Decoy Switching: Circumventing Censorship with Emerging Switch Hardware

Blake Lawson
Princeton University

Jennifer Rexford
Princeton University

Abstract

Internet censorship impacts many people around the world, but the most common censorship circumvention tools, like virtual private networks, proxies, and Tor, can be blocked by repressive governments and internet service providers. There are several new anticensorship techniques that cannot be blocked without blocking the entire internet, but these new techniques are not used in practice because they are not fast enough to keep up with internet traffic. In this paper, we present Decoy Switching, an unblockable anticensorship system, that uses a novel application of programmable switch hardware to scale where past anticensorship systems failed. We found that Decoy Switching has the potential to be thousands of times faster than other unblockable anticensorship systems, which brings us closer to deploying unblockable anticensorship in practice.

1 Introduction

There have been numerous efforts to create network-level anticensorship systems over the last six years, including Decoy Routing [15], Telex [28], Cirripede [13], TapDance [27], and Rebound [11]. In this section, we provide a high level overview of common design choices, and investigate differences between the systems' approaches to similar problems. Before going into detail, it is important to establish some terminology that we use for the rest of the paper. There is an *adversary* that operates a network and does not want people within its network to access some set of internet addresses. There is a *client* that is inside the adversary's network that wants to access a *covert destination* that the adversary blocks. The *decoy destination* is a website or service that the adversary allows users to access. This terminology is based on the definitions in the Decoy Routing paper [15].

All of the existing systems also use similar threat models which assume that the adversary can observe all traffic in its network. The adversary can also drop packets from clients that try to access covert destinations directly. The adversary has no knowledge of or control over traffic outside its network.

1.1 Common Architecture

Existing anticensorship systems use similar high-level designs. Since the adversary cannot meddle with packets outside its network, the anticensorship systems introduce custom network hardware (NH), like routers, proxies, or middleboxes, in friendly ISP's networks outside of the adversary's control. Once the NH is in place, existing anticensorship systems allow clients to access covert destinations by including hidden *tags* in requests to decoy destinations. If the NH is on the path of the client's request to the decoy destination, then the NH detects the client's tag, takes over the client's connection to the decoy destination, and reuses the connection between the client and the decoy destination to relay information between the client and the covert destination. Since the client and the NH communicate over the client's connection to the decoy destination, the adversary cannot detect that the client accessed a blocked address. Figure 1 provides a visual representation of this process.

2 Related Work

2.1 Fundamental Problems in Network-Level Anticensorship

Despite their general similarities, there are common problems that arise in building network-level anticensorship systems that existing anticensorship systems solve differently. Those problems are (1) what network hardware to use and where to deploy it; (2) how the client indicates that it wants to use the anticensorship system without alerting the adversary; (3) how the client communicates the address of the covert destination it wants to

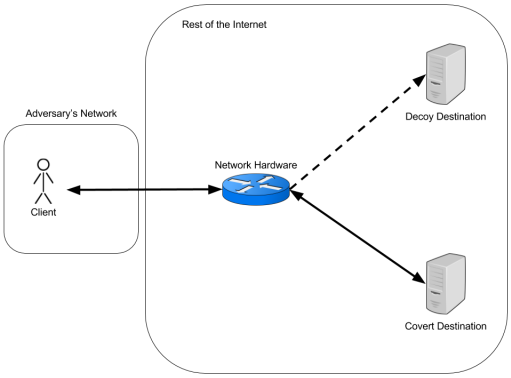

Figure 1: The client makes a request to a decoy destination that happens to go through an anticensorship system's network hardware (NH). The NH detects a secret tag in the client's request and creates a connection to the covert destination on the client's behalf. The NH uses the client's connection to the decoy destination to relay information to the client from the covert destination. The dotted line represents the client's initial connection to the decoy destination and the solid lines represent the actual flow of data once the anticensorship system is initiated. It should be noted that even though the NH is represented as a single object in this diagram, some anticensorship systems introduce multiple devices to the network.

access; and (4) how to send information from the covert destination to the client. In this section, we review different ways existing anticensorship systems solve these problems. Figure 2 has a summary of this information.

2.1.1 Network Hardware in Existing Anticensorship Systems

There are three different hardware configurations for existing anticensorship systems. The most common configuration is used by Telex and TapDance and involves a device (D) that operates in conjunction with a router in an ISP's network. The router sends every packet it receives to D. D reads every packet and occasionally modifies packets before sending them back to the router. When the router receives packets from D, it forwards them according to normal routing protocols.

Decoy Routing and Rebound introduce their own routers (decoy routers) and proxies (decoy proxies). In Decoy Routing, the decoy routers and decoy proxies may be physically separate or encapsulated on the same machine, but in Rebound the decoy router and decoy proxy functions are always on the same machine. The decoy routers behave like typical routers except that they forward packets to decoy proxies when they detect a tag in a packet from a client that indicates that the client wants to use the anticensorship system. The decoy proxies perform most of the functionality in Decoy Routing and Rebound, such as opening connections to covert destinations and sending data between the client and the covert destination. The decoy routers are placed according to [6].

Cirripede has the most complicated hardware configuration. The system includes a registration server, routers, and proxies. Clients contact the registration server to indicate that they want to use Cirripede. Once the client is registered, the registration server disseminates information about the client to every router and proxy in the system, and when a router sees traffic from a registered client, it forwards the traffic to a proxy, which hijacks the client's connection to the decoy destination and performs the rest of Cirripede's functions.

2.1.2 Methods for Requesting Censorship Circumvention

Telex, TapDance, Decoy Routing and Rebound's clients use the same method to request access to their systems. They all embed a secret tag in the Random Field in the TLS [9] ClientHello message. The exact method for embedding the tag in the Random Field varies from system to system. Cirripede, on the other hand, uses covert channels embedded in TCP [21] SYN packets.

2.1.3 Communicating Covert Destination Address from Client to Anticensorship System

All of the anticensorship systems have their clients send the covert destination's address in an HTTP GET request using HTTP over TLS (HTTPS). Despite their apparent similarity, the systems differ in the way that they obtain the cryptographic keys necessary to read the client's encrypted message. Decoy Routing sidesteps the issue by assuming that the client and the Decoy Routing proxies already share a set of symmetric keys. Telex, Cirripede, and Rebound all use the approach developed in Telex, which uses information in the client's tag to generate symmetric keys for the connection using a variant to the Diffie-Hellman key exchange [10]. TapDance gets the keys it needs by modifying the TLS handshake so that TapDance can compute the secret key used in the connection between the client and the decoy destination.

2.1.4 Relaying Information from the Covert Destination to the Client

Every anticensorship system except Rebound sends data to the client from the covert destination using the same approach. When the anticensorship system receives data from the covert destination to send to the client, the anticensorship system forges a packet from the decoy des-

	Telex	TapDance	Decoy Routing	Rebound	Cirripede	Decoy Switching
Network Hardware	Telex Station (software)	TapDance Station (software)	Decoy Router (software) and Decoy Proxy (software)	Decoy Router (software)	Registration Server (software), Proxy (software), and Router (hardware)	Decoy Switch (hardware) and Local Controller (software)
Tagging Mechanism	TLS ClientHello Random Field	Custom Stenography in Incomplete HTTP Request	TLS ClientHello Random Field	TLS ClientHello Random Field	TCP SYN	TCP SYN
Send Covert Addr.	HTTPS GET	HTTPS GET	HTTPS GET	HTTPS GET	HTTPS GET	HTTP GET
Forward Data	Forge Packet	Forge Packet	Forge Packet	Overwrite Payload	Forge Packet	Forge Packet

Figure 2: Comparison of features between Decoy Switching and existing anticensorship systems.

tination and includes the packet from the covert destination in the payload of the forged packet. In most of these systems, the client and the anticensorship system use a set of encryption keys that are different from the keys used by the anticensorship system and the covert destination, so in those cases, the anticensorship system decrypts the data from the covert destination and reencrypts it using the key it shares with the client.

Rebound's method for sending information from the covert destination to the client is different because it does not forge packets from the decoy destination. Instead, Rebound rewrites packets sent from the client to the decoy destination such that the URL in the client's request contains data from the covert destination. Since the decoy destination is extremely unlikely to have a resource specified by the rewritten URL, the decoy destination responds to the client with an `HTTP 404 Not Found` error, which includes the URL from Rebound with data from the covert destination.

3 Threat Model and Computational Assumptions

3.1 Threat Model

Before diving into Decoy Switching's implementation, we establish the threat model considered when building the system. Decoy Switching is designed with the assumption that there is an ISP-level adversary that blocks all traffic to and from a set of IP addresses. The adversary has access to all traffic within its network, but the adversary uses conventional routers that can only inspect IP source and destination addresses, the IP protocol, and the TCP (or UDP) source and destination ports for real-time analysis. Furthermore, the adversary has no knowledge of any internet traffic outside its network, and while it is capable of dropping any packet within its network, it is not able to make modifications to the packets it does not drop.

3.2 Limitations of Existing Anticensorship Systems

The common problem with existing anticensorship systems is that they are all implemented in software, which limits their ability to scale with the demands of modern networks. If network-level anticensorship systems cannot operate at line rate, or the speed at which packets are transmitted through network links, then ISPs will not deploy them because they add too much latency to the ISP's network [27]. Therefore, the goal of this project is to build an anticensorship system that can be deployed without adding latency to ISPs' networks.

To get a sense for what it means to operate at line rate we look at network switches that are used in networks where network-level anticensorship systems are meant to be deployed. It is common for network switches to operate at a rate in the order of Terabits per second (Tbps). For example, the Cisco Nexus 3016 (1.28 Tbps) [8], the Juniper EX4600 (1.44 Tbps) [14], Arista Network's 7050X Series (2.56 Tbps) [1], and Barefoot Network's Tofino (6.5 Tbps) [2] all operate in the Tbps range. It should be noted that these switches are not the very best switches that are on the market, like the Cisco Nexus 7700 Switch which boasts throughput up to 83 Tbps [7]. With this in mind, we use the Cisco Nexus 3016 switch as a basis for our analysis going forward.

Assuming that a network-level anticensorship system was deployed in place of a Cisco 3016 switch, the system will have to keep up with the network demands at that location. The Cisco 3016 has a maximum throughput of 1.28 Tbps, which according to the switch's specification, amounts to 950 million packets per second (Mpps) [8]. To make our performance estimate more conservative, lets say that the switch receives packets at a rate of 500 Mpps, which is about half of the switch's capacity. At

500 Mpps, then the switch receives, on average, a new packet every two nanoseconds.

Any system processing these packets needs to be able to keep up with this rate, but systems implemented in software execute machine instructions at the rate of the CPU clock. Assuming that a CPU clock cycle takes half a nanosecond (a 2 GHz CPU), then there would only be time for the software system to execute four machine instructions per packet to keep up with incoming traffic, and none of the anticensorship systems described in §2 can be implemented with anywhere near four machine instructions.

4 Decoy Switching Overview

Decoy Switching is designed to provide network-level anticensorship using programmable switch hardware. In this section, we provide high-level information about the system. §5 covers implementation details.

4.1 Anticensorship at Line Rate with Programmable Switch Hardware

Recently, companies have started building switches [2, 17, 18] that take advantage of advances in hardware design [4]. These switches allow users to compile custom switching logic to switching hardware that operates at line rate, and they can be programmed using a new language called P4 [3, 19], which provides greater flexibility than predecessors like OpenFlow [16]. Despite P4's advantages over previous methods for programming network hardware, P4 places significant constraints on the range of supported operations due to limitations of the underlying hardware architecture. The underlying hardware also places severe constraints on the amount of state available for processing packets.

These restrictions make it challenging to build systems in this environment, but if it is possible to design a program that operates within those constraints, then the program can be compiled to hardware and the program gets all the performance benefits associated with hardware. Namely, operating at line rate.

Therefore, this project's goal is building a network-level anticensorship system that takes advantage of emerging switch hardware to scale with the demands of modern internet traffic. The main challenge of this project is designing a system amenable to implementation in the impoverished programming environment that accompanies programmable switch hardware. The remainder of this paper is devoted to explaining and analyzing Decoy Switching, the system designed to meet this goal.

4.2 High-Level System Architecture

The Decoy Switching system is composed of many *decoy switches*, physical devices that behave like typical routers under normal circumstances, that are placed throughout the internet. With this hardware in place, clients can request to use Decoy Switching by opening a TCP connection with a decoy destination and embedding a secret tag in the connection's opening SYN packet. Once the client completes the TCP handshake with the decoy destination, the client sends the name of the covert destination that it wants to reach. If the client's traffic to the decoy destination goes through a decoy switch, then the decoy switch will have seen the client's tag and it will take over the client's connection with the decoy destination as soon as it receives the name of the covert destination from the client.

Once the decoy switch takes over, it does several things at the same time. First, the decoy switch closes the decoy destination's half of the client/decoy destination connection and leaves the client's half of the connection open. Second, the decoy switch opens a connection with the covert destination, and third, the decoy switch uses the client/decoy destination connection to tell the client that it is using Decoy Switching.

Once the client receives the acknowledgment message from the decoy switch, then any information that the client sends on the connection to the decoy destination gets intercepted by the decoy switch and rerouted to the covert destination. Similarly, any information that the decoy switch receives from the covert destination gets forwarded to the client on the old client/decoy destination connection. If the client does not receive the acknowledgment message from the decoy switch, then it closes the connection with the decoy destination and repeats the process with a new decoy destination with the hope that the new path will contain a decoy switch.

4.3 Design Decisions

Decoy Switching's design is based on existing anticensorship systems discussed in §2, but it does not exactly correspond to any of them due to Decoy Switching's goal of performing as much processing as possible in hardware. To provide a better understanding of Decoy Switching's design we explain the way that Decoy Switching addresses the core problems of network-level anticensorship as defined in §2.1. Figure 2 provides a direct comparison of Decoy Switching against existing anticensorship systems.

Required Network Hardware Decoy switches have two parts enclosed within the same device. The first part is a programmable switch that is connected to an ISP's

network and sends/receives packets to/from its neighbors in the network. The second part of the Decoy Switch is a local controller that is a computer that runs specialized software to assist the hardware switch under certain circumstances. The functions performed by the hardware switch and the local controller and the conditions under which the controller is used will be explained in §5.

Requesting Decoy Switching using TCP SYN Several existing anticensorship systems have clients request access to their systems using fields in the TLS handshake. This approach is problematic for Decoy Switching because it is difficult to efficiently detect the start of the TLS handshake in hardware without tracking the state of every connection that goes through the decoy switch and predicting when the TLS ClientHello will be sent. Tracking every connection in switch hardware is not feasible due to limited available state. Therefore, Decoy Switching encodes the client's tag in the TCP SYN packet, the very first packet in the client's connection with the decoy destination. This means that the decoy switch knows whether a connection will use Decoy Switching as soon as possible, so the decoy switch only stores information that it will actually use.

Communicating the Covert Destination's Address Since the decoy switch's programmable hardware does not have enough memory to track the state of clients' connections for long, Decoy Switching's clients send the covert destination's address at the first opportunity, which is the first packet after the TCP handshake completes. The client sends the covert destination's address in an HTTP GET request over TCP. We use TCP instead of TLS because the adversary cannot read the packet payload (§3.1).

Relaying information between the Covert Destination and the Client Decoy Switching sends the client packets from the covert destination without alerting the adversary by hijacking the client's connection with the decoy destination and reusing that connection. This was discussed in §4.2.

5 Implementation Specifics

In this section, we dive into specifics of Decoy Switching's implementation. In doing so, we (1) outline the data structures and functions that compose the decoy switch and the switch's local controller. Then, we (2) go through a detailed trace of a typical use of Decoy Switching to demonstrate how the functions discussed in (1) work together to perform Decoy Switching. Figure 3 provides an overview of the main features in the switch and the local controller.

5.1 Decoy Switch Hardware: Core Functions

The hardware portion of the decoy switch is written in P4 and contains a couple different types of data structures and performs several types of functions. Together the functions and data structures perform most of the Decoy Switching protocol, which makes Decoy Switching faster than its predecessors (§6). In this section we review the main data structures and functions that are performed in the switch's hardware.

5.1.1 Communicating with the Local Controller using Custom RPC

The switch hardware does most of the work in Decoy Switching, but there are some tasks that require the local controller. In those cases, the switch needs to be able to use a remote procedure call (RPC) to specify the function it wants the controller to run as well as the data it would like the controller to operate on.

Decoy Switching uses a custom RPC to communicate between the hardware switch and the local controller. In this protocol, the switch encapsulates data it wants to send to the local controller in a special RPC frame whose header fields include a one-byte reason field and eight bytes of padding. The reason field specifies the function that the switch wants executed and the padding is used to differentiate the RPC frame from the encapsulated packet.

After the switch encapsulates the packet within the RPC frame, it sends it to the local controller by writing the packet to the interface that connects the switch to the controller. That interface could be ethernet, virtual ethernet, PCI bus, etc. When the controller completes the requested function, it responds to the RPC by sending the switch the same packet it received when the RPC was initiated. The controller encapsulates the packet it returns in an RPC frame with the reason field set to zero. See §5.2.1 for more on the way the controller handles RPCs.

5.1.2 Key-Value Storage

In P4 and the switch hardware environment, there are two options for key-value storage: updating the switch's match-action tables to include a new entry or implementing a hash table in hardware using a register file for state and using a checksum for a hash function. The drawback to using a match-action table is that the switch cannot modify its match-action tables by itself. Instead,

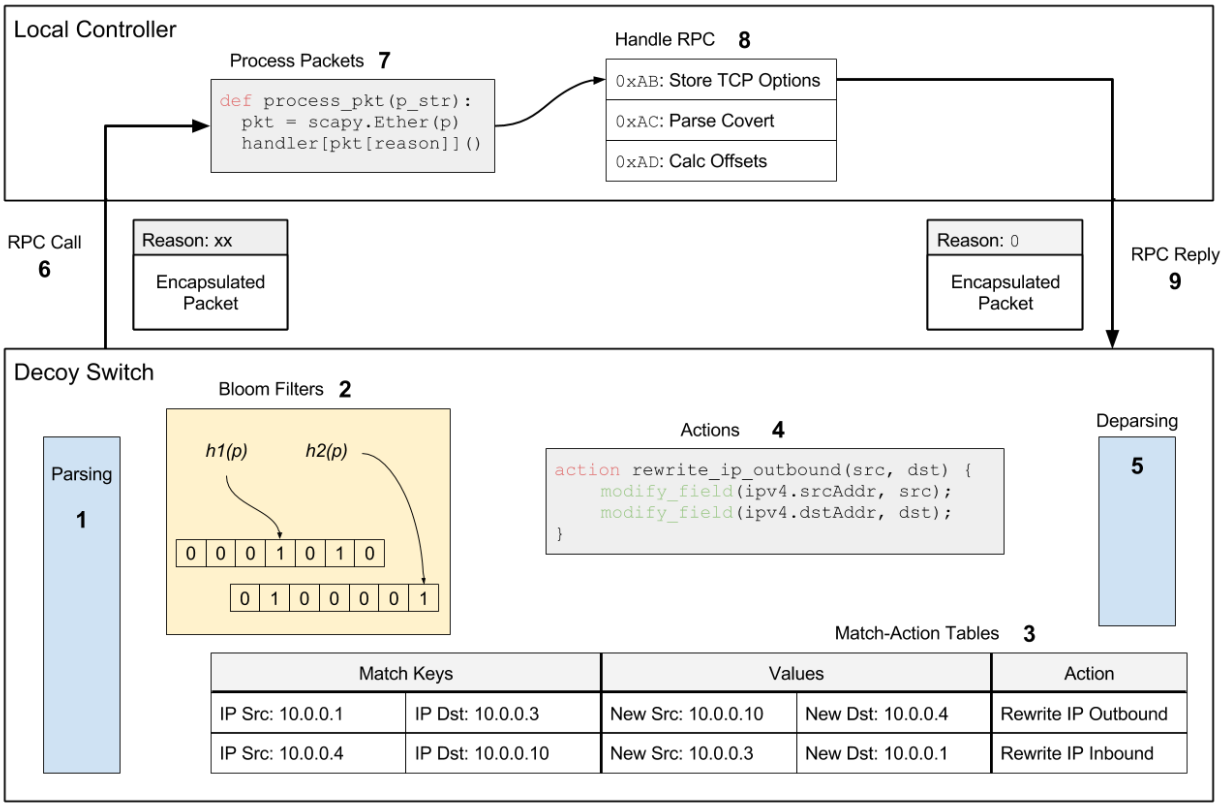

Figure 3: An overview of decoy switch components and the relationship between the switch and the local controller. When a packet (p) first enter the switch, it is parsed and decomposed into header fields that the switch can manipulate (1). Then, the switch checks for p in its bloom filters (2) to determine whether p belongs to a connection that uses Decoy Switching (§5.1.3 and §5.3.1). The switch processes p using a series of match-action tables (3), which associate data and actions with header fields in p (§5.1.2). The actions (4) are used to make modifications to p and update metadata related to p maintained within the switch. Before sending p, the deparser (5) rebuilds p's headers to include the switch's modifications to the packet. Sometimes, the switch uses a custom RPC protocol (§5.1.1) to invoke a function on the local controller (6). The local controller parses packets it receives from the switch (7), uses the `reason` field in the RPC header to determine which function to execute (§5.2.1), and (8) performs the requested function. When the local controller finishes executing an RPC, it notifies the switch (9).

the switch's local controller updates the switch's tables. This means that the switch needs to send what ever data it wants to add to a table to the local controller, which greatly reduces the efficiency of the switch. On the other hand, even though using a hardware register avoids a trip to the controller, the problem with this approach is that there are no hash functions available for indexing into the register, and the checksums that can be used as hash functions are likely to have many collisions [24].

It may be possible to combine the two approaches to make a general key-value store that has the performance benefit of the hardware table in most cases and can also fall back on the match-action tables when there is a collision, but since this method ultimately involves sending data to the controller, we decided to stick to the built-in match-action tables and accept the performance penalty.

5.1.3 Bloom Filters in Hardware

Sometimes we would like to store information to indicate that the decoy switch has seen a particular packet. It is possible to do this using the key-value storage options discussed in the previous section, but to avoid updating match action tables, we take advantage of the fact that we simply want to indicate that a packet has been seen and the fact that we do not have any additional information to store. Therefore, instead of using a key-value store, we implement a bloom filter [25] in hardware using two hardware registers and two different hash functions.

Since programmable switch hardware does not sup-

port arbitrary hash functions, there are not many choices for implementing the two hashes. Most programmable switches support some checksum calculations because routers need to recompute packets' checksums if the router modifies a packet's fields. Therefore, our bloom filter implementation uses two checksum calculations: 16-bit cyclic redundancy check (CRC) checksum [26] and the 16-bit IPv4 header checksum.

To insert an entry into the bloom filter, first compute both checksums using the packet's source and destination IP addresses, the packet's IP protocol field, and the packet's source and destination TCP ports. Second, use the CRC checksum to index into the first register and set the bit at that index to one to indicate that the entry has been seen. Then, use the IPv4 checksum to index into the second register and set the bit at that index to one. To later check whether a packet is part of a tagged connection we simply recompute the two checksums outlined above and check whether the corresponding entries in the two registers are set to one.

5.1.4 "Creating" TCP Control Packets

There are times when we would like the decoy switch to generate TCP control packets to manipulate TCP connections. For example, the decoy switch needs to close the client's connection to the decoy destination and open a new TCP connection with the covert destination.

This may sound easy, but the process is not straightforward because it is not possible to construct arbitrary packets in P4. It is only possible to manipulate existing packets as the switch receives them. Conveniently, it is possible to change any of the static fields that the switch can parse and all the fields in TCP control packets are parsable and have static length. The switch can also copy and reprocess packets. By combining these observations, it is possible to "create" any TCP control packet from any packet that goes through the switch by copying the packet, processing the original packet as normal, and stripping all unnecessary data from the copy and rewriting the copy's TCP and IP fields to turn it into the desired TCP control packet.

5.2 Decoy Switch Local Controller: Core Functions

The local controller runs a Python program that performs the parts of Decoy Switching that cannot be implemented efficiently in hardware. In this section, we look at common methods that the controller uses to execute the functions it supports. We discuss the actual functions that the controller performs for the switch in §5.3.

5.2.1 Receiving Data from the Switch and the Other Side of the RPC

The controller receives data from the switch by capturing packets on the interface that connects it to the switch, and it sends data to the switch by writing data to the same port. The controller uses Scapy [20] to capture and send packets. When the controller receives a packet, it parses the first nine bytes to see if the packet uses the RPC format specified in §5.1.1. Packets that do not use the proper RPC format are ignored, but those that are correctly formatted are stripped of the encapsulating RPC header, parsed using Scapy, and sent to the appropriate function. The controller responds to RPCs by sending the packet back to the switch with a new RPC header encapsulating the packet in which the reason field is set to zero.

5.2.2 Updating Match-Action Tables

Programmable switches typically expose an interface for updating their match-action tables. The virtual switch used to prototype the Decoy Switching system exposed a command-line interface (CLI) for manipulating match-action tables, so the controller uses this CLI to add entries to the switch's match-action tables.

5.3 Exploring the Implementation Through a Typical Interaction

It is time to dive into implementation-specific details by doing a step-by-step trace through a typical use of Decoy Switching. The primary stages of Decoy Switching are (1) tag detection and connection tracking, (2) parsing the covert destination and hijacking the client's connection with the decoy destination, (3) opening a connection with the covert destination, and (4) routing packets between the client and the covert destination. Figure 4 contains an overview of the interactions between the client, decoy switch, decoy destination, and the covert destination in typical use of the Decoy Switching system.

5.3.1 Efficient Tag Detection and Connection Tracking using Bloom Filters

When the client wants to use the Decoy Switching system, it needs make a request to a decoy destination and include a secret tag in the SYN packet that it sends to the decoy destination. The tag needs to be easily computable by the decoy switch and difficult for the adversary to detect. Since the adversary cannot read the TCP sequence number (§3.1), we use the initial sequence number (ISN) to store a 32-bit tag. The tag consists of the 32-bit CRC checksum of the source, destination, and protocol fields in the IP header as well as the source port, destination

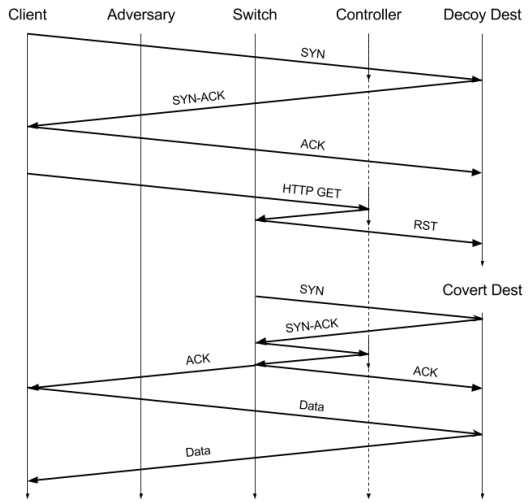

Figure 4: Complete request diagram for a typical use of Decoy Switching. The client initiates a connection with the decoy destination, and once the decoy switch gets the address of the covert destination from the client, the switch closes the connection with the decoy destination and connects to the covert destination on the client's behalf. The adversary thinks that the client is talking to the decoy destination the whole time. The local controller only receives packets from the decoy switch when there is a solid line in the diagram.

port, and window size fields in the TCP header. This method has the advantage of being computable in hardware and appearing random to observers.

Since the tag is in SYN packets, the switch computes what the tag for every SYN packet it sees and compares the result to the packet's sequence number. If the packet's sequence number is equal to the tag computed by the switch, then the switch needs to mark the connection so it can take appropriate action for future packets. The switch uses a bloom filter as described in §5.1.3 to record that a tagged SYN packet has been seen for this connection.

After detecting the tag and marking the connection using the bloom filter, the switch forwards the packet to the destination specified in the packet (this is the decoy destination). We now discuss how the switch keeps track of the state of the client's connection to the decoy destination.

Once the switch records that a client wants to use Decoy Switching, the switch needs to get the name of the covert destination that the client wants to connect to. As discussed in §4.3, the client sends the name of the covert destination in the first packet after the client completes the TCP handshake with the decoy destination. Therefore, the decoy switch needs to track the state of the client's connection with the decoy destination so it can detect the end of the TCP handshake.

In a normal TCP handshake, the client sends a SYN to the destination, the destination responds with a SYN-ACK back to the client, and then, the client sends an ACK to the destination to complete the handshake. At this point, the decoy switch has already seen the client's SYN, so it assumes that the next ACK the client sends to the decoy destination is the ACK that completes the TCP handshake.

We are again faced with the question of how the decoy switch should record whether an event has occurred. Here, the event is whether the decoy switch has seen an ACK from a client who previously sent a tagged SYN. To solve this problem, we introduce a second bloom filter that is updated when the decoy switch sees an ACK from a client in a tagged connection.

Conceptually the logic for tracking the handshake works as follows:

1. For every packet, check whether it is present in the SYN bloom filter. If it is not present, it is a normal packet and there is nothing to do beyond regular routing.

2. If the packet is present in the SYN bloom filter, check whether the packet is present in the ACK bloom filter. If the packet is also in the ACK bloom filter, then it is safe to conclude that the TCP handshake between the client and the decoy destination is complete.

3. If the current packet is an ACK and it is not present in the ACK bloom filter then this packet must be the ACK that concludes the TCP handshake, so the switch updates the ACK bloom filter and forwards the packet on to the decoy destination.

Once the TCP handshake is complete, the next packet sent by the client should contain the address of the covert destination. At that point, the switch has more work to do to perform Decoy Switching. That work is outlined in the next section.

5.3.2 Parsing Covert Destination and Connection Hijacking

As mentioned in §4.3, the client sends the name of covert destination it wants to access as an absolute URI in the request line of an HTTP GET request. When the client sends the covert destination to the decoy switch, it sends the covert destination's domain name rather than its IP address because the client cannot look up the covert destination's IP address without revealing its intention to access the address to the adversary. When the decoy switch receives the packet with the covert destination from the

client, it needs to (1) parse the covert destination, (2) look up the covert destination's IP address, (3) save the mapping between the client's address, the client's decoy destination, and the client's covert destination, (4) close the connection to the decoy destination, and (5) open a TCP connection to the covert destination before (6) responding to the client. In this section we discuss the implementation of steps 1-4. Steps 5 and 6 are addressed in §5.3.3.

In considering 1, parsing the covert destination from the client's request, we must consider the parsing limitations of the switch hardware. While it is possible to parse user-defined packet headers in the switch hardware, the headers must be of fixed length. Unfortunately, this means that the covert destination cannot be parsed in hardware because the length of the covert destination's domain name can vary. Therefore, when the switch receives the covert destination from the client, it needs to have the local controller parse the name of the covert destination. We will cover the role of the controller in more detail later in this section. For now, we continue to review the aspects of connection hijacking that are performed in the switch.

Moving on to 2, looking up the covert destination's IP address, it is not possible to perform a DNS lookup in the switch hardware for the same reason that it is not possible to parse the name of the covert destination: variable length fields. In DNS, the variable length comes from the NAME field in the DNS resource record. Fortunately, the switch already needs to send the packet to the controller to parse the name of the covert destination, so the controller can look up the covert destination's IP address while it has the packet to process.

We now consider how to accomplish 3, saving the covert destination and other necessary information for rerouting the client's traffic so it goes to the covert destination instead. Ideally, we would like to have the switch use a packet from the client addressed to the decoy destination to look up the address of the covert destination. This is exactly what the switch's match-action tables are meant to do. Normally, there would be a penalty for adding an entry to the switch's match action table, but there is no extra penalty in this case because the switch already needs to send this packet to the controller.

At this point, we have outlined three things that the controller needs to do: parsing the covert destination, looking up the covert IP address, and updating the switch's match-action tables. These three tasks are encapsulated in a controller function that can be called with a single RPC. To implement this function, the controller uses Scapy to parse the packet headers and isolate the packet payload, which should be an HTTP request line that looks something like `GET http://www.example.com:80 HTTP/1.1`, the URL can be parsed using Python string functions as well as the standard Python URL parsing library, `urlparse` [23]. Once the controller isolates the covert destination's domain name, it looks up the covert IP address using the Python `socket` library [22].

Now that the controller has the covert destination's IP address, it updates the appropriate match-action table in the switch with two new entries. One entry is keyed on the IP addresses and TCP ports of the client and the decoy destination, and the other is keys on the IP addresses and TCP ports of the decoy switch and the covert destination. These keys are used to determine whether a packet is coming from the client to the covert destination or if a packet is from the covert destination to the client. For more on the way these two entries are used, see the section on typical forwarding (§5.3.4). Now that we have covered the way that the switch and the controller work together to accomplish the first three tasks outlined at the start of this section, we finally discuss part 4, closing the connection to the decoy destination.

Closing a TCP connection means sending an RST packet to one of the participants. Here, we want to send an RST packet to the decoy destination so the decoy destination stops using its connection with the client. The switch makes a copy of the next packet it receives and applies the method in §5.1.4 to generate the RST packet for the decoy destination. The P4 code for this operation is available in figure **??**.

5.3.3 Performing TCP Handshake with TCP Options Approximation

At this point, the connection with the decoy destination has been closed and the decoy switch knows the IP address of the covert destination, so it is time to open a TCP connection to the covert destination. To begin, the decoy switch has to send a SYN packet to the covert destination to initiate the TCP handshake. In principle, it is possible to make a SYN packet to send to the covert destination using the method from §5.1.4, but the problem with this approach is that some hosts do not respond to SYN packets if they do not contain the correct TCP options, which are things like selective acknowledgments and window segment size that are used to improve TCP performance. It is not clear how to parse TCP options using P4 and programmable switch hardware because the TCP options may form a loop in the P4 parse graph, which leads to undefined behavior. One solution is to enforce a strict ordering to the TCP options as in [12], but this solution depends on the client and the covert destination supporting these options and obeying this ordering, which seems unlikely to happen.

The TCP option fields lead to deeper problems for Decoy Switching beyond parsing in P4. Since TCP op-

tions are used for the duration of a TCP connection, the connection that the decoy switch makes with the covert destination must either use the same TCP options as the client/decoy destination connection or the decoy switch must be able to keep track of the TCP options used in the client/decoy destination connection as well as the TCP options used in the switch/covert destination connection and be able to map between the two. The mapping approach becomes complicated quickly due to the window scale and timestamp options, which are used to determine how much data the TCP endpoint can receive and whether sequence number overflow has occurred, respectively.

Given that remapping TCP options is complicated and requires a lot of per-connection state on the switch, it seems like getting the covert destination to use the same TCP options as the client/decoy destination connection is the way to go, but this approach will not work in general because TCP options are optional. That is, the covert destination may not support some of the options used in the client/decoy destination connection.

For now, decoy switching addresses these issues by approximating the correct TCP options using the assumption that all internet hosts respond to SYN packet TCP options in the same way. When the decoy switch receives a tagged SYN packet from the client, it makes a copy of the packet, processes the original packet as described in §5.3.1, and sends the copy to the local controller in an RPC. When the controller gets the SYN packet, it saves the client's TCP options. Later, when the client sends the covert destination's address to the decoy switch and the switch sends the packet to the local controller for parsing (§5.3.2), the controller uses the client's original TCP options to construct a SYN packet for the covert destination. The SYN packet's source IP address is the switch's address so the covert destination's response comes back to the switch. The controller sends the new SYN packet back to the switch, which then forwards the packet to the covert destination to initiate the TCP handshake.

If all goes well and the covert destination responds to the SYN packet with a SYN-ACK packet, the switch needs to respond with an ACK to complete the TCP handshake. To do so, the switch converts the SYN-ACK packet into an ACK packet for the covert destination using the method outlined in §5.1.4 and sends the new ACK back to the covert destination.

The covert destination's SYN-ACK contains the initial sequence number that the covert destination will use as a base for all future packets it sends. The switch needs to store the new sequence number so the switch can rewrite sequence numbers from the covert destination so they match the sequence numbers from the decoy destination which the client expects to see. The switch handles this at the same time as it makes the ACK for the covert destination by making a copy of the SYN-ACK when it first arrives and by sending the SYN-ACK to the local controller in an RPC so the controller can update the switch's match-action tables.

When the controller receives the SYN-ACK packet from the switch, it (1) computes the difference between the SYN-ACK's sequence (seq) number and the acknowledgment (ack) number in the last packet the client sent to the decoy destination. The controller also (2) computes the difference between the SYN-ACK's ack number and the seq number in the last packet the client sent. The first difference is used to update the `covert/decoy table`, which is used to rewrite seq numbers in packets from the covert destination so they look like they came from the decoy destination. The `covert/decoy table` is also used to rewrite ack numbers in packets from the client so they match the ack number the covert destination expects. The second difference is used updated the `client/switch table`, which does the same things as the `covert/decoy table` except it is applied to packets going in the opposite direction.

5.3.4 Typical Forwarding and Minimizing Software Processing

At this point, there is a connection between the client and the decoy switch and a connection between the decoy switch and the covert destination, so the decoy switch is ready to perform typical forwarding between the client and the covert destination. The decoy switch needs to tell the client that it can start sending data to the covert destination. The switch repurposes the SYN-ACK packet that it gets back from the controller to make an ACK packet for the client using the method from §5.1.4. It should be noted that this packet looks benign to the adversary because the switch uses the `client/switch table` and `covert/decoy table` to set the new ACK packet's seq and ack numbers so it looks like the packet is a TCP keepalive packet from the decoy destination.

As soon as the client receives the ACK from the decoy switch, the client assumes that any data it sends over its connection to the decoy destination will reach the covert destination. To provide this functionality, the decoy switch uses the match-action table mentioned in §5.3.2 that maps client/decoy destination addresses to the covert destination and vice versa. We call this table the `mapping table`. The decoy switch uses the `mapping table` to perform typical forwarding as follows:

1. Every time the decoy switch receives a packet, it consults the bloom filters from §5.3.1 to determine whether the packet is part of a connection that uses

Decoy Switching and whether the packet has completed the TCP handshake with the decoy destination.

2. If both of the previous conditions are met, then the decoy switch looks for the packet in the `mapping table`. If there is no entry for the packet, then that connection is not ready for typical forwarding, but if the packet is present, then the table returns new IP source and destination addresses and new TCP source and destination ports for the packet. The new addresses and ports are saved for later.

3. The switch looks up the seq and ack number differences for the connection from the `client/switch` and `covert/decoy` tables, and updates the packet's seq and ack numbers as in figure **??**.

4. Once the seq and ack numbers are set, the switch updates the packet's TCP ports and IP addresses with the results from the `mapping table`. This cannot be done earlier because the packet's original addresses and ports are needed to index into the `client/switch` and `covert/decoy` tables.

Now, the packet can be processed as a normal packet and forwarded to the client or the covert destination. Note that none of these steps use the local controller, so we get all the benefits of operating in hardware, which is important for the evaluation in the next section.

6 Evaluation

In §3.2, we explained that a problem with existing anticensorship systems is that they spend too much time processing in software, which prevents them from scaling effectively. In some systems [15, 28], every packet that goes through the system needs to be processed in software regardless of whether the packets are part of a connection using the anticensorship system. Cirripede [13] reduces the number of packets that need to be processed in software because it only uses software to process SYN packets and packets that actively use Cirripede. In this section, we consider Decoy Switching's performance improvement relative to Cirripede, since Cirripede spends less time processing in software than any other network-level anticensorship systems.

We begin by analyzing the number of packets that Cirripede must process every second. In §3.2, we established that network devices need to process about 500 million packets per second (Mpps). Using the April 2016 data from CAIDA [5], we calculate that the average internet connection is 37 packets long. Assuming that all of these flows use TCP, then this means that, on average, one in every 37 packets is a SYN packet that Cirripede must process. Therefore, Cirripede processes about 10 Mpps in software.

In addition to the 10 Mpps that Cirripede must process to detect connections from clients, Cirripede also processes every packet in every flow that uses the system. Assuming that one packet in every ten thousand is part of a flow using Cirripede, then there are about 50 Kpps that Cirripede processes, which means that in total, Cirripede must process about 10.05 Mpps.

In contrast, Decoy Switching needs to use software processing for only three packets in every flow that actually uses Decoy Switching (first SYN packet to save TCP options, packet containing covert destination, and SYN-ACK from covert destination). If we assume that decoy switches see 500 Mpps and that 0.01% of flows use Decoy Switching, as we did for Cirripede, then there are about 3 Kpps that Decoy Switching needs to process in software (three packets for one out of every ten thousand new TCP flows). This means that Decoy Switching provides more than 3,000 times the performance of existing network-level anticensorship techniques. Figure 5 shows the performance benefits of using Decoy Switching over Cirripede as we vary our assumption about the percent of internet traffic that uses a network-level anticensorship system.

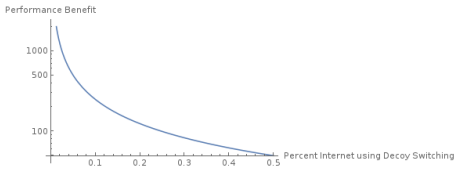

Figure 5: Plot of the relationship between the percent of internet traffic that uses Cirripede/Decoy Switching to the performance benefit of using Decoy Switching over Cirripede. We see that Decoy Switching provides less of an improvement over Cirripede as more traffic uses Cirripede/Decoy Switching.

While these calculations are not exact, they do provide an order of magnitude approximation of the performance benefits that Decoy Switching provides over existing network-level anticensorship solutions, and these results suggest that Decoy Switching has the potential to scale in ways that existing software solutions cannot.

7 Conclusion

Internet service providers and repressive governments frequently censor internet content, which limits personal expression and access to information. To combat this problem, researchers have developed censorship circumvention techniques that make anticensorship a network function. That is, the proposed systems are features of

the internet in a way that is impossible to for censors to control without blocking the entire internet. The anticensorship systems that have been proposed are promising but they are unlikely to be deployed in practice because there is too much internet traffic for them to process.

To address the deployment issues of existing anticensorship systems, we created Decoy Switching, a network-level anticensorship system that can scale with modern internet traffic, using a novel application of programmable switch hardware. This paper reviewed our implementation of Decoy Switching and showed that Decoy Switching has the potential to be thousands of times faster than existing network-level anticensorship systems.

Despite Decoy Switching's performance potential, this paper is only an initial investigation into programmable switch hardware's capacity for building censorship circumvention systems, and there is more work to do before Decoy Switching is ready for deployment. We hope that Decoy Switching is a useful basis for future work in this area.

8 Acknowledgments

I am extremely thankful for Jennifer Rexford's advising. She helped me through every step of this project including assistance choosing a topic, insightful feedback in weekly meetings, thoughtful comments on this paper, and many other contributions. I also appreciate Rob Harrison's debugging assistance and his guidance in using Mininet and P4.

References

[1] Arista 7050x series. Arista Networks, Inc. Available: https://www.arista.com/en/products/7050x-series

[2] The world's fastest and most programmable networks. Barefoot Networks, Inc. Available: https://www.barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," in *ACM Conference on Computer and Communications Security*, vol. 44, July 2014.

[4] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, *Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN*, 2013, pp. 99–110.

[5] Trace statistics for caida passive oc48 and oc192 traces. CAIDA. Available: http://www.caida.org/data/passive/trace_stats/

[6] J. Cesareo, J. Karlin, M. Schapira, and J. Rexford, "Optimizing the placement of implicit proxies," available FTP: www.cs.princeton.edu Directory: /jrex/papers/ File: decoy-routing.pdf.

[7] Cisco nexius 7700 18-slot switch. Cisco Systems, Inc. Available: http://www.cisco.com/c/en/us/products/switches/nexus-7700-18-slot-switch/index.html

[8] Cisco nexus 3016 switch. Cisco Systems, Inc. Available: http://www.cisco.com/c/en/us/products/switches/nexus-3016-switch/index.html

[9] T. Dierks and E. Rescorla, "Transport layer security (tls) protocol version 1.2," *RFC 5246 (Proposed Standard)*, August 2008. Available: https://tools.ietf.org/html/rfc5246

[10] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, pp. 644–654, Sep. 2006. Available: http://dx.doi.org/10.1109/TIT.1976.1055638

[11] D. Ellard, C. Jones, V. Manfredi, W. T. Strayer, B. Thapa, M. V. Welie, and A. Jackson, "Rebound: Decoy routing on asymmetric routes via error messages," in *2015 IEEE 40th Conference on Local Computer Networks (LCN)*, October 2015, pp. 91–99.

[12] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of TCP," *CoRR*, vol. abs/1611.01529, 2016. Available: http://arxiv.org/abs/1611.01529

[13] A. Houmansadr, G. T. K. Nguyen, M. Caesar, and N. Borisov, "Cirripede: Circumvention infrastructure using router redirection with plausible deniability," in *ACM Conference on Computer and Communications Security*, October 2011.

[14] Ex4600. Juniper Networks, Inc. Available: http://www.juniper.net/us/en/products-services/switching/ex-series/ex4600/

[15] J. Karlin, D. Ellard, A. W. Jackson, C. E. Jones, G. Lauer, D. P. Mankins, and T. W. Strayer, "Decoy routing: Towards unblockable internet communication," in *USENIX Workshop on Free and Open Communications on the Internet*, August 2011.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. Available: http://doi.acm.org/10.1145/1355734.1355746

[17] Dpdk networking acceleration. Netronome Systems. Available: https://www.netronome.com/solutions/service-node-applications/dpdk-networking-acceleration/

[18] R. Ozdag. Intel ethernet switch fm6000 series - software defined networking. Intel Corporation. Available: http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf

[19] P4 website. P4 Language Consortium. Available: http://p4.org/

[20] Scapy. Philippe Biondi and the Scapy community. Available: http://www.secdev.org/projects/scapy/

[21] J. Postel, "Transmission control protocol," *RFC 793*, September 1981. Available: https://tools.ietf.org/html/rfc793

[22] socket – low-level networking interface. The Python Software Foundation. Available: https://docs.python.org/2/library/socket.html

[23] urlparse – parse urls into components. The Python Software Foundation. Available: https://docs.python.org/2/library/urlparse.html

[24] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and crc's over real data," *IEEE/ACM Trans. Netw.*, vol. 6, no. 5, pp. 529–543, Oct. 1998. Available: http://dx.doi.org/10.1109/90.731187

[25] Bloom filter. Wikipedia. Available: https://en.wikipedia.org/wiki/Bloom_filter

[26] Cyclic redundancy check. Wikipedia. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check

[27] E. Wustrow, C. Swanson, and J. A. Halderman, "Tap-dance: End-to-middle anticensorship without flow blocking," in *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[28] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman, "Telex: Anticensorship in network infrastructure," in *Proceedings of the 20th USENIX Security Symposium*, August 2011.