.
2. *Compute a crossproduct of the non-repetitive, decorrelated policies.*
3. *Remove all policy entries that have empty selectors from the crossproduct.*

□

Proof. By theorem 21, step 2 computes a correct reconciliation. By lemmas 15 and 16, we can replace policies with equivalent ones before and after the reconciliation step. By theorem 13 and lemma 10, steps 1 and 3 replace policies with equivalent ones. Thus, the algorithm produces a correct reconciliation.

Note that step 1, i.e., computing the decorrelated policy is non-trivial because it involves set intersections and minus operations on sets. The resulting selectors may produce selectors that are not simple ranges even if all the selectors in the input were.

It is not surprising that the decorrelated policies can be reconciled by taking the cross product of their entries. What is more surprising is that the decorrelation step is, in fact, unnecessary. Instead, it suffices to retain some of the order from the component policies. The advantage of this algorithm is that that intersection is the only set operation required.

Definition 23 (crossproduct lattice order). Let P be a set of policies and E its crossproduct set. Denote the elements of P and E as in definition 17. The *crossproduct lattice order* on E is the partial order $\preceq$ on E such that $e_{(i_1,i_2,\ldots,i_m)} \preceq e_{(j_1,j_2,\ldots,j_m)}$ iff $i_k \leq j_k$ for all $k = 1 \ldots m$. □

Definition 24 (ordered crossproduct). Let P be a set of policies. Any policy that is obtained by extending the crossproduct lattice order on E to a linear order is an *ordered crossproduct* of P. □

An ordered crossproduct is clearly a crossproduct, only with more restrictions on the order of items. Thus, lemma 19 is sufficient to show that an ordered crossproduct is a policy.

It would be possible to further relax the requirements on the order policy entries. The order of two entries is unimportant, for example, if the selectors do not intersect or if the actions are equal. The above definition is, however, sufficient to prove the correctness of the algorithms presented in this paper. Further optimisations may be possible with a more relaxed definition of the ordering.

Lemma 25. *Let P be a set of policies, E its crossproduct set, and p an ordered crossproduct of P. Denote the elements of P and E as in definition 17. Let $h \in \mathbf{H}$. If $e_{(j_1,j_2,\ldots,j_m)}$ catches h in p, then $e^k_{j_k}$ catches h in p_k for all $k = 1 \ldots m$.* □

Proof. Let P, E and p be as in the theorem, $h \in \mathbf{H}$, and $e_{(j_1,j_2,\ldots,j_m)}$ the policy entry that catches h in p. Denote by $\preceq$ the crossproduct lattice order on E. $h \in s_{(j_1,j_2,\ldots,j_m)} = \cap_{l=1}^{m} a^l_{j_l} \subseteq a^k_{j_k}$ for $k = 1 \ldots m$. Thus, $e^k_{j_k} = \langle s^k_{j_k}, a^k_{j_k} \rangle$ matches h in p_k for $k = 1 \ldots m$.

We need to show that the j_kth entry is the first entry that matches h in p_k for $k = 1 \ldots m$. Assume the contrary, i.e., for some particular $1 \leq l \leq m$, the first entry in p_l that matches h is $e^l_{i_l}$ and $i_l < j_l$. Let $i_k = j_k$ for $k \neq l$. Now, the condition of definition 23 is fulfilled. Therefore, $e_{(i_1,i_2,\ldots,i_m)} \preceq e_{(j_1,j_2,\ldots,j_m)}$. Moreover, $h \in s^k_{i_k}$ for $k = 1 \ldots m$, which implies $h \in \cap_{l=1}^{m} s^l_{i_l} = s_{(i_1,i_2,\ldots,i_m)}$, i.e., that $s_{(i_1,i_2,\ldots,i_m)}$ matches h. But if that is the case, then $s_{(j_1,j_2,\ldots,j_m)}$ is not the first matching entry for h in p, which contradicts with the fact that $e_{(j_1,j_2,\ldots,j_m)}$ catches h. Since our assumption lead to this contradiction, it must be false and the j_kth entry must be the first one that matches h in each p_k for $k = 1 \ldots m$. This implies the lemma.

Theorem 26. *Let P be a set of policies and p an ordered crossproduct of P. p is a correct reconciliation of P.* □

Proof. Let P be a set of policies, E its crossproduct set, and p an ordered crossproduct of P. Denote the elements of P and E as in definition 17.

We need to show that p refines $p_k \in P$ for $k = 1 \ldots m$. Consider arbitrary $1 \leq k \leq m$ and $h \in \mathbf{H}$. By lemma 8, there is some $e_{(j_1,j_2,\ldots,j_m)}$ that catches h in p. By lemma 25, $e^k_{j_k}$ catches h in p_k. $a_{(j_1,j_2,\ldots,j_m)} = \cap_{l=1}^{m} a^l_{j_l} \subseteq a^k_{j_k}$, i.e.,

Allowed$(p, h) \subseteq$ Allowed(p_k, h). Since this is true for an arbitrary k and h, p refines p_k for all $k = 1 \ldots m$, which implies that p is a correct reconciliation of P.

A policy set may have correct reconciliations that are not an ordered crossproducts. They may be either more restrictive policies (e.g., a trivial policy that maps all headers to an empty action set), or equivalent policies with different order or granularity of entries. The following theorem proves that the ordered crossproduct is, in this sense, the most general reconciliation.

Theorem 27. *Let P be a set of policies and p an ordered crossproduct of P. Every correct reconciliation of P refines p.*

Proof. Let P be a set of policies, E its crossproduct set, and p an ordered crossproduct of P. Denote the elements of P and E as in definition 17.

Consider any $h \in \mathbf{H}$. By construction of p, Allowed$(p, h) = \cap_{k=1}^{m} a_{i_k}^k$. This is equal to $\cap_{k=1}^{m}$Allowed(p_k, h), by lemma 25, or, equivalently, $\cap_{p' \in P}$Allowed(p', h). Now suppose some policy q is a correct reconciliation of P, that is, for all $p' \in P$, Allowed$(q, h) \subseteq$ Allowed(p', h). Therefore, Allowed$(q, h) \subseteq \cap_{p' \in P}$Allowed$(p', h) =$ Allowed(p, h), as required.

Theorem 28. *The following algorithm computes a correct reconciliation of a set of policies:*

1. *Compute an ordered cross-product of the input policies.*
2. *Remove all policy entries that have empty selectors from the crossproduct.*

□

Proof. By theorem 26, step 2 computes a correct reconciliation. By lemma 16, we can replace the policy with an equivalent one after the reconciliation step. By lemma 10, step 2 replaces policies with equivalent ones. Thus, the algorithm produces a correct reconciliation.

7 Reconciliation algorithm

Theorem 22 gives an intuitive algorithm for reconciling a set of IPsec policies. The policy entries are decorrelated before the reconciliation step. The problem with this algorithm is that the selectors in most IPsec policies and implementations are simple multi-dimensional ranges (e.g. address ranges or port ranges or both). Decorrelation, however, requires one to compute set union and minus operations. (Figure 8 has pseudocode for decorrelation.) The decorrelated selectors are no longer simple multi-dimensional ranges but complex areas in the selector space. The reconciled policy will also contain such complex selectors. Since IPsec implementations do not accept policies with such selectors, one would have to divide each entry into simple subranges and create a separate policy entry for

```
Reconcile(in p1, in p2, out p)
  OrderedCrossproduct(p1, p2, p);
  RemoveEmpty(p);

OrderedCrossproduct(in p1, in p2, out p)
  p = ⟨⟩;
  for (e1 ∈ p1)
    for (e2 ∈ p2)
      p.append(⟨ e1.selector ∩ e2.selector,
                 e1.action ∩ e2.action ⟩);

RemoveEmpty(in/out p)
  for (i = e1.length downto 1)
    if (e1.entry(i).selector == ∅)
      e1.delete(i);
```

Fig. 1. Pseudocode for reconciling two policies

each. This may increase substantially the number of policy entries in the final reconciled policy.

The main result of this paper, theorem 28 shows that it is possible to avoid the decorrelation step. Moreover, intersection is the only set operations that is required to compute the reconciled policy. Figure 7 provides pseudocode for a reconciliation algorithm that is based on theorem 28. For readability, the pseudocode takes as its input only two policies but it can be easily extended to an arbitrary number of component policies.

Since the intersection of two simple ranges is a simple range, the policy crossproduct will have only simple multi-dimensional ranges as selectors. This means that the resulting policy will have at most as many lines as is the product of the number of entries in the component policies, and that the reconciled policy is directly usable in most IPsec implementations. The correctness of this algorithm is not obvious, which is why we needed to develop the theory in the previous section.

A key to understanding the pseudocode is that the the nested loops in the function `OrderedCrossproduct` output the entries of the crossproduct in a lexicographic order, which clearly is a linearization of the crossproduct lattice order (def. def:orderedcrossproduct).

The final output `Reconcile` algorithm may still contain more than one allowed action. The preferred action should be chosen based on some priority scheme, as discussed in section 4. After that, the policy may be further processed and its implementability may be checked with the algorithms presented in the next section.

8 Shadowing and collecting

The result of the policy reconciliation in the previous section may still contain redundant entries, that is, ones that can safely be removed without changing the behaviour of the policy. Removing redundant entries reduces the size of the policy and, usually, improves performance.

We discuss two specific types of redundancy, and how to eliminate them. We say that a policy entry is shadowed if its selector is covered by the selectors before it, and that it is collected if the later entries in the same policy map headers caught it to the same allowed actions. In either case, the entry can be removed. Again, the reader may want to take first a look at the example in the appendix.

Definition 29 (shadowing). Let $p = \langle\langle s_1, a_1\rangle, \ldots \langle s_n, a_n\rangle\rangle$ be a policy. $\langle s_i, a_i\rangle$ *is shadowed* iff $s_i \subseteq \cup_{j=1}^{i-1} s_j$. □

Definition 30 (collecting). Let $p = \langle e_1, \ldots, e_n\rangle$ be a policy. $e_i = \langle s, a\rangle$ *is collected* iff for every $h \in \mathbf{H}$ that is caught by e_i, $\text{Allowed}(p, h) = \text{Allowed}(p', h)$ where p' is the sequence of policy entries $p' = \langle e_{i+1}, \ldots, e_n\rangle$. □

Lemma 31. *A policy entry is shadowed iff it does not catch any headers.* □

Proof. Let $p = \langle\langle s_1, a_1\rangle, \ldots \langle s_n, a_n\rangle\rangle$ be a policy. Assume first that the policy entry $\langle s_i, a_i\rangle$ is shadowed, i.e., $s_i \subseteq \cup_{j=1}^{i-1} s_j$. Let $h \in \mathbf{H}$. If $\langle s_i, a_i\rangle$ matches h, then $h \in s_i \subseteq \cup_{j=1}^{i-1} s_j$. This implies $h \in s_j$, i.e., $\langle s_j, a_j\rangle$ matches h for some $j = 1 \ldots (i-1)$. Thus, the ith entry cannot not catch h.

On the other hand, assume that the policy entry $\langle s_i, a_i\rangle$ does not catch any headers. If $s_i = \emptyset$, the entry is shadowed by definition. Otherwise, consider an arbitrary $h \in s_i$. Then, $\langle s_i, a_i\rangle$ matches h. To prevent it from catching h, some earlier entry must match h, i.e., $h \in s_j$ for some $j = 1 \ldots (i-1)$. Thus, $h \in s_i$ implies $h \in \cup_{j=1}^{i-1} s_j$, which means that the ith entry is shadowed.

Lemma 32. *A policy is implementable iff every policy entry $\langle s, a\rangle$ for which $a = \emptyset$ is shadowed.* □

Proof. By the definition of implementable, a policy p is implementable iff $\text{Allowed}(p, h) \neq \emptyset$ for all $h \in \mathbf{H}$. By the definition of allowed actions, this is the case iff for each entry $\langle s, a\rangle$ in p, either $a \neq \emptyset$ or the entry does not catch any headers. In the latter case, by lemma 31, the entry is shadowed.

The following lemma follows directly from lemmas 10 and 31.

Theorem 33. *Removing one or more shadowed policy entries from a policy produces an equivalent policy.* □

While shadowed entries can be removed all at once, collected entries must be deleted one by one. This is because deleting one collected entry may cause another to be not collected.

Theorem 34. *If a policy entry is collected, removing it from the policy produces an equivalent policy.* □

Proof. Let $p = \langle e_1, \ldots, e_n \rangle$ be a policy and let e_i be collected.
Let $p' = \langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$ be the same policy but with e_i removed. Denote $e_l = \langle s_l, a_l \rangle$ for $l = 1 \ldots n$.

We show both that p' is a policy and that p and p' are equivalent. Consider an $h \in \mathbf{H}$. By lemma 8, it is caught by some jth policy entry in p. If $j \neq i$, then by lemma 9, $\text{Allowed}(p, h) = \text{Allowed}(p', h)$. On the other hand, if $j = i$, then none of the entries before j matches h. The first matching entry in the remaining part of the policy, i.e., $p' = \langle e_{i+1}, \ldots, e_n \rangle$ catches h. By the definition of collecting, the allowed actions at this catching entry are the same in p' as in p. This shows that, for an arbitrary h, $\text{Allowed}(p, h) = \text{Allowed}(p', h)$, which implies the equivalence. Since $\text{Allowed}(p', h)$ is defined for all h, p' must also be a policy.

Theorem 35. *The following algorithm computes a correct reconciliation of a set of policies:*

1. *Compute an ordered cross-product of the input policies.*
2. *Remove all policy entries that have empty selectors from the crossproduct.*
3. *Remove all shadowed policy entries from the ordered crossproduct.*
4. *Remove collected policy entries, one by one, until none exist.*

□

Proof. By theorem 26, step 2 computes a correct reconciliation. By lemma 16, we can replace the policy with an equivalent one after the reconciliation step. By lemma 10 and theorems 33 and 34, steps and 2-4 replace policies with equivalent ones. Thus, the algorithm produces a correct reconciliation.

From the definitions it is easy to see that any policy entry with an empty selector is shadowed and any shadowed entry is collected. It is, however, more efficient to remove the empty and shadowed entries first because the algorithm for removing collected entries is the slowest. The following theorem shows that after removal of all shadowed and collected entries, a policy is completely free of redundancy: any further removal of entries would not preserve equivalence.

Theorem 36. *Let $p = \langle e_1, ..., e_n \rangle$ be a policy, and let p' be the sequence of policy entries obtained from p by removing the ith entry e_i, for some $i = 1...n$. If p and p' are equivalent then e_i is shadowed or collected.*

Proof. We prove that e_i is collected. Let p'' be the sequence of policy entries $\langle e_{i+1}, ..., e_n \rangle$. Consider an any $h \in \mathbf{H}$. The equivalence of p and p' implies that, if $e_i = \langle s_i, a_i \rangle$ catches h in p, there must be some $e_j = \langle s_j, a_j \rangle$ with $j > i$ and $a_i = a_j$ that catches h in p'. Therefore, $\text{Allowed}(p, h) = \text{Allowed}(p'', h)$, and hence e_i is collected.

9 Algorithm improvements

The algorithms in this section are computationally more expensive than the crossproduct in the previous section because they require one to compute a decorrelation of the reconciled policy. This does not, however, create any new entries to the reconciled policy or increase the run-time overhead when the policy is used in an IPsec implementation. Instead, the computation, including decorrelation, is required only to find out which entries can be removed from the un-decorrelated reconciled policy. This computation is all done once at the time of policy configuration and not when individual IP packets are processed.

It is important to run the optimization algorithm *after* selecting the unique allowed action for each policy entry. That way, more policy entries will be removed. The invocation of `SelectUniqueActions` in the pseudocode represents this step.

The algorithms for decorrelation and the removal of shadowed and collected entries require set operations on selectors (union, intersection, difference) as well as subset checking. These operations are expensive if selectors are naively implemented as sets. Instead, the selectors could be represented as propositional formulas and the set operations as boolean operations (disjunction, conjunction, implication). These could then be efficiently implemented using (ordered) binary decision diagrams (BDDs), as is discussed in [2, 3]. A similar approach is taken in [6], where decision trees are used to identify and remove shadowing and collected entries in firewall policies.

In is important to note, however, that the algorithms described in this paper do not need to be implemented very efficiently because they are executed during policy configuration and not when processing each IP packet. The selectors in the final policy are still simple ranges if the selectors in the input policies are. Only the intermediate computation requires handling of complex sets of selectors.

10 Conclusion

In this paper, we presented an algorithm for reconciling two or more IPsec policies. The algorithm produces short and efficient policies without decorrelating the component policies first. Since the correctness of the algorithm is not obvious, we gave a formal definition of correct reconciliation and proved that the algorithm meets it. We also showed how to remove redundant entries from the policy and proved that it remains a correct reconciliation.

The results can be used to implement composition of multiple IPsec and firewall policies. We expect it to be much easier for the administrators and users to specify independent component policies, which are automatically compiled into one policy, than to manually configure one monolithic policy for each device.

```
Reconcile2(in p1, in p2, out p, out conflicts)
  OrderedCrossproduct(p1, p2, p);
  RemoveEmpty(p);
  SelectUniqueActions(p); // Not defined here
  Decorrelate(p, d);
  CheckConflicts(p, conflicts);
  RemoveShadowed(p, d);
  RemoveCollected(p, d);

Decorrelate(in p, out d)
  d = ⟨⟩;
  union = ∅;
  for (i = 1 to p.length)
    e = p.entry(i);
    d.append(⟨ e.selector \ u, e.action ⟩);
    union = union ∪ e.selector;

CheckConflicts(in d, out conflicts)
  conflicts = ∅;
  for (e ∈ d)
    if (e.action == ∅)
      conflicts = conflicts ∪ e.selector;

RemoveShadowed(in/out p, in/out d)
  for (i = p.length downto 1)
    if (d.entry(i).selector == ∅)
      p.delete(i);
      d.delete(i);

RemoveCollected(in/out p, in/out d)
  aset = ∅;
  for (e ∈ p)
    aset = aset ∪ { e.action };
  for (a ∈ aset)
    RemoveCollectedForAction(p, d, a)

RemoveCollectedForAction(in/out p, in/out d, in a)
  for (i = p.length downto 1)
    e = p.entry(i);
    if (e.action == a)
      if (d.entry(i).selector ⊆ c)
        p.delete(i);
        d.delete(i);
      else
        collect = collect ∪ e.selector;
    else
      collect = collect ∩ e.selector;
```

Fig. 2. Removing shadowed and collected entries

References

1. Kenneth J. Arrow. *Social Choice and Individual Values*. Yale University Press, 1970.
2. Joshua D. Guttman and Amy L. Herzog. Rigorous automated network security management. *International Journal of Information Security*, 4(1–2), 2005.
3. Hazem H. Hamed, Ehab S. Al-Shaer, and Will Marrero. Modeling and verification of IPSec and VPN security policies. In *13th IEEE International Conference on Network Protocols (ICNP 2005)*, pages 259–278, 2005.
4. David B. Johnson, Charles Perkins, and Jari Arkko. Mobility support in IPv6. RFC 3775, IETF Mobile IP Working Group, June 2004.
5. Stephen Kent and Karen Seo. Security architecture for the Internet Protocol. RFC 4301, IETF, December 2005.
6. Alex X. Liu and Mohamed G. Gouda. Complete redundancy detection in firewalls. In *Proceedings of 19th Annual IFIP Conference on Data and Applications Security, LNCS 3654*, pages 196–209, Storrs, CT USA, August 2005. Springer.

A Policy reconciliation example

The following example shows two component policies A and B, their reconciliation (where the grey line will be deleted as empty), and the optimized policy after all shadowed and collected entries have been removed.

Policy A: general firewall

Policy Entry	**Selector**					**Allowed actions**
	Local IP	Remote IP	Local port	Remote port	Protocol	
A1	*	10.1.*.*	*	*	*	bypass, ESP transport, discard
A2	*	*	*	*	TCP	ESP transport, discard
A3	*	*	*	*	ICMP	bypass
A4	*	*	*	*	*	discard (default policy)

Policy B: Web server

Policy Entry	**Selector**					**Allowed actions**
	Local IP	Remote IP	Local port	Remote port	Protocol	
B1	*	*	80	*	TCP	bypass, ESP transport
B2	*	*	*	*	*	discard (default policy)

Policy C: Reconciliation of A and B

Policy Entry	Selector					Allowed actions	
	Local IP	Remote IP	Local port	Remote port	Protocol		
C11	*	10.1.*.*	80	*	TCP	bypass, ESP transport	
C12	*	10.1.*.*	*	*	*	discard	collected
C21	*	*	80	*	TCP	ESP transport	
C22	*	*	*	*	TCP	discard	collected
C31	*	*	80	*	–		empty
C32	*	*	*	*	ICMP	discard	collected
C41	*	*	80	*	TCP	–	shadowed
C42	*	*	*	*	*	discard	

Policy D: Shadowed and collected entries removed

Policy Entry	Selector					Allowed actions
	Local IP	Remote IP	Local port	Remote port	Protocol	
D11	*	10.1.*.*	80	*	*	bypass, ESP transport
D21	*	*	80	*	TCP	ESP transport
D42	*	*	*	*	*	discard