AGILE PROTOCOLS: AN APPLICATION OF ACTIVE NETWORKING TO CENSOR–RESISTANT PUBLISHING NETWORKS

by

Robert Preston Riekenberg Ricci

A Senior Honors Thesis Submitted to the Faculty of
the University of Utah
in Partial Fulfillment of the Requirements for the

Honors Degree of Bachelor of Science

in

Computer Science

APPROVED:

Jay Lepreau Supervisor	Thomas C. Henderson Director, School of Computing
Laurence P. Sadwick Departmental Honors Advisor	Richard D. Rieke Director, Honors Program

August 2001

ABSTRACT

In this thesis we argue that content distribution in the face of censorship is an appropriate and feasible application of active networking. In the face of a determined and powerful adversary, every fixed protocol can become known and subsequently monitored, blocked, or its member nodes identified and attacked. Rapid and diverse protocol change is key to allowing information to continue to flow. Typically, decentralized protocol evolution is also an important aspect in providing censor-resistance for publishing networks. These goals can be achieved with the help of active networking techniques, by allowing new protocol implementations, in the form of mobile code, to spread throughout the network.

A programmable overlay network can provide protocol change and decentralized protocol evolution. Such a system, however, will need to take steps to ensure that programmability does not present excessive security threats to the network. Runtime isolation, protocol confidence ratings, encryption, and resource control are vital in this respect.

We have prototyped such a system as an extension to Freenet, a storage and retrieval system whose goals include censor resistance and anonymity for information publishers and consumers. Our prototype implements many of the mechanisms discussed in this thesis, indicating that our proposed ideas are feasible to implement.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to thank my advisor, Jay Lepreau, for providing me with the original idea that was the basis for this work, and for his support and valuable insight throughout the project. My thanks also go to those who created the Freenet Project, which I used as a platform for my work. Many individuals, including Chad Barb, Doug Winters and Chris Goller, discussed with me the concepts presented here, and their perspectives were of great help in refining my ideas. Pat Tullmann helped me with the Java knowledge necessary to complete my prototype implementation. Lastly, I would like to thank my parents, Bob and Cathy Ricci, for their invaluable support and encouragement.

CHAPTER 1

INTRODUCTION

In this thesis, we propose *agile protocols*, an application of active networking techniques to censor-resistant publishing networks. Our thesis is that diversity of hop-by-hop protocols can increase the robustness and resistance to blocking of such networks, and can be accomplished by using mobile code to replace host implementations of network protocols. We outline a model for such a system, explore the issues that must be considered when designing a network that uses agile protocols, and, finally, present the results of building our own prototype agile protocol system. We begin in this chapter by introducing the three existing technologies that are the basis for our work.

1.1 Active Networking

Active networking [33] has been the subject of much recent research [38, 35]. Its aim is to provide the infrastructure for deployment of new protocols on existing networks. In the current Internet, large scale protocol deployments at or below the transport layer[1] do not often occur in practice. This is not due to a lack of motivation for new protocols: modern applications such as streaming audio and video could obtain substantial quality and performance enhancements with updated protocols [24]. The barrier is the difficulty inherent in convincing large numbers of network providers to change installed systems, at large cost and effort to themselves. Such protocols are not generally useful until they achieve wide deployment, making it difficult to start small and grow gradually. As an example, the replacement for

[1]In the traditional OSI network model, the transport layer is responsible for making sure that network packets make it correctly from their source to their destination. The only transport-layer protocol widely used in today's Internet is TCP.

the Internet Protocol (IP), IPv6 [8] has been in the works for over five years now, and is not yet seeing significant use.

Active networking research focuses on allowing network routers to run code given to them by the endpoints of a connection. In the current Internet, endpoints have very little control of how their data is handled by routers while in transit. Allowing custom code to be installed on routers affords endpoints the ability to make non-standard changes to the way their data is handled out on the Internet. Typically, this flexibility is used to improve the performance of network protocols with special needs, such as Quality of Service constraints. The primary foci of active networking research are performance, resource management and security. Both of these topics are important at a local (router) and global (entire network) level—as Wetherall has shown [38], it is possible to enforce them locally, but global security is a much more difficult problem.

1.2 Overlay Networks

An overlay network creates a virtual topology on top of an underlying network—in current practice, usually the Internet. Nodes in the overlay network act as routers, forwarding messages they receive to appropriate peers. Overlay networks are commonly used for testing of new network protocols over existing networks [34].

The richness of the programming environment on an overlay network node, in comparison with that available on a traditional router, makes overlay networks attractive platforms for active networking research. The programming environment on a router is impoverished, since the software needs to run as close to the hardware as possible because of efficiency and latency concerns. Indeed, for these reasons, most packet forwarding is performed by ASICs, specialized hardware that is not software-programmable. In an overlay network, however, software is run with the full power of an operating system under it, with the libraries, process separation, and user authentication that are part of a modern operating system. This gives developers a larger set of tools to use, and allows them to write code that is independent of network hardware, since support for this hardware is handled by the operating system.

Overlay networks are also a good platform for active networking research because they typically have relaxed constraints regarding efficiency and latency. Since the services that an operating system provides have inherent performance costs, there are overheads for overlay networks that do not exist on routers. Compared with these overheads, the processing performed by active network code can be minimal, instead of dominating packet processing times, as it would when running on bare hardware. Because processing overheads are no longer the dominant factor in forwarding performance, more computation can be done per packet without seriously degrading performance. Internet routers are expected to be able to forward packets at line speed (the full bandwidth of each of their links,) and with a minimum of latency. However, introducing operating system overheads makes it nearly impossible to meet these constraints. Thus, overlay networks inherently have far less bandwidth and far higher latencies, and their use reflects this. Furthermore, since overlay network nodes have far fewer flows going through them than do Internet routers, they can afford to devote more resources to each one. On top of this, overlay network nodes, which are typically desktop PCs or dedicated servers, have far more processing power and storage space available to them than dedicated routers, making efficiency a matter of far less concern. Using active networking techniques on overlay networks, therefore, allows us to avoid the difficult problem of efficiency and concentrate on other areas.

Finally, overlay networks are deployed much faster than the core Internet infrastructure is replaced. Thus, it will likely be many years before active routers become common, if they are, in fact, deployed at all. There are many overlay networks, such as those discussed in the next section, that are growing rapidly. We can thus begin to learn about active networks in real-world situations without waiting for the slow-moving Internet infrastructure, providing insight that will be useful to lower-level networks in the future. Hopefully, deploying this technology in a limited capacity will provide a compelling example of its usefulness, and encourage further development and deployment of active networks.

1.3 Censor-Resistant Publishing Networks

Some of the most important potential benefits of the Internet for society are the possibilities it offers for free expression and uninhibited communication. For example, in January 2001, according to a New York Times article [21], a corrupt head of state was toppled from power, "due in no small part" to 100,000 people responding to a "blizzard" of wireless text messages summoning them to demonstrations.

However, the current trends point to a lessening of this freedom, as more and more network providers filter the content that passes through them. China, for example, has begun filtering Internet access to suppress, among other things, material that is considered to be seditious [18]. In the United States, Reuters reports [25] that ISPs are coming under pressure to disconnect customers who participate in censor-resistant networks. Clearly, if the freedom of unhindered communication is to be preserved, active steps must be taken in order to preserve it. Systems for communication that are robust in the face of hostile network environments must be developed and deployed.

Much research has been done recently on this topic. Two high-profile examples are Gnutella [15] and Freenet [5], but there are many others as well, including Publius [36], FreeHaven [10], Hordes [30], Crowds [23], and Mixmaster [7]. Though the exact goals and properties of each of these systems varies, the overall theme is the same: to create networks that are resistant to censorship or that provide anonymity. Many of these solutions take the form of overlay networks, which makes them, as discussed above, attractive targets for active networking research.

One of the most high-profile censor-resistant publishing networks is Freenet, which has been the basis for our work on agile protocols so far. Central to Freenet's strategy for creating a censorship-free, anonymous environment is the distribution of data across a large number of independently-administered nodes. Freenet is decentralized; requests are propagated through a series of peer-to-peer links. When a file is transferred, the file is cached by all nodes on the route between the requesting node and the node where a copy of the file is found, which may not be the original publisher. Frequently requested files are thus replicated at many points on the network, making removal or censorship of files infeasible. A node *forwarding* a Freenet message from one of its peers is indistinguishable from a node *originating* the message, providing a degree of anonymity for the suppliers and requesters of data.

The usefulness of a censor-resistant network is directly related to the number of nodes participating in it. For networks that also seek to provide anonymity, it is important that each user be a member of a large class of users who could have been the source of a given transmission, to prevent an attacker from pinpointing the source. The Hordes paper [30] provides metrics for quantitatively analyzing the effects of network size on anonymity. In addition, in order for a censor-resistant network to be useful, it must have both publishers and readers of information. As the number of users in the network increases, the number of possible $(publisher, consumer)$ pairs increases at a quadratic rate. The amount of content on the network will generally increase with the number of publishers, leading to a greater chance that a particular document will be present. Thus, a user requesting a document has a better chance of it being available, and a document inserted into the network has a greater chance of being read, due to the fact that there are a greater number of users who may request it. In some censor-resistant networks, such as Freenet, all nodes in the network act as publishers, because they provide some of their local storage space for keeping copies of documents that have been inserted by other users.

As overlay networks, censor-resistant networks have one major weakness in addition to the ones commonly discussed and analyzed (the FreeHaven paper [10] contains a good overview of these weaknesses.) They are only as robust as the point-to-point links between their members. This weakness falls into two categories: accidental link failure, and administrative blocking. Clearly, software cannot solve the former—it can only be solved by improved network-layer reliability and redundancy. Software *can* however, attempt to combat the latter, which is form of censorship of the censor-resistant network itself. This will be the focus of the remainder of this paper.

In the next chapter, an application of active networking principles to these censor-resistant networks is outlined. Systems that used the outlined methods are said to be using “agile protocols.” The following chapter addresses security concerns and other potential consequences of our system, and suggests the use of runtime isolation, protocol confidence ratings, encryption, and resource control to combat them. Chapter 4 discusses the prototype we have built, and Chapter 5 proposes future work that could further our knowledge of agile networking.

CHAPTER 2

AGILE PROTOCOLS

In this chapter, we begin by discussing the weaknesses of censor-resistant publishing networks that we wish to address. We then introduce agile protocols, our proposed system for dealing with these weaknesses. After giving an overview of agile protocols, we move on to a discussion of details that must be considered when designing a network that uses agile protocols.

2.1 Weaknesses to Address

There are three basic issues that must be addressed in order to improve the robustness and privacy of a censor-resistant network in the face of attempted censorship. These issues are due to fundamental characteristics of the Internet over which these overlay networks must travel, that cannot themselves be changed. Data traveling over hostile networks is at the mercy of the network operators, who may have the ability to monitor it, block it, or eavesdrop on it.

First, when traveling over a hostile network, packets may be *classified* as belonging to a censor-resistant network. This means that the network addresses of the communicating peers can be identified, and information about the time, frequency and volume of their commination can be discovered. This can be accomplished by looking at the contents of packets or by long-term traffic analysis. An attacker could thus gain information about network membership, compromising the privacy of its users and potentially leading to action taken against them. Classification today is typically done based on the network port number used by a given application. However, it is possible to classify traffic based on more sophisticated methods, such as looking for patterns contained in the data being transmitted, or even the sequences of exchanges used to control the flow of a connection. Such classification requires more processing power than is currently common in Internet routers, but emerging technologies such as network processors [31] and extensible routers [32] have the potential to make this type of attack practical at a moderate cost.

Second, once traffic has been classified, with reasonable certainty, as belonging to a particular overlay network, this traffic can be *blocked*, meaning that packets passing between the peers are simply discarded, never reaching their destination. A censor-resistant network cannot fulfill its goals if the network itself is censored in this manner. For automated blocking to occur, the traffic must be machine-identifiable; some traffic patterns may be recognizable to humans, but not fit easily into automated packet classifiers. Furthermore, the entity doing the blocking will almost certainly require a high degree of confidence in their classification strategies, since overly restrictive blocking techniques will lead to degraded service to other users.

Third, an attacker can *eavesdrop* on the contents of packets traveling between nodes, to discover what type of information is being requested and transferred, breaching the privacy of the censor-resistant network's users. This differs from classification in that not only are the addresses of network members discovered, but the contents of their transmissions as well. This type of attack can already be thwarted by existing techniques such as encryption, but agile protocols can complement this protection by hampering classification. An attacker cannot eavesdrop on a censor-resistant network link whose existence they cannot even discover.

Due to the nature of the Internet, a peer-to-peer link commonly has to travel across several independently-administered networks. This greatly increases the probability that a given node-to-node link will have to cross a hostile administrative domain. Due to the trends toward censorship mentioned in the previous chapter, these problems will be increasingly important to deal with if we wish to maintain unrestricted communications on the Internet.

2.2 Properties of a Solution

The common thread in the issues outlined above is *classification*—traffic cannot be blocked or eavesdropped upon unless it can first be identified as belonging to the censor-resistant network. Our approach, then, is to make it exceedingly difficult for an attacker to recognize the traffic of a censor-resistant network. This can be done by expanding the set of protocols used in the network, so that the attacker cannot reasonably recognize them all. In order to make classification difficult, we can design protocols whose packet formats and behaviors are intentionally hard to recognize. We can also make protocols whose traffic closely resembles that of well-known innocuous protocols, such as HTTP or FTP.

We can attempt to the address the classification problem statically—that is, by distributing censor-resistant network code that comes with a fixed number of protocols intended to be difficult to recognize. While it may be effective in the short term, this tactic is not effective in the long run. Attackers may find previously undiscovered ways to classify traffic, or the characteristics of typical Internet traffic in which peer-to-peer links wish to hide may change. The basic problem is that, with a static system, attackers have a fixed target. Given enough time, it is likely that they will devise ways to classify any static protocol, and increases in computing power will be in their favor. Since the protocol set will be publically available, in the form of the software distribution, an attacker can easily know the entire set.

Since the nodes of a censor-resistant network are not under unified administrative control, even if newer releases of the software contain new, more effective protocols, not all nodes will necessarily be upgraded. Upgraded nodes must still speak the old protocols when talking to nodes that have not been upgraded or incompatibilities will hurt the effectiveness and membership of the network. This effect will greatly slow the rate of protocol change in the network. Additionally, the rate of protocol change is limited by software release rate, and since software is usually centrally maintained, this will create a bottleneck and single point of failure for the introduction of new protocols.

2.3 Overview of Agile Protocols

Instead of using a static solution, we would rather address the classification problem dynamically, by allowing the network's protocol set to grow after the network has been deployed. Typically, the method of passing messages between two peer nodes in an overlay network is implemented with code compiled into the released executables. If we replace this with code that can be dynamically loaded while the program is running, then nodes can change the protocol they are using by loading a new protocol implementation. We can take this one step further, and allow peer nodes to be the source of new protocol implementations, thus permitting the spread of new protocols throughout the entire network. We call the passed code implementing a new protocol a *protocol object*, and a network which uses protocol objects is said to be using an *agile protocol*.

New releases of the software can include new protocols, and nodes running the old software release will be able to "learn" these new protocols when they first contact an updated node. Of course, the writing of new protocols is not limited to the core software developers. Since any node can pass a new protocol to any of its peers, a new protocol can originate from any user's node. Any user of the censor-resistant network can write their own protocol and release it into the network. There is no central bottleneck or point of failure for the deployment of new protocols, in keeping with the decentralized philosophy adopted by most censor-resistant networks. The group of potential protocol authors, and thus the potential rate of new protocol introduction, will scale with the size of the user base. In a large network, the rate of protocol change could be very high. By making the protocols upgradable in place, we give users of the censor-resistant network the means to stay one step ahead of attackers. With a protocol set that is not under central administrative control, attackers must constantly develop new ways to recognize newly released protocols, and cannot even know how much of the protocol set they have discovered. The keys to allowing information to flow in the face of attempted censorship are frequent, diverse, decentralized, and locally adaptable protocol change, to ensure a dynamic protocol set that is not known by attackers.

Agile networks are similar to active networks, but differ in that new protocols do not need to be deployed along the entire path of a communication session—for our purposes, they need only be used on a hop-by-hop basis. In fact, for the purpose of defying classification, the more protocols used along a single route, the better. It is local operation of protocol objects that we concern ourselves with, rather than the global security of the network as a whole, whose semantics we have not changed. Rather than using the replaceable protocols for performance reasons, we instead use them to make classification difficult.

2.4 Details of an Agile Protocol System

Now that we have given an overview of the problems that agile protocols are meant to address, and the methods by which we plan to address them, the remainder of this chapter will be dedicated to discussing some of the details involved in making a system based on agile protocols work.

2.4.1 Mobile Code

Since it is desirable to have many network members (See Section 1.3) we allow nodes that differ in platform and operating system to use agile protocols. Thus, we will need to implement protocol objects with mobile code. For our purposes, mobile code is defined as code that can be sent from one host to another, and executed on the second host, regardless of the operating system and platform of the receiving host. Traditional pre-compiled object files or executables will clearly not suffice, as they are highly dependent on the environment in which they were compiled. Another possibility is to send the source code for the protocol, to be compiled by the receiving host. This is somewhat more flexible, but requires all users to have a compiler, introduces significant resource and time overheads, and is still somewhat dependent on host environment. It is possible to use an interpreted scripting language such as Perl [37], Python [20], or TCL [22] for this purpose, but such languages generally do not support sufficient security provisions to address the problems discussed in the next chapter.

The Java [16] language is ideally suited to our purposes. Java source code is compiled to bytecode[1], which is executed by a virtual machine. The Java Virtual Machine can be run on many different platforms, and is widely available–for example, most modern web browsers contain a JVM. Thus, we can compile code once, have strong control over its execution, and distribute protocol objects to users with a wide variety of systems.

2.4.2 Bootstrapping

When two hosts have already established communications they can easily agree on new protocols to speak, and swap code for these protocols. However, initiating communication may be difficult. The initiating node must now know not only the network address of its prospective peer, but also which protocol it is speaking. However, since the initiating node must already obtain its peer's network address through some method, we are not introducing a new problem. The existing mechanisms can be extended to encompass the additional information required for agile networking. A larger quantity of data will now be required for peer discovery, so that it will no longer be feasible to communicate this information verbally. Node discovery with our additional information is still feasible, however, using floppy disks, Java applets, email attachments, or similar electronic exchange. Note that there are two important considerations here. The first is that the peer may be using a protocol not already known to the initiating node. Thus, peer discovery mechanisms should include the appropriate protocol object when possible. Secondly, once a node has advertised itself as speaking a given protocol, it should continue accepting connections with that protocol for a reasonable amount of time, though those connections could immediately be switched to a new protocol if desired. To address this issue, it is useful to include a timestamp in advertisements, indicating that a node guarantees that it will accept connections with the given protocol for a given amount of time.

[1]Java bytecode resembles the instructions that are usually executed on physical processors, but is executed by software rather than hardware

2.4.3 Identification

Attaching a name to a protocol object can be problematic. When inquiring whether or not a peer has a given protocol object, there must be some way to uniquely identify it. Clearly, there is no guarantee that independent developers will uniquely name their protocols, and it might be advantageous for an attacker to misrepresent a protocol object, masquerading it as another. This can be solved by identifying protocol objects by a hash function of their bytecode. With a sufficiently large result, the chance of name collisions will be negligible. Furthermore, if we use a cryptographic hash[2], we can be sure that an attacker is not masquerading one protocol object as another.

2.4.4 Planning Ahead

In the event that communications using a certain protocol are blocked, nodes should attempt to re-establish their connection. In order to do so, they should agree upon the next protocol to use. If an attacker discovers their communication, and begins blocking traffic that matches the protocol objects they are using, then it is likely, though not certain that another protocol object will still be able to make it through. If a node loses communication with a peer, it can attempt to restore communication with its fallback protocol. To increase robustness, this idea can be extended to an arbitrary number of protocols to fall back on, decreasing the likelihood that the attacker will be blocking them all.

2.5 Types of Protocol Objects

The range of functions performed by new protocol objects can be very wide, and while there need not be a strict distinction of protocol types in the censor-resistant network, it is useful to distinguish between these types for the purpose of discussing them. New protocols can, and likely will, implement a combination of the features described here.

[2]The primary property of a cryptographic hash is that it is prohibitively difficult to construct a message with a given hash code.

2.5.1 Network Port Number

The simplest way to change the appearance of a protocol is to change the TCP or UDP port number that it operates on. A protocol object could choose to operate on a well-known port used by another protocol, such as HTTP or FTP, in order to get through firewalls that permit the use of these ports. It could also change ports frequently, to prevent firewalls that block traffic based on port number, as most current firewalls do, from having a static target to block. This type of change is quite superficial, however, because network port numbers are simply a logical way of distinguishing traffic, and the contents of the protocol's packets may still contain patters that can be used to classify it.

2.5.2 Data Formatting

Another simple method for creating a new protocol object is to simply change the way that data is formatted when sent to a peer. This should be sufficient to thwart most current detection schemes, but may not be sufficient against more sophisticated attacks. Disguising the censor-resistant network's protocol as another, such as HTTP, FTP, or NNTP falls under this category. Data formatting changes are straightforward to implement and do not require any special information or network access. This type of agile protocol must simply act as a transformation function. On the sending side, it should translate the data given to it into a form that can be translated back into the original data on the receiving side.

2.5.3 Data Hiding

In addition to disguising the *protocol*, a protocol object may wish to disguise the *data* that passes through it. Steganography [19], for example, is the art of hiding data within other data. This technique is commonly used with images, where slight bit manipulations can be used to embed information without significantly changing the visual quality of the image. Thus, an observer sees only the cover image, and not the actual data being transferred. Another possibility would be to translate binary data into something resembling written language. The possibilities are endless, and because of the nature of agile protocols, new techniques can be deployed as soon as they are developed.

2.5.4 Spread Spectrum

A protocol object could use spread-spectrum techniques to make analysis and eavesdropping difficult. This tactic involves dividing data to be transmitted into small fragments, and sending these fragments separately. The receiving side then reassembles the fragments to recreate the original data. Individual fragments can be sent using disguising protocols or be hidden in other data. In TCP/IP, these fragments can also be sent over different port numbers.

2.5.5 Control Flow

If classification and filtering techniques become powerful enough, simple data formatting changes may not be sufficient to escape notice. A protocol could be recognized by its handshake patterns, flow control behavior, and so forth. So, the implementor of a protocol object should have the power to change or replace these behaviors. This goal requires that the protocol object operate on a lower level than a simple data formatting or hiding function. The API presented to protocol objects, then, should allow the protocol object to be more than a filter or transformation function—it should be allowed direct access to the underlying transport (for example, a TCP socket.) Doing so opens some security concerns, however, which will be discussed in the next chapter.

2.5.6 Alternate Transports

The most complicated type of protocol change is one that is able to use a different type of underlying transport altogether. This would be particularly useful in unusually restricted environments. For example, a protocol object capable of tunneling communication through Web proxy servers would fall into this category, and would allow nodes to operate behind many firewalls that would normally prevent them from participating. Another form of tunneling would be to send email back and forth between two nodes, resulting in a very slow connection, but one that is able to get through almost any firewall, since most organizations with Internet access allow their users to use email. Protocol objects such as these, however, would require lower-level network access than normal protocols, and may need to contact a third party. As with control flow modifications, such access presents a clear security threat.

2.6 Simplicity

The benefits from agile protocols come from users' ability to write and deploy protocols themselves—if the only protocols available are a small set written by the authors of the core system, then we have gained very little by making them agile. Thus, an important aspect of an agile protocol system is that it must be simple enough that there is a large subset of users who can implement new protocols. There should be a well-defined, simple API (ideally, consisting of a small number of functions) that protocol objects must implement. Example source code for Protocols objects will ease this task—the developer version of the core system should include source code for the protocols it includes. In most cases, simple variations, which can be obtained with minimal modification of existing source, will be sufficient to make matching mechanisms for the old protocol ineffectual against the new one. For example, a protocol that disguises itself as HTTP may be recognizable by the URL it generates, or the order in which it sends headers. Simple modifications to the URL generation schemes could make messages unmatchable by old filters. The simpler the process of creating a new diverse protocol object, the faster the rate of new protocol introduction by independent authors will be.

CHAPTER 3

SECURITY CONCERNS

Using agile protocols for censor-resistant networks offers some clear advantages. However, the ability to run mobile code on other nodes has serious security implications. And, since protocol objects can be spread from peer to peer, a defective or malicious protocol object could have devastating impacts on the whole network. In this chapter, we outline an attack model for networks using agile protocols, and propose methods for thwarting these attacks. Since agile protocols could be applied to many different types of censor-resistant networks, we do not discuss the weaknesses of the network itself, instead limiting ourselves to weaknesses of point-to-point links. For discussions of attacks on the censor-resistant network itself, such as flooding and data corruption, see [5, 10, 26].

3.1 Background on Attacks

In this section, we outline the model for attacks that may be directed against a censor-resistant network, in order to have a framework for the discussion of our defenses against these attacks. First, we outline the motivations that an attacker may have for attacking the network, in order to understand what type of threat they present. Then, we outline the differing levels of capability that the attacker may have. Finally, we combine the two, describing a set of likely attacks against the network.

3.1.1 Motivations of Attackers

First, the “attacker” may not be malicious at all. Given that a new protocol object can be inserted by any user in the network, it is unreasonable to assume that all protocol objects will be written well enough to function correctly. A usable agile protocol system must be able to cull non-functional protocol objects and continue to operate.

The attacker's goal may simply be the collection of statistics on the censor-resistant network. They may wish to determine how many nodes belong to it, how many files it contains, what types of files are stored on it, etc. While this behavior in itself is benign, it may be used for undesirable purposes, and undermines the fundamental principles of anonymity that most censor-resistant networks hope to uphold. This motivation seems most likely among researchers or developers of the software.

The aim of an attacker may be to collect information about specific nodes or users of those nodes. This may include the personal information, network address, a list of files that a node operator has requested, files stored on a node, etc. This information could be used to take action against node operators, such as disconnection from the network or litigation. It could also be gathered by marketers wishing to target users for unwanted advertisements, such as the current "spam" email that is common on the Internet.

An attacker may wish to censor certain documents. This may be attempted at the underlying network layer (ISP or network backbone), or by inserting a malicious protocol object that blocks searches or transfers of the documents to be censored. Countries with repressive governments may try to censor documents they see as seditious or revolutionary, as with the China example given in the Introduction. In the U.S.A., under the provisions of the Digital Millennium Copyright Act[11], Internet Service Providers may be pressured by third parties to remove certain content from their networks, and may resort to these tactics to do so.

Some attackers may wish to block transmissions of the censor-resistant network entirely. A network provider or government may resort to this tactic if targeted blocking is ineffective, if the majority of data on the network is objectionable, or if they feel that it is an inappropriate use of resources.

Finally, an attacker may wish to compromise the network nodes themselves. They may wish to do this in order to use those nodes to launch other attacks, to use the node's resources, or merely for thrills.

3.1.2 Capabilities of Attackers

The attacker may have only the ability to passively monitor a network, but not alter traffic passing over it. Two types of attackers are likely to have this type of capability. The first are "third parties" such as governments or corporations who can persuade or coerce network providers to install monitoring equipment. This type of monitoring is already in use by the United States government, in the form of the "Carnivore" system [4]. A worldwide message interception and monitoring system known as "Echelon" has also sparked international controversy and prompted an investigation by the European Parliament [12]. Second, the network provider may not have the capacity to do realtime monitoring, but be able to post-process logs to search for connections. This would allow them to discover connections, but only *after* they have occurred.

The attacker may have the ability to force disconnection from the network, either directly, as in an ISP, or indirectly, as in a government or corporation persuading an ISP to take action. Here, mere network membership may be grounds for disconnection, or the attacker may have to prove that a network member has transferred and/or stored objectionable material.

The attacker may have the ability to put up filtering rules, without complete disconnection, and to alter the data that passes through links. This capability seems likely at an ISP level. Internet backbones, on the other hand, typically carry enough traffic that this is not possible with current technology. However, advances in router technology such as those mentioned in Section 2.1 may make this type of attack feasible for backbones providers in the future.

Finally, the attacker may have the ability to inject malicious protocol objects into the system. Since the censor-resistant publishing networks discussed in this thesis are open networks—that is, anyone can join them—nearly any attacker can have this capability. It does not require privileges on the Internet networks over which the censor-resistant network's traffic must pass, and it does not require legal backing. Malicious protocol objects can also spread over the entire network, so attacks mounted in this manner are not local as the other capabilities are. Thus, it is the most dangerous type of attack, and one that we must sufficiently address lest the dangers to which we expose the censor-resistant network user outweigh the advantages we offer.

3.2 Threat Model

Now, we combine the motivations of attackers, along with capabilities they may have, to describe a model for attacks that are likely to threaten a censor-resistant publishing network.

3.2.1 External Attacks

External attacks are those against the underlying transport mechanism of the censor-resistant network. Attacks using monitoring, disconnection, and blocking fall into this classification. The primary defense against these attacks is the dynamic protocol set mentioned in the previous chapter. Since all of these attacks rely on being able to classify the traffic of the overlay network, defenses against external attacks will all rely on the ability to defy classification.

The defense afforded by having a dynamic protocol set can be divided into two categories: the diversity of protocol objects[1] and the activity of the set[2]. The diversity of the protocol set is defined to be the number of different distinct patterns used by the protocols in it. Writing classifiers for all protocols in use will require manual work by the attacker in proportion to the number of protocol objects in use. In addition, the processing power required for simultaneously checking against a large set of protocols will require significant processing resources, and a sufficiently large set will require more processing power than is readily available, especially for high-traffic backbone providers. If an attacker cannot simultaneously watch for the entire protocol set, then some protocols will certainly be able to get through. Rate of protocol change over time is also important, because we assert that any protocol

[1]That is, the size of the protocol set.

[2]That is, the rate at which the set changes over time.

can be classified given enough time and traffic. Thus, to be effective, new protocols must be introduced at a rate such that protocols are being introduced faster than attackers can come up with reliable methods for recognizing them.

3.2.2 Internal Attacks

The second type of attack is one launched against the system by inserting a malicious protocol object into it.

Attackers who wish to discover network membership and topology may attempt to create protocol objects to collect this information for them. There are two ways of doing this: to contact a third party[3] and divulge this information, or to tag the data being sent through the protocol object. Thus, we must prevent protocol objects from contacting third parties, and deny them knowledge of the address of the node they are running on, so that this information cannot be leaked through other means.

Given the large number of computer viruses and worms that have been seen on the Internet recently, it seems quite likely that people will attempt to write viruses for an agile network, using that network's own protocol distribution system to propagate it. Thus, we must ensure that nodes are protected against malicious code that attempts to read, modify, or delete users' files. We should also limit protocol objects' ability to use node resources, so that an attacker cannot gain unauthorized access to them.

Since some attackers may wish to interfere with the operation of the network, or may have bugs in their protocols that prevent them from functioning correctly, our system must be able to detect nonfunctional protocol objects, and prevent their spread.

Some attackers may wish to censor selectively, introducing protocol objects that fail only when transferring files that match a certain pattern. As we shall see in the next section, the strategies for dealing with targeted protocol failure are different from those for combating large-scale failure.

[3]That is, a host that is not one of the two communicating peers.

A censor-resistant network such as Freenet is designed to find information even if some nodes and point-to-point links go down. However, if a large number of links go down simultaneously, this could cause serious problems with message routing. So, our methods for thwarting attacks should deny protocols the ability to synchronize failures.

3.3 Thwarting Attacks

3.3.1 Isolating Protocol Objects

In order to prevent protocol objects from compromising local information on a node, or from using the host's resources in an inappropriate manner, it is necessary to keep the execution of protocol objects isolated from the core of the overlay network. In particular, we need to keep protocol objects from accessing information about peer nodes, locally stored files, and the node operator. We also need to prevent them from accessing the network directly, so that they cannot contact a third party to divulge information. This isolation also includes being prevented from sharing information with other protocol objects, so that malicious protocol objects cannot collaborate. Our approach to isolation is based upon the "principle of least privilege," that protocol objects should start with no privileges at all, and then be given only carefully chosen privileges that are essential to their proper operation.

We find that protocol objects actually need very little information to be able to function. They need access to the message they are to send or receive. They also need to be able to place data on the network to be sent, and to retrieve data off the network to receive messages. However, for most protocol objects, the network access need not be direct, and they do not even need to be able to find out information such as network addresses, so long as connections are set up by the core system. There is a tradeoff here, however, in that increased network access allows increased flexibility, such as the alternate transport protocol objects mentioned in Section 2.5.6.

An interesting aspect of the isolation of protocol objects is isolation from time. If protocol objects are allowed access to the system clock, then it is possible for an attacker to write a protocol object that will fail in a concerted manner on machines throughout the network. However, if we prevent protocol objects from having any sense of time whatsoever, this prevents them from performing many actions that require knowledge of time, such as measuring amount of bandwidth they are using, or using timeouts. These functions, however, require knowledge of the *passage* of time, rather than knowledge of absolute time. Thus, protocol objects should be given a function that returns the amount of time since it was last called—this allows such actions without giving an attacker a way of implementing massive simultaneous failure.

One very effective way of achieving this isolation (and the method we used in our prototype implementation) is to use a typesafe language. In such a language, we can control the types of objects to which a protocol object may resolve references. Thus, objects of protected types are opaque to the protocol object; that is, it may be able to determine their existence, but not their contents. In this way, we can make sure that protocol objects do not access restricted sections of the core, external resources, or each other. This model is already widely used in Java Virtual Machines embedded in Web browsers [17], but the restrictions we require will be more stringent than the Java applet security model, due to the anonymity that many censor-resistant publishing networks seek to provide. There are other ways of enforcing this isolation, such as running protocol objects in a separate operating system processes, but such methods are likely to be less efficient.

3.3.2 Rating Protocol Objects

In order to prevent the spread of protocol objects that fail to correctly transfer files, each node can maintain a rating, called a confidence level, of each protocol it knows. The confidence level in a given protocol starts at some constant value, and is assessed a penalty for each unsuccessful transfer through it. Confidence is increased through subsequent successful transfers. Only protocol objects with high confidence ratings, called acceptable protocols, will be passed on to peers, thus limiting the spread of protocol objects that do not function correctly. Observably malicious behavior[4] should incur higher penalties than failures which could result from normal circumstances.

There are a few important characteristics of this rating system that should be noted. First, it is important that it be maintained independently on a per-node basis. If nodes were allowed to report their established confidence level to peers, an attacker could use this mechanism to provide bogus rating information. Additionally, confidence levels should be relative, not absolute. If a fixed cutoff point for confidence is set, network conditions (such as the availability of certain files or the percent of requests that are for non-existent files) can occur such that all protocol objects have a rating below this threshold. Thus, the threshold for acceptability should be dynamically changed to reflect observed success rates, or confidence levels should be evaluated against each other—for example, considering the protocols with the n highest confidence levels to be acceptable.

One attack strategy would be to write a protocol object that only succeeds under a small set of circumstances, then pass that object to a peer. The attacker could perform a large number of transfers through that protocol object, under the circumstance in which it is successful. This would inflate the confidence rating, and the new node would be likely to pass the protocol to its peers. However, at that point, the attacker can no longer artificially inflate the rating of the protocol object, and it will be rated by its actual performance. In order to promote the protocol object in question, the attacker would need to be able to continue to contact every node that the malicious protocol spreads to and perform similar transfers, which is not a feasible proposition due to the fact that censor-resistant networks are designed to prevent knowledge of the overall network topology.

Confidence levels can be determined in two separate, complementary ways. The first is the method used in the previous example—calculating confidence by observing normal network traffic. The problem here is that a node cannot always be

[4]In some systems, such as Freenet, data can be identified using a cryptographic hash of its contents, allowing us to know with certainty whether the data returned is the data that was searched for.

certain of what response it was supposed to get back—a request could fail because the protocol object has relayed it incorrectly, or simply because the data could not be found. The second method of determination solves this problem, but incurs additional overhead. A node can evaluate a protocol object by sending messages to itself through a given protocol object. In this case, the node knows exactly what message should be recieved, and can compare the correct behavior to that observed. When a message is sent to a peer, a node can also send a copy of that message, through loopback, to itself, to ensure that the protocol object handles it correctly. However, this method is more expensive, in terms of processor and memory usage than the other, because it requires additional action to be taken, instead of simply adding low computational cost to activities that would occur anyway. Therefore, we recommend using ratings based on normal traffic, supplemented with occasional loopback tests. Failure in the should be weighted more heavily. In order to reduce the impact of the additional resource consumption, a smart scheduler can be used to only perform loopback tests when the system is idle.

While we believe that this will be an effective system for preventing the spread of malicious or poorly-written protocol objects, it is still theoretical at this stage. Its effectiveness will need to be analyzed, and perhaps improved, based on future experiences with it.

3.3.3 Hiding the Nature of Requests From Protocol Objects

One problem that protocol object ratings do not solve is the problem of selective failure by protocol objects. If a protocol object is programmed to fail only for certain requests, which constitute a smaller percentage of requests than the percentage of requests that would normally fail (for example, searches for files that do not exist,) then this will not be noticeable to the above algorithm. So, we must not only prevent the protocol objects from getting local node data (as described in Section 3.3.1), but also from getting information about the data it is transmitting. A reasonable way of doing this is to have communicating nodes perform a Diffie-Hellman [9] key exchange, which allows them, over an insecure channel (in this case, the protocol object) to agree on a secret encryption key. This key can then be used to encrypt data before it is sent to the protocol object. This way, a protocol object cannot alter its behavior based on meaningful information about the data being sent through it. Of course, it can still alter its behavior based on the encrypted data that it sees, but this will be effectively random, and not useful to an attacker.

Though Diffie-Hellman key exchanges may seem ideal for this purpose, they have one important weakness. They are subject to man-in-the middle attacks, in which a party listening on the communications channel (in this case, the listening party could be a protocol object) can impersonate one endpoint or both, compromising the key. This problem is generally solved by using a public-key cryptography system [27] to authenticate the communicators. In order for this to work in a censor-resistant network, keys would need to be securely exchanged between peers. Due to the complexity of this process, and the public-key infrastructure it requires, this level of protection may be difficult to implement in deployed censor-resistant networks. Another possible solution is to perform a key exchange using a protocol object other than the one that the generated key will be used with. This solution should be generally acceptable, though it does have a slight weakness in that co-operating protocol objects may still be able to compromise the key.

3.3.4 Resource Control

Most modern operating systems allow coarse resource control, allowing node operators to limit the overall resource (CPU, memory, etc.) usage of the process in which the censor-resistant network core runs. Finer grain controls, such as limiting the CPU usage of an individual protocol object, will require support from the language in which the system is written. As an example, Java has been the focus of much recent resource-control research efforts, such as KaffeOS [1] and Janos [13]. Agile network code can be run in a JVM that supports these resource controls in order to prevent malicious code from using more than its share of the CPU and memory.

CHAPTER 4

PROTOTYPE IMPLEMENTATION

In order to explore the concepts outlined in this thesis, and as a proof-of-concept, we have built a prototype agile protocol system. Our prototype is based on Freenet 0.3.5 [14], an implementation of Freenet led by the author of the original Freenet paper [5]. It has the capability to completely replace protocol details via mobile Java bytecode, and offers facilities for doing this replacement while the system is running. It implements protocol object isolation, and provides restricted network access for protocol object flexibility. It does not, however, implement encryption of messages beyond that provided by the base Freenet system, or protocol object ratings. We have written three sample protocol objects to illustrate and explore the types discussed in section 2.5. On the whole, we are pleased with the results, and believe that such a system could one day be used in production environments.

4.1 Results

4.1.1 Architectural Changes

In order to integrate our changes with the base Freenet system, had to make some architectural changes to Freenet. We implemented our changes primarily in two different layers. The first is the `Transport` layer—the only currently implemented transport in Freenet is TCP. The second is what Freenet calls the `Presentation` layer—its job is to take messages [1] the Freenet core, and pass them over the network. In most network software, it would be considered to be part of the Application layer, because it is responsible for passing user data to the underlying

[1]In Freenet, a message is the basic unit of communication. The most common messages types are `DataRequest`, which requests a file, `DataReply`, which includes the contents of a requested file, and `RequestFailed`, which indicates that the requested file was not found.

`Transport` layer. The fact that Freenet makes the distinction between the handling of the messages that make up the global operation of the network and the passing these messages to the `Transport` layer proved very useful to us. It allowed us to concentrate on modularizing exactly the portion of code that is most useful to agile networking.

The `Presentation` layer is where we concentrated our work. It needs no knowledge of the underlying transport, and thus fits our need for isolation well. We replaced the `Presentation` layer provided with Freenet with a wrapper class, capable of determining which protocol object needs to be invoked for each message. When the core Freenet system calls functions in our wrapper class, the function calls are passed down to the appropriate protocol object. These function calls are of basically two types. The first takes a Freenet message object and an open `Transport`-layer connection [2] and writes the message to the connection. The second takes an open connection and reads a Freenet message from it.

In order to provide protocol objects with more flexibility than could have been obtained operating solely at the `Presentation` layer, we provided a wrapped network class for their use. This class gives protocol objects the ability to open up its own TCP connections to a peer node, in addition to the `Transport`-layer connection managed by the core. This allows for spread-spectrum or port-hopping schemes. This class, called `RestrictedNetwork`, only allows the protocol object to open connections to the peer to whom it is currently talking, to prevent disclosure to an outside party. The sockets returned by opening these connection are also restricted, so that the protocol object cannot discover the IP address of the node it is running on or its peers, preventing protocol objects from discriminating against certain hosts or inserting host addresses into the data for tracing.

[2] A connection is a logical transport-layer session over which messages can be sent. Connections are managed by the core Freenet system. Since Freenet only currently implements a TCP transport layer (though the system has been explicitly designed to allow other transports to be added) connections correspond to TCP sockets.

4.1.2 Node Addresses

One important aspect of our new system is the way in which node addresses are represented. The core Freenet system represents addresses by a string in the form `transport/host:port`—for example, `tcp/eureka.cs.utah.edu:1234`. This scheme is fairly inflexible for agile protocols, because it assumes that the host and the port are the only information needed by the `Transport` layer, and does not provide a way for us to specify which protocol object is being used. Due to the way that the specification for Freenet addresses is written, it is easy to extend the address scheme by providing a new address parsing function based on the “transport” field. So, in our prototype, addresses are represented in the form `agile/transport():protocol`. The parentheses after `transport` are intended to be used to pass arguments to further customize behavior. For example, the `tcp` transport takes two arguments: the target host and port number. The equivalent of the last example under our new scheme would be `agile/tcp(eureka.cs.utah.edu,1234):FreenetProtcocol` [3]

4.1.3 Changing Addresses

Protocols in our system can be used asymmetrically. That is, communication between two nodes can use different protocols in different directions. Each node informs it peers of the addresses it is listening on—the TCP port it is listening on, and the protocol object in use on that port. At any time, the node can inform any or all of its peers that they should begin using a different address to contact it. Our prototype assists in the management of listening addresses by keeping track of the number of peers using each address, opening new ports the first time they are used, and closing them when no more peers are communicating on them.

To inform peers of a new address to use, we have introduced a new message type to Freenet, called `AddressUpdate`. Along with the new address to use, in the

[3]`FreenetProtocol` is the name associated with the traditional Freenet protocol—see Section 4.1.6 for a discussion of our other protocol objects. For simplicity, we chose to use names, rather than hashcodes, to identify protocol objects in our prototype. Changing to use hashcodes instead would be a trivial matter

form specified in section 4.1.2, this message includes all of the bytecode necessary to speak the new protocol. This bytecode may implement several Java classes, which together comprise a single protocol object. Upon receipt of such a message, a node updates the table in which it stores addresses for all of its peers, and any subsequent messages it sends will use the new address.

4.1.4 Loading Protocol Objects

In our prototype, `AddressUpdate` messages are accompanied by the bytecode for the protocol object that the sender wishes to speak. This may constitute any number of Java objects, so that custom utility classes can be included. The receiver of such a message checks its store of protocol objects, and inserts the new protocol object if it was previously unknown.

When the core Freenet system wishes to make a connection, it instantiates a wrapper class called `AgileProtocol`. This class determines which protocol object should be used for the communication, and acts as an intermediary for all subsequent function calls. In this manner, the core Freenet system need not concern itself with the different protocol objects used at the `Presentation` layer.

4.1.5 Protocol Object Isolation

Our prototype takes advantage of Java's type safety to prevent protocol objects from gaining access to sensitive APIs of the core system. Loaded protocol objects are not allowed to resolve references to classes that are not strictly necessary to their operation. This prevents them from compromising inappropriate local information, as discussed in the previous chapter. It also denies them access to subsystems such as networking, so that they cannot contact third parties, run commands on the local node, etc.

4.1.6 Sample Protocol Objects

For our prototype, we have implemented three new protocols, in addition to re-working the standard Freenet protocol to work on the new system. We feel that the task of writing a new protocol object is sufficiently easy, as discussed in Section

2.6, that a large number of users will be able to implement their own. Writing a protocol object that uses alternate data formatting is as simple as replacing four functions in the code for an existing protocol.

The first protocol object we implemented is one that disguises Freenet traffic as HTTP. The primary functionality of this protocol is the transformation it performs when putting data onto the network, and reading it off again on the other side. This transformation includes the insertion of misleading data in order to mimic real HTTP headers. It also makes some control flow changes. The standard Freenet protocol does some initial negotiation that is unlike anything used by HTTP, so our protocol omits these steps so as maintain the appearance of being HTTP.

Our second protocol object implements a simple port hopping scheme. After every message, a new port number is chosen for the next message. This protocol shows that our infrastructure correctly handles listening on new ports, and ceasing to listen on ports that are no longer in use. It also demonstrates the ease of address changing in our system—the protocol object need only call a single function, which handles updating references to the peer node, and takes care of informing the peer of the new port number to use. Though the algorithm used is very simple (it simply increases the port number by one for each message), it could easily be replaced by an algorithm of arbitrary complexity.

Our third protocol object was created as an example of a spread spectrum protocol, and to test our `RestrictedNetwork` class. It breaks up messages into uniformly-sized chunks, and sends each of these chunks on a separate TCP port. Depending on a value set at compile time, the message can be divided on any arbitrary byte boundary, down to a single byte. There are tricky issues involved in such a protocol, such as determining which ports are available on each host [4] and signaling the end of a transmission [5] that we did not investigate in depth, but we are confident that we have shown that such a protocol is possible to implement.

[4]Some ports may already be in use by other programs.

[5]The size of the message cannot be known to the protocol object until transmission is completed. This is largely an artifact of the Freenet implementation we based our work on.

4.2 Implementation Challenges

One of the biggest implementation challenges was fitting our changes into the model used by the Freenet code. For example, the core Freenet system does not need to be able to update information about its peers (for example, changing port numbers,) so many copies are made of this information. In Agile Freenet, however, peer addresses are updatable, so we had to come up with a mechanism for keeping this information up-to-date when peers decide to speak a new protocol. Our solution was to maintain a table of the current contact information for each peer, and when an address was retrieved from one of the copies, to look up its “real” information in this table. In turn, this introduced the problem of uniquely identifying nodes, which we chose to do by host IP address.

Bootstrapping clients also proved to be a problem. Nodes discover the correct addresses and protocols for their peers at initialization time, using a configuration file. However, since clients cannot be known during initialization, their contact information must be discovered after they connect. Our first method for doing this involved implicitly guessing addresses for unknown hosts when they first connected, but this proved to be problematic in the case of a second client connecting from the same host, which is a very common occurrence, Instead, we updated the clients to send an `AddressUpdate` message (see Section 4.1.3) when they first connect to inform the node of their contact address.

CHAPTER 5

FUTURE WORK

In this chapter, we suggest some avenues for further research on agile protocols.

5.1 Wireless Networks

Some types of wireless networks involve using spread spectrum radio transmission[1] or frequency hopping. Currently, this must be done by using algorithms that are agreed upon in advance, since both endpoints of the communication must use the same algorithm. As with static protocols, these algorithms can be discovered and the frequencies they use eavesdropped on or interfered with by an adversary. However, agile protocols may be used to replace these algorithms, allowing nodes to change them in response to attacks.

Wireless networks are particularly interesting for this research, because there is no threat of complete disconnection, as there is in wired networks. In fact, some work has already been done [3] on using active networking in a wireless context. On the wired Internet, a user is dependent upon their ISP for network service, and thus has a single point at which they are vulnerable to disconnection from the network. In a wireless network, however, users can communicate without the use of an intermediary ISP. The layer at which agile protocols would function on wireless connections, the data link layer, is an open resource, the airwaves, rather than a physical network under the control of a network provider. Projects to create cooperative city-scale wireless networks have sprung up in several major metropolitan areas, including Consume.Net [6] in London, SFLan [29] in San Francisco, and

[1]In this context, spread spectrum refers to the practice of transmitting data across a wider frequency range than necessary. This technique is used to resist interference and to share the airwaves with narrow band communications.

SeattleWireless [28]. Projects like this could benefit from agile technology to help continue transmission even if attempts are made to censor them.

5.2 Applications to Other Areas

Agile protocols have unexplored potential in contexts other than censor-resistant networks. These are contexts in which full active networking is not necessary, only protocol replacement on a hop-by-hop basis. In most cases, this will involve protocol replacement at the link layer, as it would with the wireless examples in the previous section. Applications for specialized processing at the link layer [2] that could take advantage of protocol replacement have already been discovered.

Agile protocols could also be used to update ciphers used to encrypt transmissions. If a method for "cracking" the established cipher is discovered, agile devices can be updated to use a new one. This is particularly appropriate for military applications, but could be used anywhere that privacy is important.

5.3 Adversarial Experimentation

Though we believe that our base claim, that agile protocols can be used to increase robustness in the face of censorship, is sound, it is difficult to demonstrate without actual deployment. However, some valuable information could be gained by performing adversarial experiments, in which an agile protocol-based network is established, and experimenters attack it. Such an experiment could vary the number and type of protocols, as well as the capabilities of the attacker in order to learn more about what types of protocol objects are effective at avoiding classification. Such an experiment would also need to provide cover traffic simulating the usage of the real Internet.

While such an experiment could provide useful information, it cannot prove the effectiveness of our methods. Only the active use of a carefully implemented system in opposition to a determined attacker can be a true test of agile protocols. The effectiveness of agile protocols depends on a large protocol set, and a diverse set of protocols developed by different users. An attacker's effectiveness is dependent on expertise and resources invested. Thus, an evaluation of an agile protocol system is really an evaluation of the people creating and blocking protocols. Only in actual practice can we evaluate whether or not agile protocols have given users of censor-resistant networks useful tools to avoid attackers.

REFERENCES

[1] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, October 2000. USENIX Association.

[2] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. *In proc. 1st ACM Int'l Conf. on Mobile Computing and Networking (Mobicom)*, page 10, 1995.

[3] Vanu Bose, David Wetherall, and John Guttag. Next century challenges: Radioactive networks. In *Proc. of the Fifth Annual ACM/IEEE Internation Conference on Mobile Computing and Networking*, pages 242–248, August 1999.

[4] The Carnivore diagnostic tool. http://www.fbi.gov/programs/carnivore/-carnivore.htm.

[5] Ian Clarke. A distributed decentralized information storage and retrieval system. Master's thesis, University of Edinburgh, 1999. Available at http://-freenet.sourceforge.net/freenet.pdf.

[6] Consume.Net. http://consume.net/.

[7] Lance Cottrell. Mixmaster FAQ, 1996. http://www.obscura.com/l̃oki/-remailer/mixmaster-faq.html.

[8] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) specification, RFC 2460, December 1998.

[9] W. Diffie and M. Hellman. New directions in cryptography. In *IEEE Transactions on Information Theory*, volume IT-22, pages 644–654, 1976.

[10] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven Project: Distributed anonymous storage service. Berkeley, California, July 2000. International Computer Science Institute. Revised December 17 2000. Available at http://www.freehaven.net/doc/berk/freehaven-berk.ps.

[11] The Digitial Millennium Copyright Act, United States H.R.2281, November 1998.

[12] The European Parliment Echelon Commite website. http://-www.europarl.eu.int/committees/echelon home.htm.

[13] Flux Research Group. The Janos project Web site. http://www.cs.utah.edu/-flux/janos/.

[14] The Freenet Project. http://freenet.sourceforge.net/.

[15] The Gnutella Project. http://gnutella.wego.com/.

[16] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[17] Sun Microsystems Inc. FAQ – applet security. http://www.javasoft.com/-sfaq/index.html.

[18] J.A. Getzlaff. The great firewall of China. *Salon.com.* February 22, 2001.

[19] N.F. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *IEEE Computer*, 31(2):26–34, 1998.

[20] Mark Lutz. *Programming Python.* O'Reilly & Associates, Inc., Sebastopol, CA, Fall 1996.

[21] Text messaging is a blizzard that could snarl Manila. *New York Times.* January 20, 2001.

[22] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 133–146, Berkeley, CA, 1990. USENIX Association.

[23] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. In *ACM Transactions on Information and System Security*, pages 66–92, 1998.

[24] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality adaptation for congestion controlled video playback over the internet. In *SIGCOMM*, pages 189–200, 1999.

[25] Reuters. Web piracy—crackdown spawns stealth platforms. Available at http://dailynews.yahoo.com/h/nm/20010430/wr/-media_web_piracy_dc_1.html.

[26] Jordan Ritter. Why Gnutella can't scale. http://www.darkridge.com/j̃pr5/-doc/gnutella.html.

[27] R. Rivest, A. Shamir, and L. Adlemann. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, volume 21, pages 120–126, February 1978.

[28] SeattleWireless. http://seattlewireless.net/.

[29] SFLan. http://www.sflan.com/.

[30] Clay Shields and Brian Neil Levine. A protocol for anonymous communication over the Internet. In *ACM Conference on Computer and Communications Security*, pages 33–42, 2000.

[31] Tammo Spalink, Scott Karlin, and Larry Peterson. Evaluating network processors in IP forwarding. Technical report, Department of Computer Science, Princeton University, November 2000.

[32] David E. Taylor, Jonathan S. Turner, and John W. Lockwood. Dynamic Hardware Plugins (DHP): Exploiting reconfigurable hardware for high-performance programmable routers. April 2001.

[33] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[34] C. Tschudin. ANON: A minimal overlay network for active networks experiments. Technical Report TR 98-10, Computer Science Department, University of Zurich, August 1998.

[35] Patrick Tullmann, Mike Hibler, and Jay Lepreau. Janos: A Java-oriented OS for active network nodes. *IEEE Journal on Selected Areas in Communications*, 19(3):501–510, March 2001.

[36] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, Web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[37] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

[38] David J. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, Kiawah Island, SC, December 1999.