Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Review-Report Turbo Tunnel Security & Privacy 03.2021

Cure53, Dr.-Ing. M. Heiderich and various Cure53 Team Members

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

> *"The idea - which I call Turbo Tunnel - is simple, but has many benefits. Decoupling an abstract session from the specific means of censorship circumvention provides more design flexibility, and in some cases may increase blocking resistance and performance. This work motivates the concept by exploring specific problems that a Turbo Tunnel design can solve, describes the essential components of such a design, and reflects on the experience of implementation in the obfs4, meek, and Snowflake circumvention systems, as well as a new DNS over HTTPS tunnel."*
>
> From https://www.bamsoftware.com/papers/turbotunnel/

This report - entitled *UCB-02* - details the scope, results, and conclusory summaries of a penetration test and security assessment against the Turbo Tunnel project, its peripheral software and dependencies.

The work was requested by UC Berkeley in January 2021 and conducted by Cure53 in March and April 2021, namely in calendar weeks CW11 to CW13. A total of 15 days were invested to reach the coverage expected for this project. A team of five senior testers were assigned to this project's preparation, testing, audit execution, and finalization. The testing conducted throughout *UCB-02* was divided into two separate work packages (WPs) for execution efficiency, as follows:

- **WP1**: White-Box Tests and Source Code review of Snowflake
- **WP2**: White-Box Tests and Source Code review of dnstt

Cure53 was granted access to all relevant sources in scope, as well as a meticulously detailed scope document that mapped out the constitution of the software architecture, purposes, limitations, security promises, and audit expectations. Given that all of these assets were necessarily required to procure the coverage levels expected by UC Berkeley, the methodology chosen here was white-box.

All preparations were completed in February and early March 2021 to ensure that the testing phase could proceed without hindrance. Preparations were handled exceptionally by the maintainer team. Communications during the test were facilitated via email, which was selected as the preferred communication medium by the Turbo Tunnel software maintainers. One can denote that communications proceeded smoothly on the whole. The scope was well prepared and clear, and no noteworthy roadblocks were encountered throughout testing. Cross-team queries were abundant owing to the complexities of the software and setup architecture, though the Turbo Tunnel

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

maintainers delivered excellent assistance toward this, cooperating with the Cure53 team in every respect to procure maximum coverage and depth levels for this exercise.

Cure53 gave frequent status updates toward the test and related findings, updating the maintainers about any and all identified issues. Live reporting was not requested and also not deemed necessary given the nature of the findings listed in this report. With regard to the aforementioned findings, the Cure53 team was able to procure exceptional coverage over the WP1 and WP2 scope items, unearthing a total of nine. One of these findings was categorized as a security vulnerability; the remaining eight were deemed to be general weaknesses with lower exploitation potential.

The highest severity level reached was *Medium* - as detailed in UCB-02-003 - concerning a potential nonce overflow. The majority of findings were severity-rated *Low* and *Informational*, which is a positive indication for the software compound in scope. Despite a comprehensive, extensive test and audit phase facilitated by excellent collaboration between the maintainers and test team, no findings of *High* or even *Critical* severity were detected.

The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. Subsequently, the report will list all findings identified in chronological order. Each finding will be accompanied by a technical description and Proof-of-Concepts (PoCs) where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the Turbo Tunnel project, its peripheral software and dependencies, giving high-level hardening advice where applicable.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Scope

- **Turbo Tunnel security and privacy review & code audit**
  - **WP1**: White-Box Tests and Source Code review of *Snowflake*
    - Cure53 was provided with a very detailed scope description that clearly outlines the goals of this audit, expected threat and risks as well as interesting areas in the code to cover
    - More info can be found here
      - https://www.bamsoftware.com/papers/turbotunnel/
  - **WP2**: White-Box Tests and Source Code review of *dnstt*
    - All relevant information was made available to Cure53 prior to the tests and code audits starting
    - More info can be found here
      - https://www.bamsoftware.com/software/dnstt/
      - https://www.bamsoftware.com/papers/thesis/
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were accessible to Cure53**
    - https://gitweb.torproject.org/pluggable-transports/snowflake.git/
    - https://github.com/xtaci/kcp-go
    - https://github.com/xtaci/smux

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list all vulnerabilities and implementation issues identified throughout the testing period. Please note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Furthermore, each vulnerability is given a unique identifier (e.g., UCB-02-001) for the purpose of facilitating any future follow-up correspondence.

UCB-02-003 WP2: Potential nonce overflow in Noise protocol *(Medium)*

The Noise implementation deployed by dnstt uses an implicit nonce counter for symmetric packet encryption. This counter is a 64-bit unsigned integer and will thus overflow after 2^64 messages encryption using the same key. This is problematic for most symmetric ciphers, because encrypting multiple inputs using the same key and nonce can have fatal consequences on the security of the cipher. The Noise specification chapter 5.1[1] describes the behavior as follows:

"The maximum n value (2^{64}-1) is reserved for other use. If incrementing n results in 2^{64}-1, then any further EncryptWithAd() or DecryptWithAd() calls will signal an error to the caller."

It has to be noted that this is not followed by the Noise implementation[2] used by dnstt.

Affected File:
noise-go/state.go

Affected Code:

```
func (s *CipherState) Encrypt(out, ad, plaintext []byte) []byte {
    if s.invalid {
        panic("noise: CipherSuite has been copied, state is invalid")
    }
    out = s.c.Encrypt(out, s.n, ad, plaintext)
    s.n++
    return out
}
```

One can recommend performing a new handshake even before transmitting up to 2^64 - 2 messages.

[1] https://noiseprotocol.org/noise.html#the-cipherstate-object
[2] https://github.com/flynn/noise

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The reason for this is that whenever the current *CipherState* key is compromised, all messages encrypted with this key become decryptable. As a result, reducing the number of messages encrypted with the same key reduces the volume of potentially-compromised messages.

Miscellaneous Issues

This section covers any and all noteworthy findings that did not lead to an exploit but might assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

UCB-02-001 WP1: Memory leak in *Handler()* routine of Snowflake client lib *(Low)*

During a review of the Snowflake client library, the discovery was made that the *Handler()* function - responsible for establishing a WebRTC connection to the remote peer - does not correctly close the connection and established smux session in the eventuality that a stream cannot be opened. This could result in a memory leak on the Snowflake client side, as well as a resource leak on the server side of the connection.

Affected file:
snowflake/client/lib/snowflake.go

Affected code:

```
func Handler(socks net.Conn, tongue Tongue) error {
	[...]
	// Create a new smux session
	log.Printf("---- Handler: starting a new session ---")
	pconn, sess, err := newSession(snowflakes)
	if err != nil {
		return err
	}
	// On the smux session we overlay a stream.
	stream, err := sess.OpenStream()
	if err != nil {
		return err
	}
	[...]
}
```

It is recommended to close all open connections using *defer*[3] in order to properly alleviate all allocated resources when the function returns.

[3] https://tour.golang.org/flowcontrol/12

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

UCB-02-002 WP2: Memory leak in *acceptStreams()* routine of dnstt server *(Low)*

During a review of the dnstt-server source code, the discovery was made that the *acceptStreams()* function - responsible for wrapping a KCP session in a Noise channel and smux session - does not correctly close the smux session in the eventuality that a stream cannot be opened. This could result in a memory leak on the server side of dnstt. Generally speaking, the opened smux session does not close when *acceptStreams()* returns, even when a failure is not present.

Affected file:
dnstt/dnstt-server/main.go

Affected code:

```
func acceptStreams(conn *kcp.UDPSession, privkey, pubkey []byte, upstream
*net.TCPAddr) error {
	[...]
	sess, err := smux.Server(rw, smuxConfig)
	if err != nil {
		return err
	}

	for {
		stream, err := sess.AcceptStream()
		if err != nil {
			if err, ok := err.(net.Error); ok && err.Temporary() {
				continue
			}
			return err
		}
	[...]
```

The recommendation can be made to close all open sessions using *defer*[4] in order to properly alleviate all allocated resources when the function returns.

[4] https://tour.golang.org/flowcontrol/12

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

UCB-02-004 WP2: Deprecated DH25519 Golang API used by Noise *(Low)*

A review of the active Noise library confirmed that the Noise implementation deploys the old, deprecated[5] *curve25519.ScalarMult(dst, scalar, point *[32|byte)* function. This function was deprecated in favor of the new *curve25519.X25519(scalar, point []byte) ([]byte, error)* call, which adds an input validation to low-order points. This check is needed as per the design of curve25519, but is intended to help identify errors with more atypical protocols.

Affected file:
noise-go/cipher_suite.go

Affected code:

```
func (dh25519) DH(privkey, pubkey []byte) []byte {
    var dst, in, base [32]byte
    copy(in[:], privkey)
    copy(base[:], pubkey)
    curve25519.ScalarMult(&dst, &in, &base)
    return dst[:]
}
```

While this is not a security issue for Noise and dnstt, one can still recommend not relying on deprecated API's solely. Therefore, it is encouraged to switch to the new API call at the earliest possible convenience.

UCB-02-005 WP2: Client ID security considerations & Noise authn'd data *(Low)*

During a code and design review of dnstt[6], the testing team observed that the dnstt server tracks dnstt clients by storing a correlation between Client ID and IP addresses. Every DNS query sent by the dnstt client is tagged with a Client ID, which is a random 64-bit string generated by the client.

One can pertinently note that Client IDs are not permanent identifiers; the IDs only persist as long as the client software is active, and the server expires stale mappings after a certain amount of time. The Client ID transmission is neither within the encrypted Noise channel nor authenticated. An attacker capable of obtaining a valid Client ID - via sniffing network communication, for example - could confuse the dnstt server and potentially receive messages intended to be transmitted to the original client corresponding to the obtained Client ID. It is important to stress that an attacker would not be able to decrypt the actual encapsulated message, as it is end-to-end encrypted and protected by Noise.

[5] https://github.com/golang/go/issues/32670
[6] https://www.bamsoftware.com/software/dnstt/protocol.html

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

While reviewing the usage of the Noise protocol within dnstt, the discovery was made that the Noise protocol's authenticated data remains unused. One could use this area to place plaintext data which should undergo an authentication, such as the referred Client ID or similar. This would not solve the aforementioned nonce synchronization issue, but would result in tighter coupling of the Noise channel with the underlying Turbo Tunnel session identified by a Client ID. The following code snippet displays the method by which the Noise implementation's *Encrypt()* function is invoked without applying any authenticated data.

Affected files:
dnstt/noise/noise.go

Affected code:
```
func (s *socket) Write(p []byte) (int, error) {
	total := 0
	for len(p) > 0 {
		[...|
		err := writeMessage(s.ReadWriteCloser, s.sendCipher.Encrypt(nil, nil, p[:n]))
		[...]
	}
	return total, nil
}
```

Affected file:
https://github.com/flynn/noise/blob/master/state.go

Affected code:
```
// Encrypt encrypts the plaintext and then appends the ciphertext and an
// authentication tag across the ciphertext and optional authenticated data //
to out. This method automatically increments the nonce after every call, // so
messages must be decrypted in the same order.
func (s *CipherState) Encrypt(out, ad, plaintext []byte) []byte {
	if s.invalid {
		panic("noise: CipherSuite has been copied, state is invalid")
	}
	out = s.c.Encrypt(out, s.n, ad, plaintext)
	s.n++
	return out
}
```

It is recommended to place the Client ID into the authenticated data in order to achieve tighter coupling of the Noise channel with the underlying Turbo Tunnel session identified by a Client ID.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

UCB-02-006 WP2: DoS due to unconditional nonce increment *(Low)*

Testing identified that an active attacker with the capability to send messages to the dnstt server and the knowledge of another client's Client ID is able to interrupt the legitimate client's communication. This is possible because an attacker can tag messages with other client's Client ID values, resulting in a decryption error on the server side within the Noise layer.

Typically, the server should remove the Noise CipherState and require a client to re-establish a new Noise session upon decryption errors. The Noise implementation used by dnstt, however, does not follow the Noise specification; nor does it do so independently of any decryption error incremented on every received message. Consequently, any valid message from the legitimate client would also lead to a decryption error, because the nonce counters would be out of sync from this moment onwards. The legitimate client would need to rerun a Noise handshake to agree on a new session key and resync the nonce counters.

The relevant section of the Noise specifications is in chapter 5.1[7]:

"***DecryptWithAd(ad, ciphertext)****: If k is non-empty returns DECRYPT(k, n++, ad, ciphertext). Otherwise returns ciphertext. If an authentication failure occurs in DECRYPT() then n is not incremented and an error is signaled to the caller.*"

Affected file:
https://github.com/flynn/noise - *state.go*

Affected code:
```
func (s *CipherState) Decrypt(out, ad, ciphertext []byte) ([]byte, error) {
    if s.invalid {
        panic("noise: CipherSuite has been copied, state is invalid")
    }
    out, err := s.c.Decrypt(out, s.n, ad, ciphertext)
    s.n++
    return out, err
}
```

It is recommended to follow the Noise specification and ensure that the nonce is only incremented on successful decryption. Alternatively, the Noise CipherState object should be removed and a new handshake established.

[7] https://noiseprotocol.org/noise.html#the-cipherstate-object

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

UCB-02-007 WP2: DoS due to missing socket timeouts *(Low)*

The discovery was made that some network sockets have established deadlines for IO operations. In a hostile environment whereby connections are subject to being dropped, one should deem it crucial to be able to reconnect quickly. *tls.Dial***,** as used in *dnstt-client/tls.go*, does not set deadlines, so any connect or read operations can block for a potentially infinite duration in the worst case scenario.

Affected file:
dnstt-client/tls.go

Affected code:
```
[...]
tlsConfig := &tls.Config{}
conn, err := tls.Dial("tcp", addr, tlsConfig)
if err != nil {
	return nil, err
}
[...]
```

It is recommended to deploy *tls.Client* in conjunction with *net.DialTimeout* instead of *tls.Dial* when establishing a new TLS connection, and to set a deadline before each read operation using *conn.SetReadDeadline*.

UCB-02-008 WP1-WP2: Lack of rate limiting in Snowflake and dnstt *(Info)*

While reviewing the dnstt and Snowflake source code, the testing team observed that neither dnstt nor Snowflake implement rate limiting on the server side. Rate limiting would effectively reduce the risk of malicious users instigating a situation whereby the dnstt server or Snowflake server is forced to exhaust resources such as memory, for example.

Affected file:
dnstt/dnstt-server/main.go

Affected code:
```
func recvLoop([...]) error {
	for {
		// Got a UDP packet. Try to parse it as a DNS message.
		query, err := dns.MessageFromWireFormat(buf[:n])
		[...]
		if n == len(clientID) {
			[...]
			r := bytes.NewReader(payload)
			for {
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
                        [...]
                        // Feed the incoming packet to KCP.
                        ttConn.QueueIncoming(p, clientID)
                }
        }
    }
}
```

Affected file:
snowflake/server/server.go

Affected code:

```
func turbotunnelMode(conn net.Conn, addr string, pconn
*turbotunnel.QueuePacketConn) error {

    [...]
    clientIDAddrMap.Set(clientID, addr)

    // The remainder of the WebSocket stream consists of encapsulated
    // packets. We read them one by one and feed them into the
    // QueuePacketConn on which kcp.ServeConn was set up, which eventually
    // leads to KCP-level sessions in the acceptSessions function.
    go func() {
        for {
            p, err := encapsulation.ReadData(conn)
            if err != nil {
                errCh <- err
                break
            }
            pconn.QueueIncoming(p, clientID)
        }
    }()
    [...]
}
```

Careful consideration is required when assessing rate-limiting implementation strategy, as this feature can also be abused by malicious users to effectively deny access to dnstt or Snowflake. Cure53 would highlight this potential area for improvement as one that should be investigated internally. One can also recommend assessing the risks of causing a Denial-of-Service situation in which the dnstt or Snowflake servers depreciate their resources.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

UCB-02-009 WP1: Brokers and proxies are not authenticated *(Low)*

Broker nodes within Snowflake handle the rendezvous by matching Snowflake clients with proxies, and are responsible for passing along WebRTC session descriptions during signaling, allowing clients and proxies to establish a peer connection. During an audit of Snowflake, the observation was made that neither brokers nor proxies are authenticated, therefore potentially allowing an attacker to register rogue broker and proxy nodes.

One can pertinently note that the lack of authentication regarding brokers and proxies has already been discussed with the client during the course of this security review, and the confirmation was made that the Tor project already has ongoing discussions around adding authentication[8] [9] [10] [11] [12]. Cure53 recommends inserting authentication of broker messages in order to eliminate the risk of rogue broker nodes and potential Denial-of-Service caused by malicious brokers.

[8] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/22945
[9] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/25593
[10] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/31124
[11] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/25681
[12] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/31804

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW11 to CW13 testing against the Turbo Tunnel project, its peripheral software, and dependencies by the Cure53 team - will now be discussed at length. To summarize, the confirmation can be made that the applications under scrutiny have left a good impression.

The software projects in scope - dnstt and Snowflake - were comprehensively audited to assess their constitution and code. Generally speaking, testing confirmed the notion that both solutions are architecturally sound and precise in this regard.

The testing team was able to understand how the individual pieces integrate and comprise the applications in a straightforward manner. All important functions and steps had comments, which assisted toward understanding code activity and helped to smooth the auditing process. Most error situations that could potentially arise are caught and handled properly.

For the review of dnstt, enhanced focus was asserted towards its protocol stack, which combines KCP, Noise and smux. These protocols are correctly deployed for the most part in any given situation, although some deviations from the Noise protocol and potential error scenarios were detected.

Rigorous attention was also paid to the Noise implementation and usage. On the whole, the selection of Noise as a platform is a great choice from a security perspective. The level of security possible via Noise greatly exceeds that which the kcp-go implementation provides, which was also briefly reviewed. It is important to note that the kcp-go encryption feature is not recommended for dnstt. KCP, Noise and SMUX seem to be completely independent of each other. While this is acceptable, depending on the thread model and level of security, this also provides a broader potential attack surface. For example, an attacker that is able to successfully guess or obtain a valid Client ID of another client could inject random packets into that message queue, causing at least some disruption toward another client’s tunnel.

While the Noise implementation is generally secure, it does not seem to follow the specification in multiple areas - particularly with regard to nonce management and error handling in the encrypt and decrypt paths. The recommendation can be made to either fix these issues in the current implementation, or use a Noise implementation that precisely follows the specification to avoid any security issues that might arise from this instance.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

One can also recommend that any errors in the encryption and decryption path should properly trigger a full reconnect of the client at the Noise level, ideally with a newly-generated Client ID. For the decrypt path, this can be loosened and the Noise channel can be kept active, but the nonce counter must never be incremented in case of an error.

Furthermore, a maximum limit on the volume of messages secured with the same Noise CipherState below 2^64 - 2 messages should be established. This would reduce the number of decryptable messages whenever a session key is compromised. In addition, the testing team observed that no rekeying method has been established either. This would impose a security risk depending on the lifetime of the session since symmetric keys should be substituted following excessive usage, depending on the application.

Testing also identified that the nonce is always incremented even when the decryption of the Noise library fails, as detailed in UCB-02-006. This confuses an internal object related to Noise since the nonces are assumed to be monotonically increasing.

Additionally, the Noise protocol's authenticated data remains unused. One can advise implementing this area to insert plaintext data required to authenticate, such as the Client ID, for example (see UCB-02-005 for further details).

If the maintainers are considering installing client authentication, it is strongly recommended to have a look at the notes on channel binding within the Noise specification. The dnstt implementation does not perform any form of channel binding at the time of the audit.

One noteworthy observation relates to the fact that the dnstt client application does not validate the provided public key (from file or input string) of the dnstt server with regard to authenticity, which would allow for Man-in-Middle attacks in the eventuality that an attacker is able to control the client's public key input file or command line arguments. The user launching the dnstt client application is forced to trust the provided dnstt server public key blindly, as there are no technical measures in place that would verify the public key.

For the review of the Snowflake constitution, considerable attention was given to the Turbo Tunnel implementation, the cryptographic primitives, and the misuse of the server APIs. As agreed with the client at the time, further additions to Snowflake on Turbo Tunnel's behalf were deemed the major focus concerning its audit. In comparison with *dnstt,* Snowflake does not use Noise, and encryption is performed one layer below using WebSocket and WebRTC. The connection reestablishment mechanism was highly scrutinized during the testing in addition; no issues were identified.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The observation was made that neither the brokers nor the proxies are authenticated, which would allow an attacker to impersonate said brokers or proxies (as detailed in UCB-02-009), which act as a proxy from the client's perspective. This allows an attacker to place and register rogue brokers and proxies.

The Snowflake and dnstt implementations are inherently vulnerable to Denial-of-Service attacks, as there is no mechanism in place to limit the volume of connection attempts as described via ticket UCB-02-008.

In summation, dnstt and Snowflake made a mature impression. Evidence clearly suggests that the developers are aware of secure-coding best practices and understand the method by which to write clean and robust code in Golang. This is corroborated by the unusually low number of identified vulnerabilities and miscellaneous issues. The majority of identified issues related to the Noise implementation that is deployed by dnstt and generic design decisions implemented; for example, the lack of authentication of brokers and proxies as detailed in UCB-02-009, or missing rate limiting as detailed in UCB-02-008, for instance. Conclusively, extensive auditing confirmed that the Turbo Tunnel concept and its implementation in dnstt and Snowflake are robust with regards to security albeit with some room for improvement, as demonstrated within this report.

Cure53 would like to thank Xiao Qiang from the UCB team and David Fifield for their excellent project coordination, support and assistance, both before and during this assignment.