UbiCrypt: Making ubiquitous encryption compatible with enterprise security

Weisse, Ofir *Trippel, Timothy* *Erickson, Jeremy*
EECS 589, Fall 2015
University of Michigan, Ann Arbor

Abstract

In many enterprise environments, network traffic introspection is a necessity. To make this possible, current implementations man-in-the-middle all network traffic using a technique known as split TLS. This has many drawbacks, including breaking the fundamental notion of end-to-end encryption. In this project we present UbiCrypt, an alternative to split TLS, which maintains end-to-end encryption, while still allowing trusted introspection on local network traffic. UbiCrypt provides a mechanism to securely leak ephemeral session encryption keys to a trusted gateway. UbiCrypt is easily deployable as it only requires software modifications and additions to the client and trusted gateway, not endpoint servers, which may not be owned by the enterprise.

UbiCrypt was demonstrated using the QUIC protocol and its implementation in the publicly available Chromium source code. To add client support, we made slight modifications to the QUIC client code and added an external software module. To add gateway support, we built a prototype using iptables, a Linux firewall kernel application. We evaluated Ubicrypt using a virtual network topology, built using minimega, a virtual machine (VM) management tool. We demonstrate that Ubicrypt is practical by loading a small (1.6KB) webpage, showing that the overhead observed when using UbiCrypt was very similar to the overhead observed when using split TLS, 45.08 and 39.00 milliseconds respectively.

1 Introduction

With recent data breaches[2], many in the security community are calling for ubiquitous encryption to become the norm. That is, all online communications, whether secret or mundane, should be end-to-end encrypted. Several well-known projects[8, 5] are pushing for all web traffic to be encrypted with Transport Layer Security (TLS), most widely used in HTTPS web communications. As this trend continues, users will become more secure from intermediate entities snooping on their data or being able to modify it in transit. Ubiquitous encryption is very promising.

However, the same defenses that protect users against a malicious third party viewing or modifying data in transit block enterprise security measures from being able to introspect on corporate traffic and provide legitimate security services to the company. For instance, many companies use Snort[14], a common network Intrusion Detection System (IDS), to parse incoming network traffic for signatures indicating malware has been downloaded by an employee, but Snort is unable to identify malicious signatures[4] in encrypted flows.

In enterprise environments, often the need to introspect on corporate network traffic is greater than the need for true end-to-end security. The naive solution in place

today is for a company to simply man-in-the-middle its employees' secure connections at the enterprise gateway[3], also known as "split TLS". It does this by supplying an enterprise-owned root Certificate Authority (CA) to each user and creating two encrypted sessions for each connection — one between the client and gateway and one between the gateway and server.

This has several downsides stemming from the fact that the client must trust the gateway to securely connect to the server on its behalf. The gateway may trust a certificate that the client does not, thus reducing the client's ability to discern untrusted connections, or the gateway may not trust a certificate that the client does, thus either blocking the connection or making the connection without the assurance of authentication the client would be able to achieve on its own. For instance, one of the core tenets of public/private key authentication is that, even without a Public Key Infrastructure (PKI) to distribute and validate keys, the client can manually import a public key as trusted and form a secure connection to the owner of the corresponding private key. Not so if the gateway intercepts the connection.

In this paper we preset, implement, and evaluate an alternative to split TLS, UbiCrypt, one which preserves the desired properties of end-to-end encryption while still allowing introspection on network traffic by securely leaking session encryption keys to a trusted gateway. To demonstrate our design, we present a modification to the QUIC[11] protocol, an application layer protocol developed by Google, and software support at the gateway. This paper makes the following contributions:

- A new protocol architecture, UbiCrypt, for providing end-to-end encryption while still allowing a trusted gateway to introspect on network traffic.
- Client support for UbiCrypt through a modification to the QUIC protocol and its implementation in the Chromium source code.
- Gateway support for UbiCrypt through the implementation of new software to handle the reception and storing of leaked TLS session encryption keys.
- A performance evaluation of UbiCrypt using an emulated enterprise network environment, implemented with minimega[9], with deployments of the modified Chromium source code, gateway software, and QUIC server.

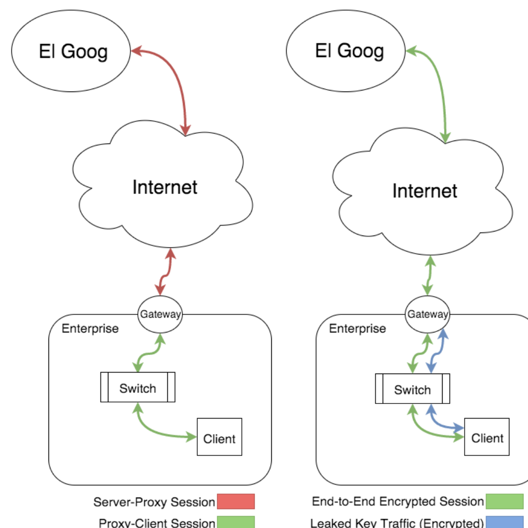

Figure 1: Comparison of existing split TLS network flows (left) vs. proposed UbiCrypt network flows (right)

2 UbiCrypt: An Alternative to Split TLS

To overcome the drawbacks of split TLS, we have designed UbiCrypt. UbiCrypt preserves end-to-end encryption while still allowing trusted introspection by providing a mechanism for clients to securely leak their session encryption key to the gateway. Figure 1 illustrates the architecture of UbiCrypt (right) in comparison to the split TLS architecture (left).

Initially, the UbiCrypt enabled gateway allows the first session negotiation packet through, until the session encryption keys are generated. Once the session keys have been generated for the initial flow, a UbiCrypt client makes a separate connection to the UbiCrypt enabled gateway to securely leak the session keys. In this architecture, the gateway buffers all packets, except the initial pair of negotiation packets, until it receives valid session keys for the corresponding flow. If the gateway does not receive valid keys after a set period of time, it drops all buffered packets in the flow and blocks all future incoming traffic for the same flow. This allows the trusted gateway to enforce its introspection policies while still maintaining a sense of end-to-end encryption.

UbiCrypt was designed as a modification to the QUIC protocol [11]. QUIC is an application layer protocol that sits on top of UDP and utilizes a cryptographic protocol similar to TLS for authentication and encryption. This allows UbiCrypt to work with existing network stack software. QUIC is also a recently-developed protocol that has already been implemented in an open-source code base, Chromium, and as adoption is still growing, there is more of a chance to have our proposed functionality incorporated in the mainline.

3 Support for UbiCrypt

3.1 Server Support for UbiCrypt

For UbiCrypt to succeed, it must be relatively easy to adopt. Requiring server-side modifications to enable UbiCrypt means that the enterprise will be forced to choose between only allowing access to those websites which adopt UbiCrypt, or more realistically, falling back to split TLS. Consequently, UbiCrypt requires no server-side modifications and is compatible with all sites that use QUIC. This model also holds if UbiCrypt is extended to handle TLS connections.

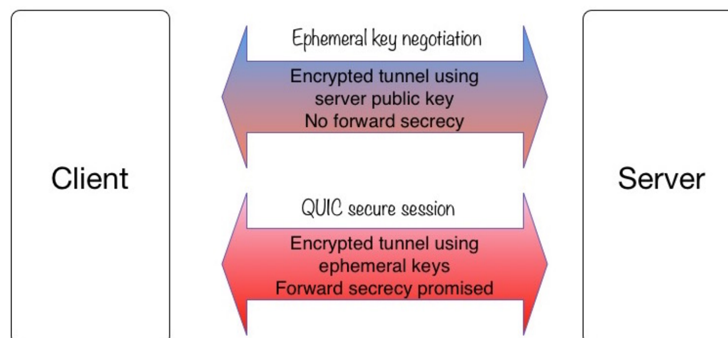

Figure 2: Establishing secure, forward secrecy preserving tunnel.

3.2 Client Support for UbiCrypt

In order to enable support for UbiCrypt, the client needs to exfiltrate the session keys to the gateway. During a QUIC session there are two sets of keys used. The first set

is generated based on the QUIC server public keys. This set could theoretically be used for the entire session. Unfortunately, this set does not provide forward secrecy. If they were used alone, a malicious eavesdropper could record the encrypted session transcript and save it for later use and, if in the future the eavesdropper were able to obtain the long-lived server private key, it could decrypt the entire recorded session. In order to prevent such an attack, the QUIC protocol uses two sets of keys. The first set is based on the server public key, and used to authenticate the server and establish an initial connection. Inside this encrypted session the client and the server negotiate ephemeral keys which will be used for the remainder of the session. The server will delete the ephemeral keys after the session is over, thus preventing future decryption of the transcript. Figure 2 depicts the establishment of the secure tunnel.

The QUIC source code can be roughly split into two parts: session management and cryptographic. The cryptographic part is responsible of generating key material and performing the encryption and decryption of packets. In the QUIC source code there are separate classes for encryption and decryption. Each one has a `SetKey` method which is called when new key material is generated. We intercepted this call in order to deliver the keys to the gateway. The first time `SetKey` is called we generate a unique identifier, based on the time stamp. Then, for each call we write the key material to a file. The file name contains the identifier, to classify this key material with the instance of the client. A separate Python module was written to then read the keys from the files and deliver them to the middle box. This technique has the benefit of performing the minimum work required in the client code in order to leak the keys. The time consuming task of opening a connection to the gateway and transmitted the keys is implemented in the external Python module. Figure 3 depicts the process of leaking keys to the middle box.

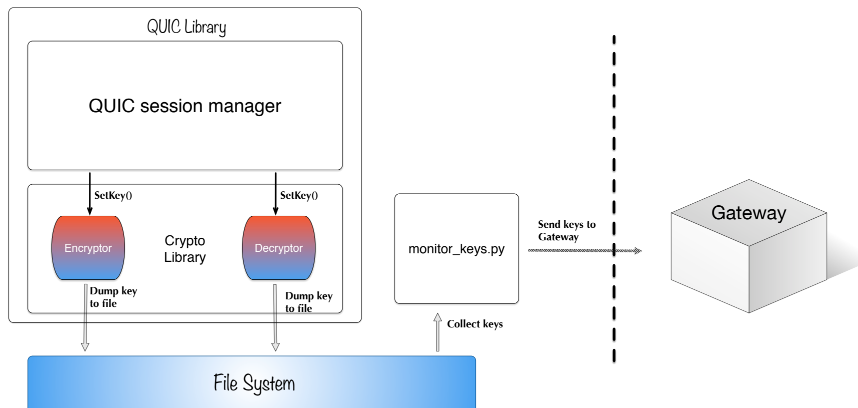

Figure 3: Leaking keys

3.3 Gateway Support for UbiCrypt

Fundamentally, it is the gateway's role to provide ingress and egress services for the enterprise network. In smaller enterprises, the gateway may simply be a middlebox between the main enterprise router and the Internet. In larger enterprises, the gateway could be a site-wide proxy. In truly huge or geographically distributed enterprises, the gateway could be replicated to serve the entire enterprise workforce. Regardless, by design, the gateway is the place at which the enterprise has decided to implement network monitoring and will implement the enterprise's security and privacy policies, whether that is to filter content, maintain audit logs, or detect suspicious or malicious behavior. In this case, we enable these actions by allowing the enterprise to enforce

the decryption of QUIC-encrypted content in real-time, while maintaining the ability of the client to verify the identity of the server.

It is important to realize that there are a number of potential implementations of the UbiCrypt gateway. For instance, we originally investigated the NTOP and NDPI projects as a way to decode and filter QUIC packets at 10 Gbps speeds, appropriate for even very large enterprise networks. However, due to the closed nature of the NTOP platform and the limited time constraints, we instead focused on a proof-of-concept implementation using IPTABLES, NFQUEUE, and the SCAPY[13] Python library.

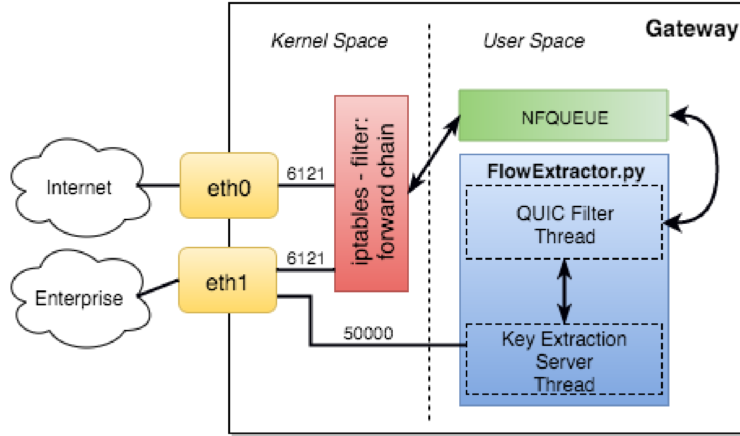

Figure 4: Diagram of gateway software architecture to filter QUIC flows and receive leaked session keys

Our gateway, conceptually, has one or more network interfaces connected to the Enterprise network, and one or more network interfaces connected to the Internet. All network traffic routed between the Enterprise and Internet will pass through our IPTABLES forward chain in kernel space. Of these packets, we filter all UDP port 6121 (the default QUIC port) traffic to a special NFQUEUE in user space that buffers the packets in memory until an application makes the decision to either accept or reject them. Figure 4 details our experimental architecture.

We have written a Python script, FlowExtractor.py, which opens a TCP server thread to accept keys from enterprise clients and then monitors the NFQUEUE and processes incoming QUIC packets. It uses the SCAPY packet processing library to interpret each packet's ethernet, IP, and UDP headers, and processes the raw QUIC header information manually. It groups QUIC packets by their Connection ID (CID) field, which is a random, unique field for each connection, and processes the packets of each connection, using their `packet_number` field, as follows: The first packet in each direction is accepted and allowed to reach its destination. The first packet, from client to server, is encrypted with the server's public key. The return packet, from server to client, is encrypted with the ephemeral key, although until the client receives it, the client cannot generate the ephemeral key pair or leak it to the gateway, so we accept this packet as well. The remaining packets in the connection are delayed and buffered. Once the ephemeral keypair for the connection is received from the client, these packets are accepted as well. If the keypair is not received, a timeout will occur and the buffered packets will be rejected.

This allows the enterprise to enforce its policy of network introspection—clients cannot evade the policy by simply failing to leak their keys or else their network connections will terminate before they can communicate with the server. We acknowledge that the initial handshake is a potential source of information leakage in our current implementation, and we discuss ways to handle it more rigorously in Section 5.2.

4 Evaluation

4.1 Experimental Setup

To evaluate the performance of UbiCrypt, a simple network architecture was built using minimega [9]. minimega is an open source virtual machine (VM) management tool. It enables the easy creation of complex virtual network topologies using Open vSwitch[10] and KVM[7]. To simulate a small enterprise network, we launched three Linux VMs as shown in Figure 5. The client VM was allocated 512MB of RAM and 1 virtual CPU. The gateway was allocated 2GB of RAM and 2 virtual CPUs. The server was allocated 512MB of RAM and 2 virtual CPUs. All virtual machines were run on a commodity desktop with 16GB of RAM, an Intel® Core™i7-4790K CPU clocked at 4.00Ghz, and a Samsung 850 Pro Solid State Drive.

The client and server VMs are isolated from each other and can communicate only through the gateway VM. The gateway is configured with static routing rules to direct traffic between the client and server to their respective networks. The gateway is also configured, as discussed in Section 3.3, with iptables forwarding rules that pass UDP traffic on port 6121 up to the gateway's filtering logic in user space for determination of acceptance, buffering, or rejection. Due to the latency of copying packets from kernel space to user space, we expect to see some latency even when QUIC packets are immediately accepted.

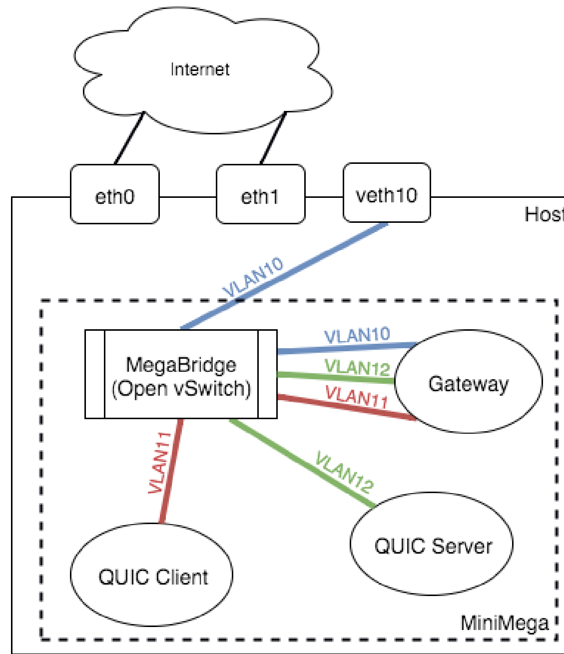

Figure 5: Diagram of network architecture, built using minimega [9] used for the performance evaluation

To effectively compare our implementation against the state of the art, we downloaded and built a split TLS module written by Daniel Roethlisberger, SSLsplit[12], on the gateway. We ran an Apache2[6] web server configured with TLS on our server VM, and used the wget web client to retrieve its HTTPS page. The SSLsplit application is written in C, optimized for production use, and we believe models the performance of an enterprise-grade split TLS installation[1].

[1] Although, dedicated solutions such as those sold by Blue Coat will presumably perform even better.

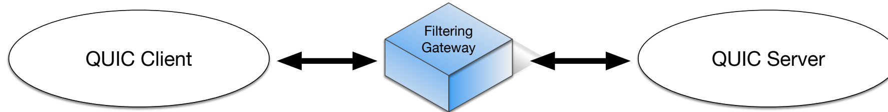

Figure 6: Logical network diagram

4.2 Performance

In evaluating UbiCrypt, we are primarily interested in the effect it has on the two standard metrics for network performance: latency and throughput.

4.2.1 Latency

To measure latency, we performed an initial set of experiments in which the client loads a small 1.6KB webpage from the server. For each such trial, we loaded the page 1000 times. As shown in Figure 7, a completely unmodified client completes this page load in an average time of 20.42 milliseconds. To measure the overhead of writing the QUIC secret keys to a file, we performed this experiment again on the modified QUIC client, which incurred only 0.32 milliseconds of additional latency. We believe this is negligible.

Next, we tested the page load time while gateway filtering was active. The page loaded in an average time of 65.50 milliseconds. Our initial thought was that our gateway filtering code was poorly optimized and we could reduce this number by reducing the number of times each packet was copied. After revising the gateway filtering code to eliminate spurious packet copying, we reran the page load time experiment. With our "optimized" gateway filtering code, the page surprisingly loaded in an average time of 66.72 milliseconds. This may indicate that our optimizations were poorly chosen, that the time spent processing packets is small compared to the time moving the packet from kernel space to user space. We discuss our optimization strategy and further potential optimizations in Section 5.1.

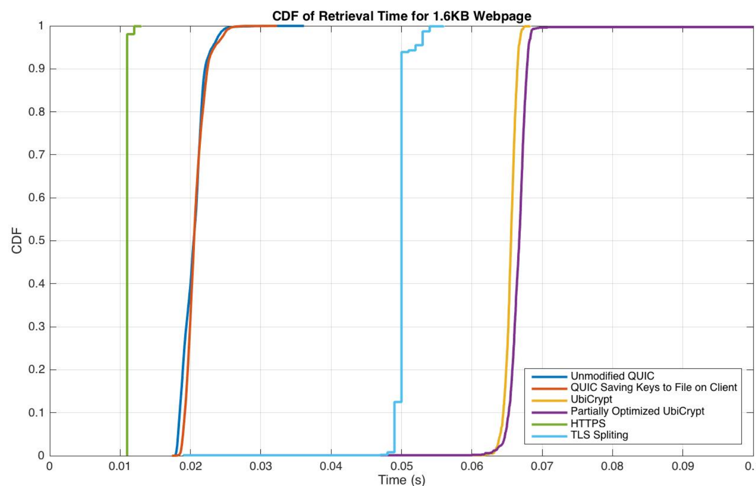

Figure 7: CDF plot of time to retrieve 1.6KB webpage with different client-server protocols

To compare this to the state of the art, we ran another set of tests loading the same 1.6KB webpage using Apache2, SSLsplit, and wget. Without SSLsplit in the way, the

page loaded in an average of 11.02 milliseconds. With SSLsplit, the page loaded in an average of 50.02 milliseconds.

We consider the overhead times of 45.08 and 39.00 milliseconds for UbiCrypt and SSLsplit, respectively, to be sufficiently similar to demonstrate that Ubicrypt is a viable alternative to SSLsplit, while also enabling end-to-end authentication.

4.2.2 Throughput

To gain a rough understanding of the throughput of UbiCrypt, we loaded an 80MB webpage to test the relative throughput of QUIC with and without UbiCrypt. Without UbiCrypt, the 80MB page loaded in an average of 8.92 seconds. With UbiCrypt filtering the QUIC packets, this payload took an average of 76.93 seconds, almost an entire order of magnitude slower. We discuss ways to reduce this overhead in Section 5.1. Unfortunately, SSLsplit caused the large download over HTTPS to fail, so we are unable to compare its performance relative to UbiCrypt.

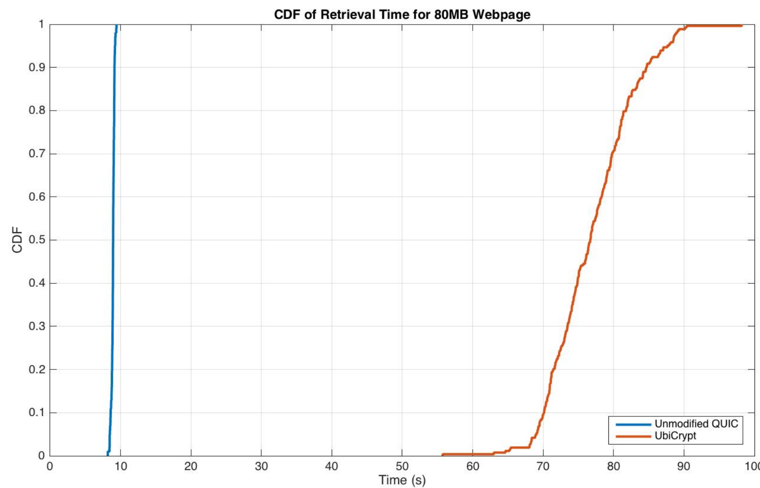

Figure 8: CDF plot of time to retrieve 80MB webpage

5 Discussion

5.1 Filtering Optimization

While performance of the current implementation of Ubicrypt appears similar to that of SSLsplit, we believe it can still be substantially improved.

In our attempt at optimizing the Python filtering code, we replaced a routine that repeatedly copied the packet payload in memory with code that used a single copy of the payload and an offset to keep track of specific packet header fields. We anticipated this would significantly speed up the filtering, and were surprised when it actually slightly decreased the performance. From this, we intuit that perhaps the actual processing of the packet headers is a very minimal source of latency. However, if we are wrong and the packet processing is in fact introducing overhead, there are still several optimizations we can perform to speed it up. First, we are are currently using a single thread to process all packets. Even if multiple packets are queueing up, we will continue to process them one by one. This could be fixed by multithreading the packet processing. Unfortunately, this is not immediately possible with Python's

libnetfilter_queue library, but there is another project, suricata[15], that has developed a multithreaded implementation of netfilter queues. Second, we are using Python for convenience when writing our prototype implementation, but Python is considered very slow for performance-intensive tasks due to the large data structures it uses. Moving to C, Go, or another lower-level language could potentially improve performance substantially.

Whether or not improving our filtering code would significantly improve performance, we know that a huge amount of overhead is incurred by the transition from processing packets in kernel space to user space. Ideally, we would want to leave the vast majority of packets in kernel space while making the decision to accept, delay, or reject. Packets containing a CID for which we already have the decryption key could be forwarded on with very minimal overhead, as could the first packets of each connection, using the packet_number field. Platforms such as NTOP tie into the system with a kernel module that gives them much faster packet monitoring. We envision a two-tiered system, similar to Open vSwitch[1], in which a small kernel module performs the core monitoring and filtering, and takes configuration about which connections to accept, delay, and reject from a more complex userspace module that handles decryption and/or interfacing with other enterprise appliances.

In comparison, we believe that the complexity of a split TLS scheme makes integration at the kernel level difficult if not impossible. Split TLS must not only intercept traffic, but decrypt the traffic, forge a new ephemeral certificate for the connection, and re-encrypt the traffic before passing it on. Vulnerabilities in such a complex kernel module would be likely and so, similar to the trouble Open vSwitch had in getting their original complex kernel module accepted into the mainline Linux kernel, we believe such an approach would not be accepted by the Linux kernel team. Thus, we believe the approach taken by Ubicrypt has significant long-term performance benefits over split TLS.

5.2 First-packet Introspection

One significant weakness with UbiCrypt's current implementation is that it invariably allows the first packets of each connection to pass through its filter without necessitating the ability to introspect on them. For an enterprise that aims to harness complete introspection capabilities on its users, this appears to be significant information leakage. The first packet in QUIC from client to server is encrypted with the server's public key information and so the client is actually incapable of leaking the key required to decrypt it.

To rectify this, we propose the addition of an "unlock" packet to the client key leakage implementation. That is, before the client can send the initial packet to the server, it must send the gateway the plaintext message of the subsequent encrypted packet, as well as any nonces or other plaintext components, such that the gateway can recompute the ciphertext of the initial handshake packet. This would allow the gateway to maintain its ability to introspect on the first packet of any connection.

Similarly, the current UbiCrypt implementation allows the first return packet from server to client through its filter. This packet is encrypted with the connection key, which the server has computed but the client does not yet know and therefore cannot yet leak to the gateway. So that the gateway can introspect on this packet as well, we propose that the client also share its half of the ephemeral key material with the gateway in the initial "unlock" packet. Then, the gateway can compute the ephemeral key when the return packet arrives, verify its ability to decrypt the packet, and forward the packet to the client. As a major side-benefit of this approach, since the gateway is able to compute the ephemeral key itself, there is no need for the client to subsequently leak the ephemeral keys as it does in the current implementation.

6 Conclusion

In this paper we present Ubicrypt, an effective alternative to split TLS for enterprise network traffic introspection that does not disrupt the guarantees of end-to-end authentication. Based on the performance results of our prototype implementation, it is clear that this approach is tractable, incurring similar overhead to a production-ready implementation of split TLS, SSLsplit, and we believe we can achieve even better performance with an optimized approach.

It is difficult to quantify the benefits of maintaining end-to-end authentication in enterprise environments, but as the world moves towards ubiquitous encryption, pure man-in-the-middle approaches will continue to pose both logistical and security problems in practice. We need a better compromise that maintains the security properties users assume they have while allowing enterprises to protect those users as well as enterprise assets. We believe that Ubicrypt may just be the answer.

References

[1] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130. ISBN: 978-1-931971-218. URL: `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff`.

[2] Edward Moyer. *NSA disguised itself as Google to spy, say reports*. Sept. 12, 2013. URL: `http://www.cnet.com/news/nsa-disguised-itself-as-google-to-spy-say-reports/`.

[3] Tim Chiu. *The Growing Need for SSL Inspection*. June 18, 2012. URL: `https://www.bluecoat.com/security/security-archive/2012-06-18/growing-need-ssl-inspection`.

[4] Joel Esler. *SSL/TLS*. Dec. 4, 2012. URL: `http://manual.snort.org/node147.html`.

[5] Barry Schwartz. *Google SSL Default, Goodbye Query Referrer Data*. Oct. 19, 2011. URL: `https://www.seroundtable.com/google-ssl-drops-query-data-14188.html`.

[6] *Apache HTTP Server Project*. URL: `https://httpd.apache.org/`.

[7] *KVM: Kernel Virtual Machine*. URL: `http://www.linux-kvm.org/page/Main_Page`.

[8] *Let's Encrypt*. URL: `https://letsencrypt.org/`.

[9] *minimega: a distributed VM management tool*. URL: `http://minimega.org/`.

[10] *OpenvSwitch*. URL: `http://openvswitch.org/`.

[11] *QUIC, a multiplexed stream transport over UDP*. URL: `https://www.chromium.org/quic`.

[12] Daniel Roethlisberger. *SSLsplit - transparent and scalable SSL/TLS interception*. URL: `http://www.roe.ch/SSLsplit`.

[13] *Scapy*. URL: `http://www.secdev.org/projects/scapy/`.

[14] *Snort*. URL: `https://snort.org/`.

[15] *suricata*. URL: `https://doxygen.openinfosecfoundation.org/source-nfq_8c_source.html`.