Switzerland Design

Peter Eckersley (pde@eff.org)

Version 0.3, May 2008

Motivation

The Comcast affair has demonstrated that the threat of interference by ISPs in user traffic is not just theoretical.[1] Debate will continue about what if anything should be done about such intervention, but in any case it is important for the technical community and the broader public to be well informed about ISPs' treatment of their packets.

In 2007, EFF contributed to ISP transparency efforts by producing a guide for users wishing to perform this kind of testing manually,[2] and by publishing some simple code called `pcapdiff` for spotting subtle differences between packet sniffer logs at either end of a connection.[3]

Unfortunately, the process of collecting synchronized packet traces and comparing them, with `pcapdiff` or by other means, is labor intensive and requires a moderate to high level of care and technical expertise.

This document discusses the design for a project, codenamed *Switzerland*, to address these limitations. We have been working on Switzerland through 2008 and are ready to release an alpha version of the code. The design has client and server components; the use of the word "Switzerland" as a location will refer to the server, while the clients will be nicknamed Alice, Bob, etc.

Design Objectives

Several goals may be considered when designing more complex, automated tools for testing network non-discrimination. These include objectives that increase the coverage of ISP behaviour by testers:

* reducing the level of technical sophistication required to run tests for ISP interference;
* making the process of testing faster;
* increasing the validity of test results by eliminating or detecting likely sources of operator error (such as a misconfigured firewall);
* enabling broader tests (eg, those involving more than two participating IP addresses);

Other objectives improve the usefulness of the data obtained in tests:

* aggregating the community's test results in one or a small number of databases, which leads to

1 See http://www.eff.org/wp/packet-forgery-isps-report-comcast-affair ; http://www.eff.org/testyourisp/ .
2 http://www.eff.org/wp/detecting-packet-injection
3 http://www.eff.org/testyourisp/pcapdiff/

several flow-on benefits:
  - the ability to quickly recognize and confirm interference on networks that investigators may not have access to, or using software and protocols they may not have code for;
  - the ability to run statistical tests;
* ensuring the synchronization of test circumstances, such as the start and end times for packet logs;
* finding ways to test for interference which is more subtle than forged or modified packets (that is, measuring differences in quality of service – such as latency and dropped packet rates – for different applications).

Lastly, we may wish to raise public awareness of problematic ISP activities by allowing individuals to readily see when their providers interfere with their traffic.

This Switzerland design document sets out a protocol which we believe is suited to meeting all of the above design objectives well, although some of them may require further components before they are fully realized. For instance, a basic implementation of the protocol will reduce the barriers to ISP testing, but a good user interface would reduce it further (and we haven't specified one yet). Similarly, the basic protocol will allow the collection of datasets that are useful for detecting latency or bandwidth related discrimination, but appropriate statistical tools should be used to mine those datasets (and we haven't specified any yet).

Where Does Switzerland Fit in the ISP Testing Design Space?

There several design dimensions along which ISP testing systems can vary. These include:

- Actively send "synthetic", pre-determined test traffic, or passively observe the way the network treats natural traffic;
  - If the traffic is synthetic, the testing software may try to cope with the complex variation in operating systems and network environments, or try to simplify things by creating or insisting on a known test environment;
  - Passive testing systems may focus on one or a small number of protocols, or they may try to test for interference in any protocol that is present;
- The software may (1) be unilateral, just trying to detect interference or delay by examining what's happening on a single computer's network connection, or (2) be multi-party, synchronizing and comparing records from computers that are talking to each other, or (3) be in between, only having authoritative records from one end but possessing special knowledge about how the other end will behave;
  - Non-unilateral testing systems may rely on a central server, or they may just try to coordinate records in a peer-to-peer fashion;
- Software may operate at the packet level, measuring integrity, latency and reliability on a per-packet basis, or it may operate at a higher level, confirming (for example) that web pages arrive intact or that a link is running at a certain speed, without worrying about any of the individual packets.

It's a good idea for the Internet community to be pursuing most of these different possibilities, because they're all useful in different situations, and we don't yet know which techniques will prove to be the most important.

We are aware, for instance of efforts by NNSquad and a research team which are initially focusing on unilateral testing, and of the Glasnost project at the Max Plank Institute for Software Systems, which we understand performs tests that are "in between" bilateral and unilateral tests, using synthetic BitTorrent traffic.

Switzerland's first architectural objective is different to these systems: we are doing full multi-party passive monitoring, looking for forged packets in any protocol as well as unusual dropped packet, latency and bandwidth statistics. Our choice of passive monitoring costs us a little in terms of simplicity of deployment (users have to actually be exchanging genuine traffic of some sort between their machine in order to peform any tests), but offers benefits in terms of generality: Switzerland will detect subtle interference in any IP flows, regardless of protocol, and offers a good infrastructure to collect latency datasets.

Some Design Questions

This section sets out the thinking that lead us to our current design proposal.

We begin with some simple assumptions about the discrimination testing problem: the software will record some IP communications between Alice and Bob, and will use a secure mechanism (the Switzerland server) to determine whether there are unexpected inconsistencies between their respective observations of the exchange.

We need to decide who Alice and Bob are; under what circumstances their communications will be recorded; what the mechanism looks like.

Who are Alice and Bob?

In a general purpose system, there should be no restrictions on who Alice and Bob might be. Rather, the design question is what role the Switzerland code will play in matchmaking to put them together.

One option is to rely on users to find each other by independent means. In that case, Alice and Bob (and perhaps Cindy and Daphne) contact Switzerland and ask it to monitor communications amongst them. It does so. If Eric and Fred have made a similar request to Switzerland, and Alice and Eric are exchanging TCP packets, and an intermediary injects RSTs into their connections, Switzerland will not have the ability to observe or report this fact.

Another option is to have each of these parties contact Switzerland independently, and effectively report all of the IP addresses they are communicating with[4] and for Switzerland to notice when any pair of them is communicating. This way, if Alice and Bob are exchanging data over Gnutella, but each has no idea that the other is running a Switzerland client, their link can still be tested. We are disinclined to implement such an aggressive matchmaking policy, since it is basically impossible to do so without

4 Alice, Bob & co do not literally need to provide a list of IP addresses; they could provide hashes; the hashes could be salted with the hour in UTC. This could make it expensive for Switzerland to learn the entire list of communication partners for each of Alice, Bob & Co, while still allowing easy testing for whether Alice is talking to Bob.

leaking a great deal of information about who is communicating with whom.[5]

We plan to implement named and password-protected sessions on the Switzerland server. Alice will only sniff packets amongst IPs participating in a given session. We nickname such a multi-party packet comparison session a measurement "circle". When Switzerland is used as a manual network diagnostic tool, the users will share their session names and passwords themselves. In other more automated kinds of deployments, session names and passwords may be transparently chosen by the client software.

One important use case for Switzerland is when Bob is a server run by a centralized party (possibly the same party as the Switzerland server, though it is necessary that Bob and Switzerland be separate IP addresses) running similar tests for a great many Alices. The design supports this kind of automation of Bob's role with very little specific engineering.

How do Alice and Bob join and leave a circle?

Handling the synchronization of the beginning and end of packet capture windows would be conceptually easier if there were two phases to the life of a measurement circle: matchmaking and testing. Clients could all join during matchmaking, synchronize clocks, start their packet captures at the same time, wait for a period that ensures any packets that were in transit during the setup phase are not mistaken for forgeries, run their tests, and then end their packet captures at the same time with a similar closing window so that packets in transit are not thought to have been dropped.

This simpler-to-engineer arrangement seems to have some drawbacks. For one thing, it makes the user experience more awkward. For another, it does not avoid the more complicated synchronization concerns that apply when, for instance, Bob drops out during the life of the circle.

For this reason we intend to allow users to join and leave the circle as they wish, and to handle the cases of packets near the start and end of their participation with care.

How does Switzerland detect interference between Alice and Bob?

The Solution We Did Pick

Switzerland orchestrates a key k for each Alice-Bob pair (we will use /dev/urandom and other OS equivalents). Alice and Bob calculate a keyed SHA-1 hash $h = \mathrm{HMAC}_k(p)$ where p is the packet with all reasonably variant bits masked out. Alice and Bob send the first n bits of h to Switzerland for each packet. We expect to set $n = 24$ or so. These mini-hashes are sent in batches, along with a timestamp for the newest packet in the batch. We handle multiple packets with the same mini-hash gracefully.

Switzerland calls a packet from Alice to Bob a forgery if Bob reports receiving a mini-hash h in a batch timestamped with time t and Alice has reported sending packets in this flow with a timestamp $t' > t + \delta$, but has not reported sending h. δ should be greater than the maximum possible error between Alice and Bob clocks, minus the minimum possible latency between Alice and Bob.

---

5 In particular, this architecture will always allow malicious parties to test if particular machines are running Switzerland client. It will also always allow the Switzerland server operator to easily test any theories about third parties Alice might be communicating with.

Switzerland calls a packet from Alice to Bob “dropped” if Alice reported sending it in a batch timestamped t and Bob has reported receiving packets in a batch with timestamp $t' > t + \Delta$, where Δ is as large as the protocols resonably allow (should this be 120s? 60s? 30S?).

Because Eve (who wishes to modify packets without detection) does not know k, she only has a 2^{-n} chance of spoofing or editing a packet without detection.[6]

Note that the Switzerland server should assign one key to every pair of participants in a circle, that is, $\frac{1}{2}N(N-1)$ bitmasks in total for a circle of N. This ensures that Eve cannot learn Alice and Bob's key even if the credentials for their circle are public. Eve can of course fool Switzerland about *her* communications with Alice and Bob, but this problem is fundamental: participants in any network discrimination testing system can always lie about the packets they are sending and receiving. The $\frac{1}{2}N(N-1)$ overhead may become too large if the circles grow large. In that case we will devise a protocol for assigning keys only to those pairs of nodes that are actually exchanging packets.

Solutions we didn't pick

The brute-force algorithm for detecting interference between Alice and Bob is for each of them to send a copy of all sent and received packets to Switzerland for reconciliation. In the case where Alice and Bob send equal amounts of data to each other, this would approximately triple their outgoing traffic volumes,[7] and may therefore be deemed impractical.

A substantial saving can be had by sending a cryptographically secure hash of the invariant portions of each packet,[8] rather than the whole datastream. A SHA-1 sum of each packet could theoretically reduce the traffic stream overhead towards 20 / 1460 = 13.5%.[9] After this saving, running a Switzerland client would increase outgoing traffic volumes by approximately 27% (again, assuming symmetrical traffic between Alice and Bob). This is more acceptable than a 200% overhead, but we would still like to do better.[10]

One trick to further reduce the monitoring overhead would be to send hashes of groups of packets, with backtracking to identify which packet(s) were mismatched when some were dropped or forged.[11] The question with this arrangement is how Alice and Bob can simultaneously know *which* packets they should hash together, and in what manner they should be combined as input to the hash function. Most protocols, including TCP, ICMP, IPSec and some of the protocols layered over UDP, have sequence numbers which could provide a reliable means for such spontaneous co-ordination between Alice and Bob.[12] A good implementation of this solution, however, would require a fair degree of engineering:

* sequence detection or sequence definition code for numerous UDP protocols (and a workaround

---

6 Eve can use traffic analysis against the encrypted Switzlerand protocol as an oracle to detect when she has correctly fooled Switzerland with a tampered packet, but this is not particularly useful: even if she could easily break the weakened hash function, her attempts to search the bitmask space would be very visible to Alice, Bob and Switzerland.

7 More precisely, Alice's outgoing traffic is increased by the sum of her incoming and outgoing traffic.

8 Certain fields within captured packets may be expected to change. All of the link-layer fields (eg, Ethernet fields) should be different at each end. Some IP fields may change too, such as the time-to-live (TTL) IP field or, in cases where TCP checksum offloading to the network device hardware cannot be disabled.

9 A SHA-1 sum is 160 bits or 20 bytes; the maximum payload of a TCP/IP packet is 1460 bytes.

10 In particular, because most residential broadband connections are asymmetrical, adding 13.5% of the downstream data rate to the upstream load can be a very significant burden in cases when the other end is faster.

11 The might also incorporate some error-detection features; data structures such as hash trees may tell Switzerland a subset of packets that caused the hash to not match.

12 Intuitively, one may search for other methods of grouping and ordering packets that are not based on explicit sequence

for UDP protocols that lack sequence numbers);

* a means of dealing with unknown or unforeseen session-layer protocols (SCTP or other more obscure protocols for which support has not been implemented), or unorthodox or incorrect TCP or UDP implementations on a host that are nonetheless not the ISP's fault;
* an efficient way for working around dropped packets and retransmission in TCP;
* a way of deciding how many packets to combine, and how to deal with partial packet sets when transmissions slow or cease,[13] possibly depending on the packet transmission rate;
* a choice of an appropriate hash tree or similar algorithm for deciding how to combine the set of packets;

It may be that the payoff for engineering this solution is positive at some point, but for the time being our solution keeps Switzerland's traffic overhead low and is fairly simple to implement.random

What does Switzerland log? When does it report discrimination?

We wish to ensure that Switzerland logs sufficient information to constitute solid evidence in cases of interference, without aggregating records on the server at an impractically high rate.

The first important piece of metadata that Alice and Bob send to Switzerland is metadata about the *flows* that are active between them.[14] This occurs regardless of any interference. In addition to saying when streams start and end, Alice will periodically report how many packets are being sent in each stream, along with their mean and median sizes, *and latency related data*.[15]

In cases where Switzerland detects probable interference (ie, forged packets) between Alice and Bob, the server will ask them both for actual copies of the packets that were interfered with (using a chain of protocol messages called `forged-in`, `fi-context`, `forged-out`, `fo-contex`, and `forged-details`, documented belw. They should in fact send a range of the packets sent before and after interference, both in time and within the stream that was interfered with.

Unlike forged/modified packets, dropped or heavily lagged packets are not in and of themselves evidence of interference. Statistical analysis will neverthless be possible to say, for instance, whether some kinds of flows have disproportionately high numbers of dropped packets or unusually high latencies. In all likelihood, this kind of analysis will have to be conducted offline, using the server's records, since the tests are statistical.

One last possibility for future research is that the Switzerland server may be able to keep track of the kinds of interference it has seen before, such as "TCP RST forgery in BitTorrent sessions", "modification of HTTP status responses", etc. When entirely novel forms of interference are detected, Switzerland may decide to request a full copy of the session log from Alice and Bob, to maximize the

---

numbers, but we have not thought of any that work well in the presence of significantly delayed and reordered packets.

13 This case is particularly tricky because Alice and Bob need to make the same decisions about which packets to hash together.

14 *Flows* are logically connected sequences of packets. For instance, each TCP connection is a flow, as is any series of UDP packets between the same hosts and ports. Port scans constitute the most expensive case for this kind of record keeping, since every packet sent in one is its own flow; we may at some point wish to find a way to compress them efficiently.

15 Alice and Bob each report average transmit/receive times for each flow reporting period. In the absence of dropped and retansmitted packets, these allow the calculation of average latency. We still need a good definition of average latency in the presence of drops and retransmissions!

amount of forensic information available. It might or might not be a matter of client configuration as to whether Alice keeps these records during the session and sends them to Switzerland upon request.

How is Switzerland deployed?

Switzerland is intended to support several modes of deployment.

The current development version of Switzerland uses a universal circle; all nodes will report traffic to and from all other nodes.

The simplest mode we plan to retain in the long term is for researchers to create a circle, and to run some protocol(s) of interest between the members of the circle. We believe this will be useful both to application and protocol developers as well as to parties investigating ISP conduct in general. Standard public circles for testing specific protocols could easily be deployed and co-ordinated over a wiki ("here is a circle for testing skype: join it if you want to help out; here is a circle for testing HTTP that contains both servers and machines configured to fetch a test page from your HTTP server"; here is a circle for testing Lotus Notes; etc).

A second mode of deployment is to include automated test scripts that send synthetic traffic to a Bob server. The test scripts may be designed around known patterns of ISP interference. These may be run as a diagnostic tool by a much wider population of users, who wish to know if their ISP is engaged in any discrimination. The user interface can tell the user as it performs each test, and whether her ISP has passed or failed.

A third and more ambitious mode of deployment is to roll out a network of Alices functioning as a giant laboratory. The Alices would periodically and automatically fetch test scripts that cause them to join different circles and send synthetic traffic (the scripts would of course need to be audited and signed by a trusted party operating the laboratory). One of the most important properties of this kind of testing is that it would be capable of generating the large amounts of data about latencies and dropped packet rates that are necessary for detecting ISP discrimination that is more subtle than outright interference with the data. Users could be informed of the progress of their tests by a small feedback widget that keeps them engaged with the collective task of network preservation (other distributed computing projects such as SETI@Home have had success with this kind of mechanism).

Protocol

For rapid development, the first version of Switzerland's servers and clients has been implemented in Python, and the first version of the wire protocol uses stream of objects serialized by the Python Cerealize (secure Pickle) library[16] over an OpenSSL link. Later versions may benefit from a lower-level protocol implementation, or from the use of C or C extensions to Python.

Protocol events involve an initial message in one direction, with one or more followups exchanged.

-> denotes message from Alice to Switzerland
<- denotes message from Switzerland to Alice

16 http://home.gna.org/oomadness/en/cerealizer/index.html. We have patched Ceralizer slightly to support multiple cerealized objects in the one stream. We believe the patch will be incorporated upstream soon.

?> a message in either direction
<? reply

Messages that expect replies, and the replies themselves, contain sequence numbers (which we hide here). Some messages require `ack` replies. These are not documented here yet.

The protocol will begin with OpenSSL connection establishment, over which the Switzerland protocol begins its handshake, which has two parts:

The first part is raw binary[17]:

-> `WellHello`<unsigned int32 client protocol version>[18]

If the server is capable of speaking the protocol version requested by the client, it replies:

<- `Switzerland`<highest protocol version spoken by the server (unsigned int32)>

if it cannot speak the protocol version requested, it must reply:

<- `Incompatibl`<protocol version (unsigned int32)>

Assuming they have agreed on a protocol version, each party should begin a stream of cerealized Python structures:

```
?> ["my-ip", private ip]
<? ["you-are-firewalled", your public ip]
```
or `<? ["not-firewalled"]`

`-> ["parameters", {"clock dispersion" :` root dispersion reported by NTP on the localhost`]`

```
<- ["new-members",{ip1:hashmask1,ip2:hashmask2,..},
{(option_name,val), (option_name,val),..}]
```

These messages tell Alice who the other members of her circle are, as well as the hashmasks she should use for reporting traffic with each of them.

---

17 In order to prevent potential future non-Cerealize variants of the protocol from having to perform any cerealizing or de-cerealizing.
18 XXX is there an issue of Endian-ness for these unsigned 32 bit integers?

```
?>[“ping”]
```

Ping just tests to see if the other party is alive.

```
?>[“error-bye”, explanation]
```

A fatal error.

```
?>[“error-cont”, explanation]
```

A non-fatal error .

```
<- [“farewell”, ip]
```

One of the other clients has left.

```
<- [“incognito”, [list,of,ips]]
```

Can't ping these machines (not yet implemented)

```
-> [“active-flows”, [(flow_id1,opening_hash1,flow_tuple),...],
[flow_id3, flow_id4]]
```

The first list in the arguments here contains newly active flows. This will allow Switzerland to decide when an (Alice, Bob) pair are participating in the same flow. Flow tuples are standard flow representations: `(src_ip, src_port, dest_ip, dest_port, protocol).` Switzerland uses `(src_ip, dest_ip, opening_hash)` to decide if flows reported by Switzerland. It may also use one or both port numbers if one or both sides are not firewalled. [XXX Also, in the future, flows to an unfirewalled Bob should be handled as a special case if their opening hashes do not match.]

The second list of arguments here contains flows which are no longer active and which may be deleted by Switzerland. [XXX we need to decide what to do in the case where one side continues to report activity on a flow and the other asks to delete it]

```
-> [“sent”, flow_id, timestamp, ”concatented_masked_hashes”]
```

and

```
-> ["recd", flow_id, timestamp, "concatenated_masked_hashes"]
```

These are the main reporting mechanism; there's no response or ack because a lot of these are going to be sent. The size of the set of concatenated masked hashes should be tuned to ensure that one TCP packet gets sent per Switzerland "sent" event.[19]

When packets are dropped on their way from Alice to Bob, Alice receives:

```
<- ["dropped-out","concatenated_masked_hashes"]
-> ["dropped-details", {masked_hash1 : pcap_packet, ..}]
```

Alice should not send details for dropped packets that she has already sent elsewhere, such as in `fo-context` or `fi-context` messages. In the mean time, Bob receives:

```
<- ["dropped-in","concatenated_masked_hashes"]
```
(Switzerland waits for `dropped-details` from Alice)
```
<- ["dropped-details", {masked_hash1 : pcap_packet, ..}]
```

(`dropped-details` messages have not yet been implemented)

```
<- ["forged-in", flow_id, [(timestamp, hash)]]
-> ["fi-context", {hash1 : [(timestamp1a,packet1a),
(timestamp1b,packet1b)], hash2:...}]
```

Bob responds to news that packets he received are forgeries by sending full details for those packets and for surrounding packets in the flow. Switzerland then passes this information to Alice:

```
<- ["forged-out", flow_id, [(timestamp, context_message)]]
```

where the context_message is the list indexed by hash from the `fi-context` message from Bob. All context messages start with the forged packet first followed by the others in chronological order.

Alice tries to identify the outgoing packets most likely to have triggered the forgeries:

```
-> ["fo-context", {hash1 : [(timestamp,packet),..], .. }]
```

Switzerland awaits `fo-context` from Alice before being able to send `forged-details` to Bob..

```
<- ["forged-details", {hash1 : [(timestamp,packet),..]
```

19 This will require some examination of Cerealize and OpenSSL overheads.