Securing the interactions between X clients.

Prem Uppuluri
Department of Information Technology
Radford University
Radford, VA, U.S.A

Alok Tongaonkar Dept. of
Computer science Stony
Brook University Stony
Brook, NY, U.S.A

Vivek Diwakara*
Microsoft Corporation
Redmond, WA, U.S.A

Vikas Rajegowda*
Acme Packet
Burlington, MA, U.S.A

Abstract

The security of Xwindows is usually divided into authentication/authorization of connections, and authorization of Xclient interactions [5]. The first issue has been well-addressed in research through mechanisms such as xhost and xauth[1]. In this paper we discuss the approaches to the last issue: one of authorization. We present a taxonomy of different approaches and discuss the effectiveness of a light-weight mechanism which we proposed.

1 Introduction

Several security issues with Xwindows have been discussed in previous works [5, 11]. These include:

- Issues related to authentication and authorization of Xclients to connect to an Xserver,
- Authorization of Xclients after they connect to an Xserver. This deals with the actions that Xclients can perform once they are authenticated [5, 11].

*Work done as students at the School of Computing and Engineering, University of Missouri, Kansas City, MO 64110 from 2004-2005.

[1]www.x.org/archive/X11R6.8.1/doc/xauth.1.html

The security issues as mentioned in [11, 5] are that once Xclients have been authenticated they can perform a number of actions that circumvent the access control mechanisms provided by operating systems. These include: reading, modifying and/or controlling information displayed by other Xclients as long as they are connected to the same Xserver [5]. Figure 1 illustrates some such threats as discussed in [5].

These threats motivate the need for mechanisms to manage the interactions involving information and control flow between Xclients. In this paper we:

- Present a taxonomy of the different approaches that address these issues.
- Present a formal treatment of a light-weight specification based approach which we developed.
- A discussion of the effectiveness of such a light-weight mechanism.

We presented preliminary results on how to use this approach to prevent information-leaks in Xwindows [14]. This paper extends and differs from this previous work in the following ways:

- We present a more detailed discussion of related work.
- We present more detailed explanations for the formal aspects of our preliminary work.

Manuscript received September 5, 2009
Manuscript revised September 20, 2009

Attack category	**Example attack(s)**
Confidentiality	Snoop on information displayed by Xclients using screen capture or *copy* operation
Integrity	*Paste* data into an open Xclient window being run by another user.
Availability	Repeatedly grab the mouse or close other Xclients.

Figure 1: Example of how attacks can exploit Xclient interactions based on [5]

2 Taxonomy of approaches to Xclient security

An important approach was the Compartmented Mode Workstation (CMW) model[3]. The proposed model resulted in several mechanisms to develop secure Xwindows [11]. According to [3] the model proposed “security requirements” [3] for multi-level security. [3] also mentions that the CMW proposed concepts such as “labeling”[3] and a “trusted window management system”[3]. A more thorough analysis of the implementation of the CMW model is discussed in [11].

The Xsecurity extension[2] [12] proposed extending Xfree86 windowing system to make it more secure. Xclients are ”divided into two groups: trusted and un-trusted”[12]. The trusted Xclients can interact with each other without any monitoring while un-trusted Xclients are “limited in what they do” [12].
The Eros Window System (EWS)[11] builds a windowing server from scratch focusing on security [11]. EWS is built on the EROS (*Extremely Reliable Operating System*)[10] operating system which is based on the implementing access control using *capabilities*[6]. Coloring schemes are used to label trusted windows to inform users of the capabilities of each client. The researchers/developers of EWS point out in [11] that by using capabilities they are able to prevent a security flaw in the CMW model: *one client’s vulnerability cannot be exploited to effect the others* [11], thus preventing attacks within a certain trust level. While EWS is unlike CMW in that it doesn’t support mandatory access control, the authors note that this is a small extension [11].
SELinux[5] is a research effort that was inspired by the CMW model. It is currently distributed as part of several standard linux distributions prominently with Red Hat Linux versions. Unlike EROS, SELinux is not an OS from ground up but rather builds on the standard Linux distribution. Access control is enforced using a form of domain type enforcement [7]. Currently SELinux is available as extensions to several Linux distributions including Fedora Core(TM)[3] and Ubuntu(TM)[4]. In [5], the authors discuss the key security requirements and design issues related to making Xserver secure on an SELinux installation. Some salient features of their design include:

- There is only a single Xserver with which all the clients communicate. However, the clients can be in different domains and information flow between them is restricted by SELinux policies.
- It provides comprehensive security. Specifically, policies can be specified to mediate information flow both at the kernel level as well as at the Xserver level.

As alternative to these approaches are light-weight systems that are either wrappers (e.g., sandboxes) built around existing installations or seek to fix the issues with existing installations. We survey two such approaches: the Xbox sandboxing system [1, 2] and retrofitting current Xserver code [4].
XBox/MAPBox[1, 2] was one of the first such systems. It is a sandbox that restricts each client to only execute Xprotocol messages on the resources it created (with a few exceptions). The clients are not allowed to communicate with other clients (with exception to child clients) and hence operations such as copy and paste may not be allowed. This system was built using the Janus [15] sandbox specification language that allows for filtering based on individual

[2]www.xfree86.org/current/security.pdf

[3]http://docs.fedoraproject.org/selinux-faq/

[4]https://wiki.ubuntu.com/SELinux

events such as Xprotocol messages.

A recent approach has focused on "retrofitting" [4] the existing X server code with techniques to enforce authorization [4]. They do this using two mechanisms which they call **AID** and **ARM**. AID is used to search for "security-sensitive operations"[4] by looking at application traces. Whenever such an operation is found, the code is instrumented by ARM with call backs to an authorization module before the operation is executed. The authorization module checks "the *subject*, the operation the subject wants to execute on an *object* and either allows or denies the operation"[4]. According to [4] it does automatically what SELinux did manually over a few years. This retrofitted Xserver was executed by them on top of SELinux [4] and hence we consider this a light weight system.

2.1 Categorizing different approaches

To achieve security, windowing systems need to satisfy certain requirements. In [5] the authors identify some of the requirements from the CMW model as needed for implementing secure windowing systems [5]. These include *labeling* which requires that users are informed about the trust level of each Xclient by the use of labels, *trusted path* between the user and the Xclient, *confidentiality*, *integrity*, and *application compatibility* which requires that existing clients should be able to execute unmodified on the secure server [5]. [5] also notes that all of these requirements must be incorporated with as few changes to the existing Xfree86 as possible.

In addition if the windowing systems are to be used in a multi-level security model (such as the motivation for CMW, SELinux and EROS) it is easy to see that the approach must support the following functionality:

- *Selective management of Xclients.* To manage information flow between Xclients, users should be able to apply different security models. For instance, to apply the simple property of the Bell-LaPadula model[8], the administrator must be able to: (a) group users and assign hierarchical trust levels to them, and (b) permit information flow to groups with higher trust levels and deny information flow to groups with lower trust levels. In terms of Xclients, this implies that the information displayed on Xclients being run by some users belonging to a group can be *read* by Xclients run by users in a group at a higher trust level but not by Xclients run by users at lower trust levels. This is similar to the policy discussed in [4].
- *Dynamically manage Xclient groups:* Consider an example of a transitive policy. Suppose a user wishes to enforce a simple confidentiality policy which states that information displayed by an Xclient X can be copied by Xclient Y, but not by Xclient Z. In addition, assume that the current trust relationships allow Z to copy the information displayed by Y. An information leak can occur, when the following interactions occur: Y reads from X, and then, Z reads from Y, thus effectively reading X's information. Such leaks must be prevented.

In addition to the above requirements the following categories can be used to distinguish these approaches:

- Heavy-weight vs. light-weight: We define heavy-weight systems as those that in some ways represent a significant change from the existing windowing systems in use. We consider EWS and SELinux are both heavy-weight approaches. EWS is part of a completely new operating system: EROS[10]. While EROS was developed for an academic environment, its successor the CapROS[5] is being developed as a commercial system. However to our knowledge, based on the current website of CapROS[5] it has been ported to run only on a few architectures (IA-32 and ARM-9) and has limited functionality. While SELinux unlike EWS builds on standard linux installations (and is infact being shipped with several Linux flavors), it represents a paradigm shift from the discretionary access control mechanisms that current OSes support. While there are efforts to develop GUI based policy edi-

[5]http://www.capros.org

[5],

tors such as SEedit[6] to make policy development easier for SELinux there are certain common functions that users perform on general purpose OSes such as porting new applications seems to be non-trivial as pointed out in a blog[7]. Even as these are subjective opinions, we believe that regular users currently used to simple discretionary based access control model need to learn about domains, types and other facets of SELinux in order to effectively deploy its security mechanisms and moreover, the learning curve to do so maybe large. Hence, we consider it to be a heavy-weight system.

- Comprehensive vs. Limited to windowing system: Some of the systems provide comprehensive security right from regular applications to windowing clients. For instance in EWS and SELinux the information flow policy can be enforced not only at the windowing level but also across every process. While XBox is specific to Xwindows, the authors note in [1] that it should be used in conjunction with a more comprehensive system such as their kernel level sand- box, MAPBox [1]. Xbox by itself is however a light-weight sandbox. Similarly, while we put [4]'s mechanism into Limited, we must note that the researchers executed this system on top of SELinux [4] making it comprehensive.

Figure 2 summarizes all the approaches. In summary, a light-weight system such as Xbox fails to provide multi-level security, while EWS and SELinux are either not easy to use or have not been widely deployed for general purpose environments. Our goal was to develop a light-weight system that addresses these issues.

3 Overview of our approach

The preliminary results of the mechanism we developed were presented at [14].

For ease of explaining we will refer to our mechanism as *Xfilter* in the rest of the paper. Xfilter builds on top of MapBox/Xbox[1] by seeking to add a multi-level security system and more expressive policies. Our approach is based on specifying and dynamically managing *trust relationships* between Xclients. A trust relationship between any two clients connected to the same server defines the policies that govern the interactions between them.

[14] presented an overview and description of the approach. In this section we present a summary of that work, in some places reproduced verbatim and in some places expanded for clarifying the approach. In our approach (based on Xbox) Xprotocol messages are intercepted and delivered to a policy enforcement engine (EE). Note that this is an extension from Xbox which simply intercepts Xprotocol messages and makes a decision on them. An EE maps to an Xclient and monitors its interaction with the Xserver.

The EE is very similar to the detection engines (DEs) from our work on intrusion detection[13], the cruicial difference being that they are tailored towards Xprotocol messages. An EE is composed of three parts:

- *Class specification.* A monitored client can have a different trust relationship with each of the other clients. For instance, if trust were categorized hierarchically (i.e., multi-level security), such as in a military establishment, the trust between a monitored client and another client depends on the sensitivity of both the clients. Class specifications, allow a user to specify these different classes of trust and assign specific trust relationship policies for each class.
- *Policy specification*: These specify the valid and invalid sequences of interactions between clients. For instance, a client can have a policy to prevent some clients from reading its data buffer. Policies are specified using a high level specification language. We used the Behavior modeling specification language [13] which we developed for intrusion detection [9]. BMSL was developed specifically to express security policies and hence was ideally suited for this work.

[6]http://seedit.sourceforge.net

[7]http://blog.stevecoinc.com/2008/08/selinux.html

	SELinux	**EWS (EROS window system**	**Retrofit [4]**	**Xbox/MapBOX**
Installation type	Heavy-weight	Heavy-weight Comprehensive	Light-weight	Light-weight
Applicability	Comprehensive		Limited	Limited (comprehensive when used with a kernel-level mechanism such as MAPBox[1])
Security aspects:				
Trusted path	Yes	Yes		No
Controlling Copy and Paste	Yes	Yes	Yes	Yes
Labeling	Yes	Yes	Yes	No
Support Multilevel security models	Yes	Yes (implements Capabilities)	Yes (Can be supported by developing specific authorization code)	Limited (Sandbox is fixed before client executes
Policy granularity	Not very granular.[7]. Policies are mostly based on a single event	NA	Can be extended with a high-level language.	No. Policies are simple sandbox rules based on Janus[15].

Figure 2: Taxonomy of different approaches to controlling information flow in Window systems

- *Reactions/Responses:* are triggered when a sequence of interactions match a specified policy. These reactions can be simple, such as disallowing the interactions or can be more complex.

3.1 Managing Trust relationships between Xclients

Our goal was to allow users to manage the interactions between Xclients with a high-level of granularity. As this mechanism will run on general purposes OSes that support discretionary access control, the user can either be a system administrator or a regular user. Specifically, the user must be able to (a) selectively apply different policies between different Xclients, and (b) the policies should be able to capture the interactions precisely. To achieve these goals we designed the framework such that it provided the following functionality.

- **Flexible grouping of Xclients to selectively apply policies** : An Xclient can be identified based on two types of characteristics:
  - *runtime specific*. At runtime, each Xclient has two specific identities: *effective user id* and a *client id* – which is a unique id that the client gets from the server. In addition, an Xclient maybe executing remotely or locally.
  - *static characteristics*. Every client has a *program name*, a *program owner* and a *program category* such as editors (can edit information they display) and browsers (which cannot edit displayed information).

Out of these *client id* is a runtime characteristic which is arbitrarily assigned by the Xserver and does not capture any property of the Xclient. Hence, it does not have any useful information when specifying trust relationships. The rest of the characteristics on the other hand, can be used to specify trust relationships.

- **Specifying expressive policies to manage the interactions between Xclients:** Every interaction is a sequence of messages and events between the Xclients and the Xserver. Policies to manage these interactions must capture the sequential nature of these messages. Moreover, messages (both requests and events) have arguments, such as client id, whose relationships across the messages must be captured for precise specification of the interaction. For instance, the *copy and paste* operation, described in [14], which allows one Xclient to copy certain information from another Xclient can only be captured precisely by considering the sequence of messages/requests involved. [14] includes more details on this.
- **Reactions to launch when managing interactions:** Managing interactions involves specify permissible or non-permissible interactions. When the interactions match the policies, then suitable reactions need to be launched. Reactions can be simple. For instance, if an interaction matches a non-permissible policy, a simple reaction would deny that interaction. Reactions can also be complex. For instance, assume that the policy permits copy and paste between Xclients X and Y and also between Y and Z, but *not* between X and Z. Consider the sequence of interactions in which Y copies information from X (which is permitted) and then Z copies the same information from Y (which is also permitted). These sequences bypass the intention of the policy. To prevent this, when Y copies from X, the policy could trigger a reaction which restricts Y's interactions with other Xclients to those of X's interactions. In our approach reactions can be developed using an general purpose programming language such as C/C++.

3.2 Grouping Xclients

We use a simple language based on the way groups are organized in Linux (which we will call Xclient trust specification language (Xspec) in the rest of the paper), to selectively group Xclients and associate the groups with policies. This is inspired by the way groups are created in UNIX operating systems. Each Xspec has three parts: in the first two parts Xclients are grouped based on their static and runtime characteristics respectively and in the third part, policies are selectively applied to the various groups.

Groups based on static characteristics are created using the following syntax (note similarity to UNIX):

$$\begin{aligned} clientType &\longrightarrow clientTypeName\,`\texttt{:}'\,clientSet \\ clientSet &\longrightarrow clientName[\,'\texttt{,}'\,]clientSet \\ clientName &\longrightarrow string\ id\,|\,'*'\,|\,\varrho \\ \\ clientTypeName &\longrightarrow string\ id \end{aligned}$$

Terminals are indicated within single quotes. The terminal '*' is a special name which can be used to specify any Xclient program.

In the first part, each group has a name followed by the Xclient program names separated by a ",". For instance, consider two groups (example first used in our previous paper [14]) based on the type of Xclients: `editors` which groups Xclients that modify the files they display, and `browsers` which groups Xclients that only display files and do not modify them. They are specified as:

```
editors:  XEmacs, gedit, Xfig
browsers:  acrobat, xdvi
```

To group based on runtime characteristics we use the following syntax:

$$\begin{aligned} user &\longrightarrow usrTypeName\,`\texttt{:}'\,usrGroups \\ usrGroups &\longrightarrow usrGroupName\,'\texttt{:}' \\ &\qquad IP\ Address\,`,'\ usrGroups \\ usrGroupName &\longrightarrow usrName\,'\texttt{,}'\,usrGroupName \\ usrName &\longrightarrow string\ id\,|\,`*'\,|\,\text{epsilon} \end{aligned}$$

Here are two examples (from our previous paper [14]) of such groups: `localUsers` (represents all the users on the local machine, i.e., IP address 127.0.0.1) and the `privilegedUsers` representing administrator defined users such as `root` on a local machine and and user `P` on a machine with IP address 192.168.0.0.

```
localUsers:  X, Y:127.0.0.1
privilegedUsers:  root:127.0.0.1, P:192.168.0.0
```

The semantics of a group is the set of Xclient processes, whose characteristics (runtime or static) match the groups characteristics. For instance the Xclient processes with program names `XEmacs` and `gedit` match the `editors` group, while Xclients which are being run by user X on the local machine, match `localUsers`. Special characteristic "*" can also be used in groups and it matches any Xclient. Each Xclient can thus be represented as a tuple: `(static group, runtime group)`. For instance, consider a tuple: `(browsers, localUsers)`. This tuple represents several Xclients including: `{(acrobat, Y:127.0.0.1)`.

Groups are associated with policies Trust relationships are specified as rules which associate a set of policies with a *source* Xclient characteristic C and a *destination* set of Xclient characteristics C. Informally, it specifies the policies that govern the interaction of Xclient processes that match the specified destination characteristics C, with the Xclient which matches the source characteristic C . This definition allows us to apply different policies to the same set of Xclients, depending on which Xclient initiated the interactions. An example of such an association (first shown in [14] is:

```
copy:  (browsers, privilegedUsers) ::
   (editors, localUsers), (editors,
            privilegedUsers)
```

Here, `copy` is the name of the poly specification that defines the policies that govern copying buffers from Xclients which match the source characteristics defined by `(browsers, privilegedUsers)` to Xclients which match one of the characteristics in the set of destination characteristics `{(editors, localUsers), (editors, privilegedUsers)}`. This policy has been described in [14].

In addition to the above rules which explicitly associate policies with specific Xclient interactions, every Xspec specification also has a default rule, which associates all the interactions that are not part of the above associations with specific user defined policies. This rule is specified using the keyword `default`.

4 Examples of specifying trust relationships

In [14] we presented a confidentiality policy using our framework. Here, we present a simple integrity policy which seeks to preserve the integrity of information being edited using an editor such as `XEmacs` by privileged users `X` and `Y`. The requirements in terms of Xclients are that, no client other than those run by the two users locally on the machine running the Xserver can change the information. The complete specification for this policy is shown in Figure 3 where the event `ConvertSelection` is assumed to denote the event that occurs during a paste operation.

4.1 Implementation

As [14] presents more detailed implementation results of our preliminary prototype, in this paper we focus only on how our implementation extends Xbox [1]. Specifically, the Xbox source code in C programming language has been instrumented as follows:

- Every time an Xprotocol message is intercepted it is redirected to an enforcement engine using a method call: `deliverEvent(...)`. This was inspired (and is similar) to our work on intrusion detection systems [9].
- The enforcement engines (currently need to be written manually) keep track of the sequence of Xprotocol messages and apply the appropriate response when necessary.

In our prototype Xspec and EE compilers were not implemented and required manual integration.

```
/* Specifying types of Xclient programs */
    editors:  XEmacs, gedit, xfig, kedit  /* editors = Xclients which can modify information */
    allXclients:   *   /* allXclients is the group of all Xclients */
    allUsers:   *  /* all the users, remote or local */
    privilegedUsers:  X,Y:127.0.0.1
/* Specifying trust relationship rules */
    paste:  (editors, privilegedUsers) ::  (allXclients, privilegedUsers)
    denyPaste:  (editors, privilegedUsers) ::  (allXclients, allUsers)
/* BMSL specification */
    /* paste specification contains one policy. ConvertSelection is an event that occurs during the paste operation.  */
    paste {
      (any())* · (ConvertSelection(clientID)) → { allow() ; }
    }
    /* denyPaste specification is the same as paste, except it denies all copies */
    denyPaste {
      (any())* · (ConvertSelection(clientID)) → { deny() ; }
    }
/* Default section of the specification, denies copies to all other interactions */
    default:  denyPaste
```

Figure 3: Example of an Integrity Policy Specification

- We extended Xbox to handle clients connected via SSH tunnel.
- Finally the effective user id of a user is extracted using the `getsockopt` system call on the socket created by the clients. We are therefore using the same user-id given by the operating system.

4.2 Experiences and Discussion

Based on the performance results and the effectiveness results from our preliminary work in [14], we can summarize that the approach effectively manages information flow between Xclients through the Xserver at a very low-level of granularity. However, such a light-weight approach does not still address certain issues. Specifically,

- Complete mediation is not possible. Xfilter assumes that the clients do not communicate with themselves directly. Such an assumption could be flawed. Furthermore, as Xfilter (and Xbox [1] on which it is based) runs as a user-level process, many attacks can easily launch a DOS attack on it. One solution for this is to build the filter as a kernel level monitor. As the clients connect to the server using sockets, the kernel level module could intercept, the socket system calls (e.g.., the `socketcall` in Linux). However, this is not trivial: as pointed out in [5] there is a need to keep Xserver structure separate from the kernel.
- Trusted path is not possible. Once again, we believe a user-level filter cannot simply achieve trusted path without kernel support or without modifications to the Xserver.
- Comprehensive security: for a user-level extension such as Xfilter to provide comprehensive security there needs to be a kernel level security mechanism such as an intrusion detection system. As [1] points out, even Xbox requires to be run in association with the MapBOX software. In addition, without kernel support for mandatory access control of some form, it is not possible to support multi-level security.

5 Conclusion

In this paper we reviewed the current state of the art in windowing system security. As our paper shows, while there are comprehensive heavy-weight solutions, they are much harder to use or are not widely used. Meanwhile achieving a light-weight solution is possible, but these solutions may not address all the security relevant issues. Hence, we conclude that currently a heavy-weight system that provides security and flexibility such as SELinux maybe best suited for monitoring.

References

[1] M. Raje A. Acharya. Mapbox using parameterized behavior classes to confine applications. Technical Report TRCS99-15, University of California, Davis, 31, '99.

[2] Anurag Acharya and Mandar Raje. Mapbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[3] Jeffrey L. Berger, Jeffrey Picciotto, John P. L. Woodward, and Paul T. Cummings. Compartmented mode workstation: Prototype highlights. volume 16, pages 608–618, Piscataway, NJ, USA, 1990. IEEE Press.

[4] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting legacy code for authorization policy enforcement. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 214–229, Washington, DC, USA, 2006. IEEE Computer Society.

[5] D. Kilpatrick, W. Salamon, and C. Vance. Securing the x window system with selinux. Technical report, NAI Labs, January 2003.

[6] Theodore A. Linden. Operating system structures to support security and reliable software. *ACM Comput. Surv.*, 8(4):409–445, 1976.

[7] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly, 2004.

[8] J. A McLean. A comment on the basic security theorem of bell and lapadula. In *US Naval Research Library*, 1985.

[9] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, 1999.

[10] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[11] Jonathan S. Shapiro, John Vanderburgh, and Eric Northup. Design of the eros trusted window system. In *13th USENIX Security Symposium*, 2004.

[12] Chris Tyler. *X Power Tools*, pages 186–187. O'Reilly Publications, 2007.

[13] Prem Uppuluri. *Intrusion Detection/Prevention Using Behavior Specfications*. PhD thesis, SUNY Stony Brook, August 2003.

[14] V. Rajegowda V. Diwakara and P. Uppuluri. Preventing information leaks in xwindows. In *Communication Network and Information Security (CNIS)*, 2005.

[15] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.

Authors Biography

Dr. Prem Uppuluri is an Assistant Professor of Information Technology at Radford University, U.S.A. His research interests are Computer Security and Grid computing.

Dr. Alok Tongaonkar graduated with a PhD in Computer Science from Stony Brook University. He currently works in Qualcomm Innovation Center Inc, U.S.A. His research focuses on performance optimization mainly in the fields of computer networks and compilers.

Vikas Rajegowda and Vivek Diwakara worked on the implementation during the course of their Masters Degree program at the University of Missouri Kansas City, U.S.A