A Formal Treatment of Accountable Proxying over TLS

Karthikeyan Bhargavan[1], Ioana Boureanu[2], Antoine Delignat-Lavaud[3], Pierre-Alain Fouque[4], and Cristina Onete[5]
[1] *Inria de Paris,* [2] *University of Surrey, SCCS* [3] *Microsoft Research,* [4] *Université de Rennes 1, IRISA,*
[5] *Université de Limoges, XLIM, CNRS 7252*
Email: karthikeyan.bhargavan@inria.fr, i.boureanu@surrey.ac.uk, antld@microsoft.com, pa.fouque@gmail.com, cristina.onete@gmail.com

***Abstract*—Much of Internet traffic nowadays passes through active proxies, whose role is to inspect, filter, cache, or transform data exchanged between two endpoints. To perform their tasks, such proxies modify channel-securing protocols, like TLS, resulting in serious vulnerabilities. Such problems are exacerbated by the fact that middleboxes are often invisible to one or both endpoints, leading to a lack of *accountability*. A recent protocol, called mcTLS, pioneered accountability for proxies, which are authorized by the endpoints and given limited read/write permissions to application traffic.**

Unfortunately, we show that mcTLS is insecure: the protocol modifies the TLS protocol, exposing it to a new class of *middlebox-confusion attacks*. Such attacks went unnoticed mainly because mcTLS lacked a formal analysis and security proofs. Hence, our second contribution is to formalize the goal of accountable proxying over secure channels. Third, we propose a provably-secure alternative to soon-to-be-standardized mcTLS: a generic and modular protocol-design that carefully composes generic secure channel-establishment protocols, which we prove secure. Finally, we present a proof-of-concept implementation of our design, instantiated with unmodified TLS 1.3 draft 23, and evaluate its overheads.

***Keywords*-mcTLS, TLS 1.3, provable security**

I. Introduction

Internet protocols are largely designed around the end-to-end principle, which says that all application logic, except for the mundane activity of forwarding packets, should reside at the endpoints. However, a good portion of Internet traffic today passes through one or several active proxies or *middleboxes* that inspect, filter, and transform packets, based on dynamically configurable policies. These middleboxes include content delivery networks, personal and enterprise-level firewalls, compression proxies, malware scanners, parental-control content filters, and many other in-network functionalities that are considered desirable by client institutions, web servers, and network operators.

All of these functionalities require read access and/or write access to client-server traffic, and are consequently hindered by the prevalence of end-to-end encryption protocols like Transport Layer Security (TLS). To continue working, middlebox providers have resorted to a variety of *ad-hoc* techniques that enable them to decrypt TLS traffic between clients and servers. These techniques necessarily contradict the end-to-end security goals of TLS, by turning a well-studied two-party secure channel into a non-standard multi-party cryptographic protocol with unknown security properties. In fact, middleboxes often introduce new threats, since adversaries aiming to break secure connections can now attack not just endpoints but also the middleboxes. Our goal in this paper is to precisely identify these threats and to formally define the security goals of proxied TLS connections, in order to enable a rigorous evaluation of existing and new middlebox-based designs.

Content Delivery Networks. CDNs like Akamai and Cloudflare are good examples of widely-used active proxies. For example, Akamai owns 233,000 HTTP and HTTPS caching proxy servers in over 130 countries and within more than 1,600 networks around the world, serving 27% of the world's network traffic on behalf of many of the top websites. In October 2015, they estimated that 45% of their traffic was TLS-encrypted traffic [29] and this number has undoubtedly significantly increased since.

In order to serve TLS-traffic on behalf of a website, CDNs like Akamai need the website-owner to allow them to hold a valid X.509 certificate for the website's domain(s) and the associated private key on their behalf, essentially licensing the CDN to impersonate the website to connecting web-browsers. This form of delegation endows a high degree of trust in the CDN infrastructure, since a bug or attack on any of the CDN's "edge" servers around the world may allow an attacker to steal the website's private key or other sensitive user data (*e.g.*see the recent Cloudbleed bug in Cloudflare [35]). Websites can try to mitigate this risk by delegating only certain subdomains to the CDN, rather than the full web-applications, but this has limited effectiveness on websites where JavaScript from one subdomain is routinely loaded on another.

An alternative to the CDN-architecture above, denoted Keyless SSL, is offered as a premium service by the CDN called Cloudflare. Cloudflare only gets the X.509 certificate for a CDN-ed domain (and the public key within the cert), and Cloudflare's customers securely store the associated private keys on their own servers, only granting a limited key-usage API to the Cloudflare's proxy servers. This design requires a minor refactoring of the TLS handshake, but even this small change results in a new three-party protocol whose security guarantees are subtle. Indeed, recently, Bhargavan et al. [4]

DOI 10.1109/SP.2018.00021

demonstrated several vulnerabilities of Keyless SSL; they used a provable security approach to formally analyze it and proposed alternative designs that achieve stronger security goals, albeit with reduced performance.

Client-side firewalls and content filters. Firewalls are commonly used in enterprises and educational institutions to protect computers from malware and age-inappropriate content. To be able to inspect TLS traffic between machines inside the firewall and the open Web, these firewall proxies effectively mount a man-in-the-middle attack on the TLS connection by asking all clients to install a CA certificate that they can then use to issue certificates for any web server. This gross misuse of the public key infrastructure completely bypasses the end-to-end guarantees of TLS, and hence requires complete trust in the design and implementation of the proxy itself.

Worryingly, recent studies of such proxies uncovered a plethora of serious security issues, from inappropriate or inexistent certification validation and ciphersuite downgrades, to the execution of completely invalid TLS handshakes [15], [34]. A typical example of what can go wrong is the Superfish scandal of 2015 [18] where the private key of the CA certificate used by a client-side proxy was leaked, allowing attackers to impersonate any website to any client who used the proxy. More generally, client-side proxies that install root CA certificates are completely invisible to both the client and server, making it impossible for web browsers or websites to impose their own security policies for sensitive transactions, say between a client and a bank.

Fine-grained access control with mcTLS. A recent protocol called mcTLS [33] proposes a different design where all middleboxes are fully visible to both the client and the server, both of whom must agree on the read/write privileges of each middlebox before the middleboxes get to intercept the TLS connection. Hence, mcTLS offers fine-grained access control to both the client and server over all proxied TLS connections. To achieve this efficiently, the protocol significantly modifies the TLS handshake protocol, aggressively sharing messages and key material across multiple hops to perform several TLS handshakes at once.

Because of its excellent performance and support for fine-grained policies, mcTLS is being standardized within the ETSI standards organization as the default middlebox security protocol for network operators.[1] Consequently, the protocol is poised to have a big impact on practical middlebox designs. However, before the protocol is standardized and widely deployed, we believe it is important to formalize its security goals and verify that mcTLS meets them.

The authors of mcTLS do offer an informal security analysis [33] of the protocol, but without a formal proof. Indeed, as we show in the next section, certain intended use-cases of mcTLS are demonstrably insecure, which motivates the need for formal analysis, and provides a stark warning against tampering with existing protocols like TLS without a formal proof. This is particularly important for TLS 1.3, the new version of TLS which was designed to be provably secure and has received considerable attention from the academic community, but whose strong guarantees can still be completely broken by badly designed proxies.

Our approach and contributions. In this paper, we adopt a *provable security* approach to proxied TLS connections:

- We show that existing proxying mechanisms, including mcTLS, fail to ensure intuitive security notions of authentication, confidentiality, and integrity, even in common proxying scenarios.
- We provide (to our knowledge) the first fully formalized security definition for proxied TLS. We call this definition *authenticated and confidential channel establishment with accountable proxies* (ACCE-AP).
- We provide a modular protocol construction that only allows proxying with the full, explicit knowledge and consent of both partners (akin to mcTLS). Unlike mcTLS, our construction is provably secure and can be instantiated with any authenticated key-exchange (AKE) protocols which respect some reasonable conditions[2]. In particular, our design composes TLS connections in a way that allows us to rely on existing proofs of TLS, rather than prover our protocol's security from scratch.
- We describe a proof-of-concept implementation, showing how our new design can be deployed modularly on top of miTLS, a high-assurance TLS 1.3 library.

We provide a more extended review of related literature in Appendix A.

II. Active Proxy Architectures and their Flaws

The goal of an active proxy is to provide in-network functionality between a client and a server, while preserving (as much as possible) end-to-end confidentiality, authentication, and data integrity. Figure 1 depicts a typical scenario with one proxy. We note that the proxy may offer its services to many clients and servers, some of whom may be malicious or otherwise controlled by an adversary. Furthermore, we assume that the TLS connections between the client and middlebox, and between the middlebox and server are subject to standard network attacks.

A. Proxying Architectures

Existing proxying architectures can be broadly categorized in terms of which participants are aware of the proxying. We discuss two common setups below, before focusing on mcTLS, one of the most sophisticated and convincing proxying protocols designed to date.

[1]https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=52930

[2]On the one hand, this means that we can use TLS 1.3 as our building-block. On the other hand, we do not use any specific property of a given version of TLS 1.3, not in the construction nor in the proofs.

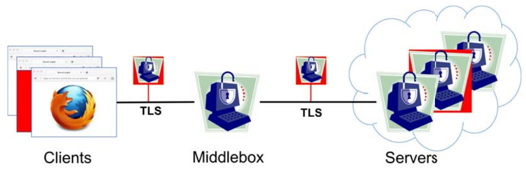


Figure 1: Active proxying

Invisible Proxies. Before encrypted connections became the norm, a variety of in-network proxies were used to provide compression, caching, load-balancing, and other bandwidth-saving or latency-reducing services. Once they were installed and configured, these proxies were *fully invisible* to both the client and server, since they could operate directly on unencrypted data. To continue working over TLS, these proxies need to be able to impersonate the client and the server to each other, a "feature" called *SSL Interception*. On the Web, clients are usually anonymous, but servers are identified by X.509 public-key certificates, and so such proxies need to have access to a valid certificate (and private key) for the server. This means that the proxy cannot be perfectly invisible to both parties. Either the proxy needs to be fully trusted by the server, who provisions it with a certificate, or it needs to be even more trusted by the client, who allows it to install a CA certificate that the proxy can use to impersonate any web server. We call the former design *client invisible* and the latter design *server invisible*.

Invisible proxies are attractive because they require little change to existing client-server deployment, but they are vulnerable to a number of attacks because the security of the TLS ecosystem relies on a careful collaboration between mechanisms implemented by both clients (web browsers) and servers (websites), and these mechanisms can be bypassed if one or both endpoints are unaware of the proxying. For example, modern browsers remember the certificates of important websites, through a technique called public-key pinning, and are able to enforce sophisticated certificate revocation policies through protocols like OCSP, but these protections no longer apply if the proxy uses its own certificate. Similarly, a browser or website may require a modern version of TLS with a strong ciphersuite, but the proxy may downgrade the connection to use a legacy version of the protocol. In addition, the high level of trust needed in the proxying software itself has been undermined by a series of flaws and attacks on middleboxes [15], [34], [18].

Accountable Proxies. The many vulnerabilities of invisible proxies stem from their adoption of ad hoc mechanisms based on misusing the public key infrastructure. An alternative is to design proxies that are not only visible, but request explicit authorization from one or both endpoints, and do not interfere with connections for which they do not have authorization. This would allow sensitive websites, like those for online banking, to forbid middleboxes and ensure that their clients' account details are not visible to any party other than the user's web browser. Such *accountable* designs may even become mandatory because of privacy laws like the recently adopted Data Protection Regulation (EU regulation 2016/679), which requires that personal information pertaining to any citizen may only be sent, handled, and stored with that individual's express consent.

Accountability allows network operators to reintroduce in-network functionality like caching and compression proxies over TLS without requiring full trust from clients and servers. For example, the Keyless SSL proposal from Cloudflare [4], and the related Lurk draft standard [30], enable server-visible caching proxies that require the proxy to contact a key server on every TLS session, allowing the key server to easily disable proxying based on a variety of server-side policies. These designs are not without their flaws, and alternative designs can offer even stronger and provably secure notions of accountability [4].

A new and radical proxy architecture, called mcTLS, offers even more fine-grained control over proxies' behaviors, by requiring both the client and the server to negotiate and agree upon the read-write access rights given to each intermediate proxy. In the rest of this section, we describe this important new protocol, curently undergoing standardization by ETSI, and evaluate its security. In particular, we describe several *middlebox confusion attacks* against mcTLS and use them to motivate a rigorous formal analysis of active proxying.

B. *The mcTLS protocol*

The mcTLS protocol uses the same message formats and cryptographic constructions as TLS 1.2 (hence the name) but is a completely new multi-party protocol that modifies TLS in significant ways. The protocol features a client C, a server S, and a number of middleboxes situated between them. The server and the middleboxes are provisioned with public-key certificates. Each middlebox is given read and/or write permission to various mcTLS *contexts*, which are portions of an application-data stream, such as HTTP bodies/headers. For example, a firewall middlebox may only have read access for a particular HTTP header, whereas a caching middlebox may have write access to the full HTTP response. To enforce these access rights, the client and server generate read and write keys for each context and deliver them to each authorized middlebox over point-to-point TLS-like channels.

Figure 2 depicts the mcTLS protocol when run for a single middlebox, denoted MW. The handshake consists of three interleaved TLS-DHE (ephemeral Diffie-Hellman) handshakes that share messages and key material: the primary handshake between the client and the server, and two intermediate handshakes between the client and middlebox, and middlebox and server. The server and middlebox authenticate themselves on all connections, but the client is unauthenticated. The client and server exchange nonces $\mathsf{N}_C, \mathsf{N}_S$ and Diffie-Hellman key shares g^x, g^y; the middlebox adds its own nonce N_{MW} and separate key shares g^z, g^t for use with the client and server.

At the end of the handshake, three sets of channel keys are computed using the TLS 1.2 key-derivation mechanism:

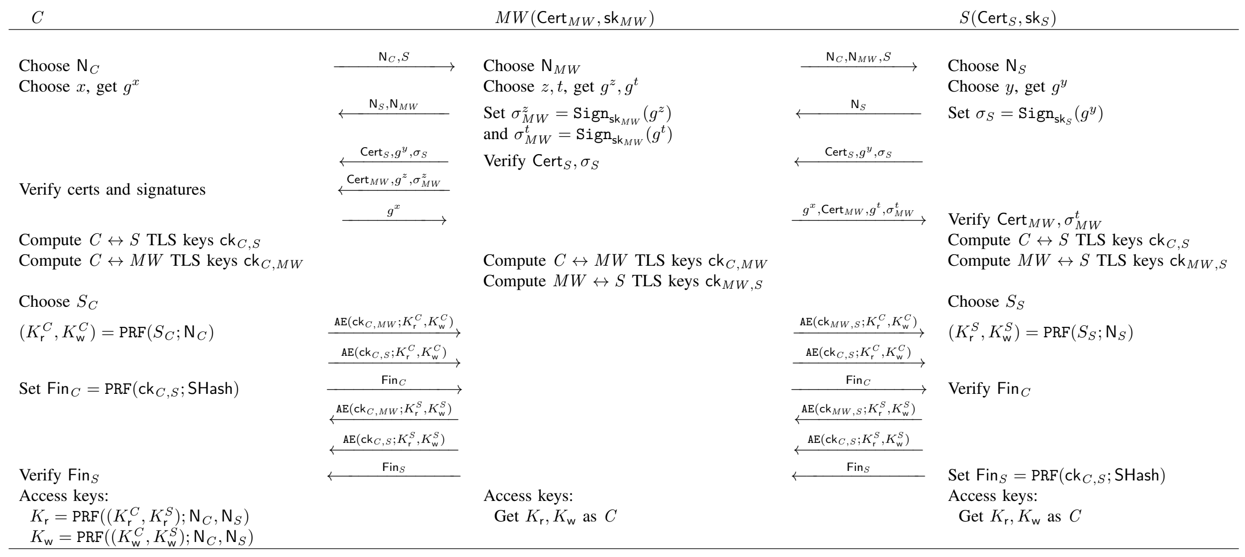

C
$MW(\mathsf{Cert}_{MW}, \mathsf{sk}_{MW})$
$S(\mathsf{Cert}_S, \mathsf{sk}_S)$
Choose N_C
Choose x, get g^x
N_C, S
Choose N_{MW}
Choose z, t, get g^z, g^t
$\mathsf{N}_C, \mathsf{N}_{MW}, S$
Choose N_S
Choose y, get g^y
$\mathsf{N}_S, \mathsf{N}_{MW}$
Set $\sigma^z_{MW} = \mathtt{Sign}_{\mathsf{sk}_{MW}}(g^z)$
and $\sigma^t_{MW} = \mathtt{Sign}_{\mathsf{sk}_{MW}}(g^t)$
N_S
Set $\sigma_S = \mathtt{Sign}_{\mathsf{sk}_S}(g^y)$
$\mathsf{Cert}_S, g^y, \sigma_S$
Verify $\mathsf{Cert}_S, \sigma_S$
$\mathsf{Cert}_S, g^y, \sigma_S$
Verify certs and signatures
$\mathsf{Cert}_{MW}, g^z, \sigma^z_{MW}$
g^x
$g^x, \mathsf{Cert}_{MW}, g^t, \sigma^t_{MW}$
Verify $\mathsf{Cert}_{MW}, \sigma^t_{MW}$
Compute $C \leftrightarrow S$ TLS keys $\mathsf{ck}_{C,S}$
Compute $C \leftrightarrow MW$ TLS keys $\mathsf{ck}_{C,MW}$
Compute $C \leftrightarrow MW$ TLS keys $\mathsf{ck}_{C,MW}$
Compute $MW \leftrightarrow S$ TLS keys $\mathsf{ck}_{MW,S}$
Compute $C \leftrightarrow S$ TLS keys $\mathsf{ck}_{C,S}$
Compute $MW \leftrightarrow S$ TLS keys $\mathsf{ck}_{MW,S}$
Choose S_C
$(K^C_\mathsf{r}, K^C_\mathsf{w}) = \mathtt{PRF}(S_C; \mathsf{N}_C)$
$\mathtt{AE}(\mathsf{ck}_{C,MW}; K^C_\mathsf{r}, K^C_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{C,S}; K^C_\mathsf{r}, K^C_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{MW,S}; K^C_\mathsf{r}, K^C_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{C,S}; K^C_\mathsf{r}, K^C_\mathsf{w})$
Choose S_S
$(K^S_\mathsf{r}, K^S_\mathsf{w}) = \mathtt{PRF}(S_S; \mathsf{N}_S)$
Set $\mathsf{Fin}_C = \mathtt{PRF}(\mathsf{ck}_{C,S}; \mathsf{SHash})$
Fin_C
Fin_C
Verify Fin_C
$\mathtt{AE}(\mathsf{ck}_{C,MW}; K^S_\mathsf{r}, K^S_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{C,S}; K^S_\mathsf{r}, K^S_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{MW,S}; K^S_\mathsf{r}, K^S_\mathsf{w})$
$\mathtt{AE}(\mathsf{ck}_{C,S}; K^S_\mathsf{r}, K^S_\mathsf{w})$
Verify Fin_S
Fin_S
Fin_S
Set $\mathsf{Fin}_S = \mathtt{PRF}(\mathsf{ck}_{C,S}; \mathsf{SHash})$
Access keys:
$K_\mathsf{r} = \mathtt{PRF}((K^C_\mathsf{r}, K^S_\mathsf{r}); \mathsf{N}_C, \mathsf{N}_S)$
$K_\mathsf{w} = \mathtt{PRF}((K^C_\mathsf{w}, K^S_\mathsf{w}); \mathsf{N}_C, \mathsf{N}_S)$
Access keys:
Get $K_\mathsf{r}, K_\mathsf{w}$ as C
Access keys:
Get $K_\mathsf{r}, K_\mathsf{w}$ as C

Figure 2: The mcTLS Handshake.

the client and server derive keys from g^{xy}, N_C, and N_S, the client and middlebox derive keys from g^{xz}, N_C, and N_{MW}, the middlebox and server derive keys from g^{yt}, N_{MW}, and N_S. These channel keys are then used to distribute context-specific read and write keys. The client and server generate independent context keys, say by applying a key derivation function to local temporary secrets (S_C, S_S), and deliver these keys to each other and to each authorized middlebox. After delivering the context keys, the two endpoints complete the handshake by exchanging Finished messages that contain MACs over the *entire transcript* as seen by each endpoints over the two sessions it runs, to ensure that none of the handshake messages has been tampered with.

Next, the client and server start exchanging application data; the data stream is divided into fragments, each fragment is labeled with a specific context and encrypted with the corresponding context key. Middleboxes that are authorized to read or write a particular context can then use the corresponding context keys to decrypt and/or reencrypt these packets, but not others.

The mcTLS protocol extends and deviates from the standard TLS 1.2 handshake in many ways. First, it adds a middlebox negotiation extension to the ClientHello and ServerHello messages, so that the client and server can agree upon the sequence of middleboxes they wish to use. Second, mcTLS reuses nonces and ephemeral Diffie-Hellman key shares between multiple connections, which is not recommended but is relatively harmless. Third, mcTLS only completes a full TLS handshake between the client and server; the client-middlebox and middlebox-server handshakes do not include any Finished messages, and so the middlebox cannot know whether the handshake has been tampered with. This is a more significant change to TLS, and as we shall see below, it leads to serious attacks on mcTLS. Fourth, mcTLS modifies the record-layer protocol to use context keys instead of channel keys for encryption. This is again an important change that exposes the protocol to record-layer attacks. Finally, to reduce server overhead, the mcTLS authors [33] propose a *client key distribution mode* where the client unilaterally generates and distributes context keys to the server and all middleboxes. As we shall see, this mode also weakens the protocol and enables attacks.

C. Middlebox Confusion Attacks on mcTLS

To better understand the security guarantees of mcTLS, let us first consider some of its intended use case scenarios.

In the first scenario, a client C uses a firewall middlebox MW to protect it from malware on the web. It requires MW to filter all its connections, including those to a trusted server S, and those to an attacker-controlled website $\mathscr{A}$. MW has read access to all incoming data and it imposes different filtering policies for different websites; in particular, it has stricter rules for $\mathscr{A}$ than for S. The adversary's goal is to bypass MW and deliver malware to C.

In the second scenario, a client C uses a caching proxy MW to speed up its accesses to the web. The proxy retrieves and caches static pages for a trusted server S as well as an untrusted server $\mathscr{A}$. When C connects to S via MW and asks for a specific resource, *e.g.*GET/login.html, MW looks in the cache it holds for S and immediately delivers the page if it is found. The attacker's goal is to read or write private user data (*e.g.*her password) that is sent between C and S.

Note that in both these scenarios, the middlebox MW makes important decisions based on the identity of the server. Consequently, if an attacker can confuse the middlebox about the identity of the server, it can fool the middlebox into making incorrect decisions. While mcTLS seeks to

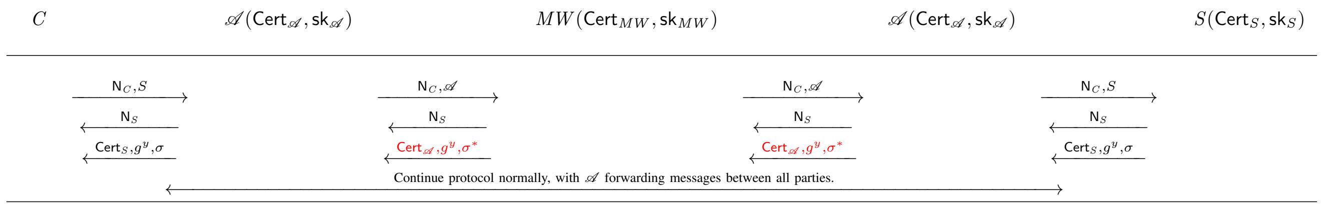

C
$\mathscr{A}(\mathsf{Cert}_{\mathscr{A}}, \mathsf{sk}_{\mathscr{A}})$
$MW(\mathsf{Cert}_{MW}, \mathsf{sk}_{MW})$
$\mathscr{A}(\mathsf{Cert}_{\mathscr{A}}, \mathsf{sk}_{\mathscr{A}})$
$S(\mathsf{Cert}_{S}, \mathsf{sk}_{S})$
N_C, S
N_S
$\mathsf{Cert}_S, g^y, \sigma$
$\mathsf{N}_C, \mathscr{A}$
N_S
$\mathsf{Cert}_{\mathscr{A}}, g^y, \sigma^*$
$\mathsf{N}_C, \mathscr{A}$
N_S
$\mathsf{Cert}_{\mathscr{A}}, g^y, \sigma^*$
N_C, S
N_S
$\mathsf{Cert}_S, g^y, \sigma$
Continue protocol normally, with $\mathscr{A}$ forwarding messages between all parties.

Figure 3: Unknown Key Share Attack on mcTLS Middleboxes.

protect endpoints against malicious middleboxes, the security analysis of the protocol appears to overlook attacks against middleboxes themselves, a category of attacks that we call *middlebox confusion attacks*.

Poisoning Caches with an Unknown Key Share Attack. In our first attack, a client C wants to communicate with a trusted server S through an honest middlebox MW. However, the network between C and MW is controlled by the adversary, which interferes with the mcTLS protocol, as depicted in Figure 3. Importantly, the attacker only modifies messages that are seen by MW; it then restores the correct messages and forwards them to C and S, so that they cannot detect this tampering.

When C sends its connection request (ClientHello) to MW, the attacker replaces the identity of S (typically indicated in the TLS-SNI extension) with its own identity $\mathscr{A}$. MW then initiates a connection with the attacker's server, and $\mathscr{A}$ forwards these handshake messages to S. When S returns its certificate and signature over its key share g^y, the attacker replaces them with its own certificate and signature over the same server key share. When MW forwards these messages to C, the attacker intercepts and replaces the certificate and signature with the original messages sent by S. The mcTLS handshake proceeds normally to conclusion, since the transcripts at the endpoints match. The transcript at the middlebox is quite different, but MW does not send or receive any Finished MACs in mcTLS, and so it cannot detect the attacker's tampering.

At the end of this handshake, the client and server correctly believe that they have a connection to each other via MW, but MW believes that it is proxying a connection between C and the attacker-controlled website $\mathscr{A}$. This is a form of *unknown key share* attack on the middlebox, and even though the attacker does not know any of the context keys, it can still mount a serious attack. If the middlebox is a caching proxy, it will now deliver to C content that was previously retrieved from the attacker's website, thereby allowing the attacker to inject data (and hence JavaScript) into connections between C and S. Hence, a network attacker can completely break the server authentication guarantees of caching proxies in mcTLS.

Bypassing Firewalls using Client Key Distribution. Our second attack is similar in structure to the previous scenario, but is exploitable in a different way. This time, the client C wishes to connect to the attacker's website $\mathscr{A}$ via a firewall proxy MW. The attacker intercepts the client's request and tampers with it to make MW believe that the C wishes to connect to the trusted server S instead. The attacker then completes the handshake with the client (using its own key shares). The connection with S cannot be completed, since the transcripts at C and S do not match, but if the protocol uses the client-key-distribution mode, it then unilaterally generates and sends context keys to $\mathscr{A}$ and MW.

At this point, the C correctly thinks it is connected to the attacker's server $\mathscr{A}$ via MW, but MW thinks it is proxying a connection between C and S. This is a *server impersonation* attack on the middlebox. It allows the attacker to inject malware to the client, which will not be filtered as strictly as it should be, because MW thinks that the message is being sent by the trusted server S. Hence, mcTLS with client key distribution cannot enforce site-specific firewalling policies.

Record-Layer Attacks. mcTLS modifies the TLS record protocol to enforce fine-grained access control, but the resulting protocol is too weak for many use-cases. For example, an attacker who controls the network between C and its firewall MW can always inject malware into C, bypassing the firewall. This is because mcTLS privileges the endpoints over the middleboxes, allowing them to read and write all contexts. So all the attacker has to do is to present an innoccuous data stream to MW and then it can decrypt, modify, and reencrypt data between MW and C.

As a more subtle example, consider a scenario where the attacker has obtained the reading key for some context, by compromising some middlebox, for instance. Since it does not have write access, the attacker should not be able to tamper with the data. However, since a reading context-key can be used both to verify and to create new MACs, the attacker can fool honest middleboxes into accepting tampered data, and then restore the original data before the endpoint sees it. Hence, even with read-only access, an attacker can inject data into the caches for trusted servers, or it can cause firewall protections to be bypassed.

Towards Provably Secure Accountable Proxies. The mcTLS protocol represents the most robust and flexible proxying proposal to date, and we believe that its approach is commendable. The middlebox confusion attacks described above can be prevented by using stronger variants of TLS, such as TLS 1.3 or TLS 1.2 with the session-hash extension. The record-layer attacks are harder to prevent without significant redesign. The main lesson from our attacks, however, is that the security guarantees of active proxying scenarios can be subtle and even well-designed proposals can benefit from rigorous formal analysis.

III. Defining ACCE-AP security

So, in this section, we formally capture the security guarantees we believe should be attained by secure and accountable proxying, in the shape of the following *authenticated and confidential channel establishment with accountable proxies* (ACCE-AP) notion.

Using ACCE. Traditional authenticated key-exchange (AKE) security models [2] aim to prove that obtained session keys are indistinguishable from random. This is a very strong, composable guarantee; pseudorandom keys may then be securely used by any symmetric-key primitive. By contrast, Authenticated and Confidential Channel Establishment (ACCE) security [20] relaxes this, focusing on the security of the channel obtained by using those keys. An ACCE-secure key-exchange protocol guarantees that the established keys are able to construct a secure channel. ACCE security was designed to capture the complexities of TLS 1.2, whose key-confirmation step acts as a distinguishing oracle for the pseudorandomness of the session keys, without damaging the security of the established channel [25], [8].

TLS 1.3 was designed to (provably) guarantee composable AKE security, unlike TLS 1.2. Indeed, the session keys in TLS 1.3 are indistinguishable from random, which, coupled with the security of the authenticated encryption algorithms, should yield strong, composable security. However, the composition of AKE and secure record-layer exchanges is not trivial [9]. As a result, proofs such as [13] cannot simply stop at the key-establishment step. We choose instead to use the ACCE definitions, which provides no composability, but –in turn– makes explicit the guarantees of confidentiality, authenticity, and integrity we may expect of the established channel.

We focus on accountable proxying over secure channels. A crucial aspect of this is indeed defining what security (in particular authenticity and integrity) can be guaranteed at the record layer. That is where the augmented risk of added middleboxes is most obvious. To highlight this, and to describe precisely what is lost by giving third parties access to encrypted traffic, we choose to rely on ACCE security. We review both AKE and ACCE terminology in the appendix.

On proxy visibility. One implicit assumption of mcTLS and of our work is that both the client and the server are aware, when starting the handshake, of all the proxies that can be found between them. We believe this is a necessary property when sensitive communication is exchanged between the client and the server. Indeed, a client might *e.g.*, be more cautious with the data it requests from a server if it knew that the traffic is proxied by a middlebox.

In practice, the client and server might not be aware of all the proxies connected to them, and especially, to their communication partners. However, it would suffice to ask those proxies to identify themselves (and request permissions) in a first protocol step, orthogonal to our design. No cryptographic requirements would be made of this step: *i.e.*, we authentication is needed. This will be taken care of during our protocol.

Notations. Following the approach of Bhargavan et al. [4], the two-party ACCE notion with mutual authentication will be denoted 2-ACCE, whereas for server-only authenticated handshakes we use the notation 2-SACCE. Like Bhargavan et al., we view the proxied handshake in the presence of (possibly multiple) middleboxes as a number of linked 2-party protocols that are executed in parallel.

A. *An intuition of the* ACCE-AP *model*

Tuples of 2-party protocols. Our protocol is run by several parties, which are either clients, or middleboxes, or servers. We view an n-party session (with one client, one server, and $n-2$ middleboxes) as a set of smaller, 2-party sub-protocols, in which instances of one party play the client or the server of a traditional 2-party handshake. This is defined as the *role* that the party plays in a 2-party execution.

Each party may run multiple concurrent executions of a 2-ACCE or 2-SACCE protocol: each protocol session is executed by a party *instance*. As a consequence, in an n-party handshake, various instances of the same party maybe executed at the same time: in some of these the instance will be a client, in others, a server. The m-th instance of party P_i is denoted π_i^m.

Although this may seem restrictive, it is more a matter of notation: indeed we choose to separate the communication between, say, the middlebox and the client, and the middlebox and the server because it is easier to keep track of it. We do assume that those instances may share state, which is certainly the case for the TLS handshake.

Instance partnering. Two-party handshakes use the notion of *partnering* or *matching conversation* to define which instances of two parties execute the protocol together. When moving from 2- to multi-party handshakes, a crucial notion that extends partnering is that of *session binding*, defining which 2-party handshakes take place as part of the same multiparty handshake. The binding is established through stored *local* information on direct communication partners, but also *global* information describing the configuration of the handshake, the parties acting as endpoints, the precise proxies used, and their access rights. In our model, all the parties involved in the handshake need to keep track of some cumulative *attributes*. Since the endpoints control the handshake, they also additionally need to store *master* parameters, such as the access rights of each middlebox. We refer to the endpoint instances that authenticate the master parameters by *master instances*.

Configurations and contexts. Session binding is defined in terms of a handshake *configuration*, which is a suite of party identities ordered from the client to the server. We always assume that the client remains anonymous in the handshake, *i.e.*, it always runs server-only authenticated handshakes. The middleboxes may play the role of either client or server, but always authenticate to their partner.

We divide each message into disjoint *contexts*, following mcTLS terminology. The list of all possible contexts is denoted Γ. Contexts can be defined, *e.g.*, as the concatenation of the message *type* (either request or response) and a message *fragment* (the header, the body, or a fragment thereof). Thus, one context could be *"request | header"*.

Each middlebox will have some *read*, *write*, or *none* access *permissions* on the defined contexts. We assume that entities which have write permissions for a given context may also read those contexts.

ACCE-AP **security.** We capture the security of the accountably-proxied secure-channel establishment protocols (called ACCE-security with accountable proxies – in short, ACCE-AP) in terms of the following requirements:

The *(S)ACCE security* of each independent, 2-party handshake, when the two partners are honest;

The *soundness of the configuration* in these 3 cases:

Fully honest. If all parties are honest, then all the instances are properly coupled, and all parties finish the configuration successfully, with all the appropriate keys.

Malicious MW. No collusion of malicious proxies learn anything about keys they have no right to.

Malicious endpoint. If an endpoint behaves maliciously w.r.t. the established configuration, contexts, or permissions, this will be detected by the remaining participants.

B. The ACCE-AP *model*

In what follows, we formalize configurations, permissions, and contexts as described above.

Oracles in ACCE-AP**.** Let k be an arbitrarily fixed integer. We consider AKE protocols executed between a client, a server, and up to $k-2$ middleboxes. More formally a *handshake configuration* $\mathsf{HConfig}$ is a list of parties $P_1, \ldots, P_k$, such that P_1 is a client, P_k is a server, and the remaining parties are middleboxes. The configuration is ordered so that conversation will be forwarded in increasing order of indices for request messages, and inversely for the response.

Moreover, in each handshake, each of the proxies is associated with a list of contexts and corresponding permissions, as formalized below. For any configuration $\mathsf{HConfig}$ of size k, its corresponding *contextual access list* $\mathsf{acl}_{\mathsf{HConfig}}$ is a hash-table of size $k-2$, indexed by *middleboxes* in that handshake. For a given proxy P_i, the entry in $\mathsf{acl}_{\mathsf{HConfig}}$ corresponding to P_i is a list of length $|\Gamma|$ which, for each context c, contains the permissions (*i.e.*, exactly one value amongst `none`, `read`, or`write`) given to P_i for that context c.

As explained, a proxied handshake consists of a number of parallel AKE sessions executed concomitantly. Each session is run between two instances, one for each of the two communicating parties. The number of executions per proxied handshake is protocol-specific: mcTLS for instance ran three sessions for the 3-party case, and six for the 4-party case. The multiple parallel sessions are also taken into account by our modification of the traditional $\mathsf{NewSession}$ oracle (presented in the Appendix).

Party attributes. Following the ACCE model, *parties* keep track of values such as their long-term secret keys, while party *instances* store session-specific values, *e.g.*, session and partner identifiers, the secret bits used by the Encrypt and Decrypt oracles, and established keys. These attributes are described in detail in Appendix C.

Instance-local ACCE-AP **attributes.** In addition to typical ACCE attributes, presented in the Appendix, the following attributes are specific to sole party instances:

$\pi_i^m.\mathsf{bid}$.The *binding identifier* $\pi_i^m.\mathsf{bid}$ of an instance π_i^m is a special session-identifier that ties π_i^m to a proxied handshake. Its value is set upon configuration acceptance. Note that $\pi_i^m.\mathsf{bid}$ could differ from the same instance's *session* identifier $\pi_i^m.\mathsf{sid}$[3].

$\pi_i^m.\beta$. The *configuration-acceptance* bit $\pi_i^m.\beta$ of instance π_i^m takes values in $\{0,1,\perp\}$. Initially, $\pi_i^m.\beta$ is set to $\perp$. If π_i^m accepts the configuration input to $\mathsf{NewSession}$, then $\pi_i^m.\beta$ is set to 1; in case of rejection, it is set to 0.

Cumulative ACCE-AP **attributes.** Cumulative attributes store the partnering, session, and binding information of all the instances of a given party taking part in a single handshake (which we call siblings).

$\pi_i^m.\mathsf{siblings}$. The *siblings* $\pi_i^m.\mathsf{siblings}$ of instance π_i^m are all the instances of P_i output by the $\mathsf{NewSession}$ oracle that generated π_i^m, including π_i^m itself.

Cumulative attributes are inherent to k-party handshakes; they are needed to established accountability.

$\pi_i^m.\mathsf{c.pid}$. The *cumulative partner identifier* $\pi_i^m.\mathsf{c.pid}$ of an instance π_i^m is a table of entries of the type $(\pi_i^n, \pi_i^n.\mathsf{pid})$, for each $\pi_i^n \in \pi_i^m.\mathsf{siblings}$.

$\pi_i^m.\mathsf{c.bid}$. The *cumulative binding identifier* $\pi_i^m.\mathsf{c.bid}$ of an instance is a tuple of binding identifiers $\pi_i^\ell.\mathsf{bid}$. Each identifier $\pi_i^\ell.\mathsf{bid}$ corresponds to one instance $\pi_i^\ell \in \pi_i^m.\mathsf{siblings}$, *i.e.*, $\pi_i^m.\mathsf{c.bid}$ is $(\pi_i^\ell.\mathsf{bid})_{\pi_i^\ell \in \pi_i^m.\mathsf{siblings}}$.

$\pi_i^m.\mathsf{c.config}$. The *cumulative configuration* $\pi_i^m.\mathsf{c.config}$ of a proxy instance π_i^m stores the configuration it has accepted for a given handshake. This attribute is *middlebox*-specific. Endpoints will store a *master* configuration attribute $\pi_i^m.\mathsf{m.config}$ (see below).

$\pi_i^m.\mathsf{c.perm}$. The *cumulative permission attribute* $\pi_i^m.\mathsf{c.perm}$ is a list of $|\Gamma|$ elements, each taking a single value in the set

[3]Both the partner and session identifiers are protocol-specific, and typically include public and private information that makes each session unique.

$\{\mathsf{none}, \mathsf{read}, \mathsf{write}\}$. This value represents the permission P_i has on the corresponding context. If $P_i \in \mathscr{C} \cup \mathscr{S}$, then all permissions are set to write.

$\underline{\pi_i^m.\mathsf{c.ctxt.keys}}$. The *cumulative context-key list* $\pi_i^m.\mathsf{c.ctxt.keys}$ consists of $|\Gamma|$ elements, each of them a tuple of values corresponding to the read and write keys for a given context (or $\perp$ if the value is unavailable). All sibling instances of a configuration-accepting party P_i share the same keys; if moreover $P_i \in \mathscr{C} \cup \mathscr{S}$, all the keys are non-$\perp$.

Master **ACCE-AP attributes.** Only stored by the handshake's endpoints, these attributes keep track of information on the handshake configuration and all the access-rights pertaining to the proxies. Thus, endpoints will be able to enforce accountability for all proxies.

$\underline{\pi_i^m.\mathsf{m.config}}$. The *master configuration attribute* is the endpoint equivalent to the middleboxes' $\pi.\mathsf{c.config}$. For an endpoint instance π_i^m accepting the configuration HConfig, $\pi_i^m.\mathsf{m.config}$ stores HConfig if it is the configuration for which NewSession was run.

$\underline{\pi_i^m.\mathsf{m.perm}}$. The *master permission-list attribute* $\pi_i^m.\mathsf{m.perm}$ stores, for each of the $k-2$ middleboxes input to NewSession, their permissions per context if they correspond to the $\mathsf{acl}_{\mathsf{HConfig}}$ value input to NewSession. Permissions are listed in the tabular style of $\mathsf{acl}_{\mathsf{HConfig}}$.

$\underline{\pi_i^m.\mathsf{m.bid}}$. The *master binding identifier* $\pi_i^m.\mathsf{m.bid}$ is a list of tuples indexed by parties P_j in the configuration, containing binding identifiers $\pi_j^\ell.\mathsf{bid}$ of instances π_j^ℓ involved in that handshake. We include P_i's values as well; in particular, $\pi_i^m.\mathsf{bid}$ is also included.

$\underline{\pi_i^m.\mu}$. The *master configuration-acceptance bit* $\pi_i^m.\mu$ of a master instance π_i^m, with $P_i \in \mathscr{C} \cup \mathscr{S}$ is a value $\{0, 1, \perp\}$. Initially $\pi_i^m.\mu$ is set to $\perp$, and may later change to 1 or 0, depending on whether that master instance has validated a given configuration or not.

$\underline{\pi_i^m.\mathsf{m.ctxt.keys}}$. The *master context key list* $\pi_i^m.\mathsf{m.ctxt.keys}$ is a list of $|\Gamma|$ pairs of read and write keys, one for each possible context.

Partnering. We extend 2-party partnering to our proxied case in Def. 1. For an instance π_i^m we require two sets: a set $\pi_i^m.\mathsf{PSet}$ of entities partnered to π_i^m (incl. P_i), and a set $\pi_i^m.\mathsf{InstSet}$ of instances partnered to π_i^m (incl. π_i^m).

Definition 1. ***(Partnering in*** **ACCE-AP.*)*** *Let π_i^m be an instance of P_i. We define $\pi_i^m.\mathsf{PSet}$ and $\pi_i^m.\mathsf{InstSet}$ as:*

If $P_i \in \mathscr{C} \cup \mathscr{S}$, $\pi_i^m.\mathsf{InstSet}$ contains all instances π_j^n such that $\pi_j^n.\mathsf{bid} \in \pi_i^m.\mathsf{m.bid}$ (incl. π_i^m). The set $\pi_i^m.\mathsf{PSet}$ includes all P_j with instances in $\pi_i^m.\mathsf{InstSet}$ (incl. P_i).

Let $P_i \in \mathscr{MW}$. Find the unique $\pi_j^n.\mathsf{m.bid}$ of a party $P_j \in \mathscr{C} \cup \mathscr{S}$ such that $\pi_i^m.\mathsf{bid} \in \pi_j^n.\mathsf{m.bid}$. The set $\pi_i^m.\mathsf{InstSet}$ is equal to the set $\pi_j^n.\mathsf{InstSet}$ (computed as in the previous bullet point), and $\pi_i^m.\mathsf{PSet} := \pi_j^n.\mathsf{PSet}$.

Correctness. We cannot limit the correctness of proxied handshakes to the correctness of partnering and keys. Indeed, we will also need to account for the aspects of session binding, cumulative and master configurations, and proxy permissions.

Definition 2. ***(Correctness).*** *Consider a handshake executed for a configuration* HConfig *and a contextual access list* $\mathsf{acl}_{\mathsf{HConfig}}$. *Consider the parties $P_i \in$ HConfig and their instances in this handshake. The following conditions must hold simultaneously:*

A. ***Partnering correctness.*** *This is three-fold:*

Instances. *For each instance π_i^m, there must exist an instance of $\pi_i^m.\mathsf{pid}$ whose session identifier* sid *equals $\pi_i^m.\mathsf{sid}$ (the instance's partner is in that handshake).*

Parties. *All instances of P_i correctly accumulate the partner- and session-information of their siblings (cumulative attributes are consistent w.r.t. each sibling's partnering and session information).*

Configuration. *Consider a* NewSession *query outputting instances π_j^n, including master instances π_i^m of endpoints P_i (client or server), and middlebox instances π_k^ℓ of middleboxes.*

We require: (i) All instances accept partner authentication and the handshake configuration; (ii) all accepting instances store the same configuration; (iii) the master binding identifier *of master instances π_i^m includes exactly twice the* binding identifier *$\pi_j^n.\mathsf{bid}$ of each instance π_j^n output by the same* NewSession *query (this includes the binders of P_i)*[4].

B. ***Access-control correctness.*** *We require the consistency of permissions allowed by the endpoints to all middleboxes. Consider a* NewSession *query yielding a handshake for which the instances complete in an accepting state. The following must hold simultaneously:*

Master context. *The master permission set $\pi_i^m.\mathsf{m.perm}$ of both endpoints coincides to $\mathsf{acl}_{\mathsf{HConfig}}$.*

Middlebox context. *The permissions for the handshake stored by each proxy, stored in $\pi_i^m.\mathsf{c.perm}$, must be consistent with those given in $\mathsf{acl}_{\mathsf{HConfig}}$.*

C. ***Key correctness.*** *Partnered instances are also required to compute the same channel keys, and all read/write keys for a given permission must coincide.*

Channel keys. *Instances with the same session identifier compute the same channel key $\pi_i^m.\mathsf{ck}$.*

Master keys. *All master instances compute the same* master *context keys.*

Middleware keys. *Each middlebox stores keys consistent with the permissions and context keys of master instances.*

ACCE-AP **security.** We define security in terms of security games played by an adversary against a challenger. The adversary will have access to a number of oracles including

[4]Each binder is stored twice because our handshakes consist of tuples of 2-party handshakes.

traditional 2-ACCE oracles (described in the Appendix). However, we modify the NewSession oracle to take into account the transition from 2 to k parties:

NewSession(HConfig, $\mathsf{acl}_{\mathsf{HConfig}}$). The query takes as input a handshake configuration HConfig and a contextual access list $\mathsf{acl}_{\mathsf{HConfig}}$. The output is a list of k sets of tuples $(\pi_i^j, \pi_i^j.\rho, \pi_i^j.\mathsf{pid})$. Each set $\{(\pi_i^j, \pi_i^j.\rho, \pi_i^j.\mathsf{pid})\}_{j=1}^{\ell_i}$ corresponds to a party P_i in the HConfig. Here $\ell_i \geq 1$ is public, but not adversarially-fixed; it depends on the protocol and number of participants[5]. Each set $\{(\pi_i^j, \pi_i^j.\rho, \pi_i^j.\mathsf{pid})\}_{j=1}^{\ell_i}$ is a series of oracle-instances π_i^j (with roles ρ and partner identifiers pid) of the party P_i.

The remaining, ACCE-like oracles we use are:

Send. The adversary uses this oracle to send a message to a given, existent instance, forwarding its reply.

Reveal. This oracle allows the adversary to learn the channel keys of a given party instance.

Corrupt. By this oracle, the adversary corrupts either middleboxes or servers, obtaining their long-term keys.

Encrypt. This is a left-or-right oracle, which allows the adversary to encrypt one of two possible messages of equal length, depending on a hidden bit b.

Decrypt. This oracle allows the adversary to decrypt only ciphertexts not returned by Encrypt.

The security properties. We define the security of a proxied handshake in terms of three properties: instance entity-authentication, instance channel-security, and configuration soundness. The first two are backward compatible with traditional 2-party (S)ACCE models, with a difference in the partnering. In the ACCE-AP model, each instance has more partners, none of which can be corrupted.

Each game will begin with a setup phase, in which all the $\mathsf{n_P}$ parties are instantiated, with corresponding long-term keys and certificates, returning the public parameters to the adversary $\mathscr{A}$. The adversary's winning advantage is quantified over the randomness of all participants.

Definition 3. ***(Entity Authentication).*** *In the* entity authentication *game, after setup, $\mathscr{A}$ is given arbitrary access to the new* NewSession *oracle above, and also to* Send, Corrupt, *and* Reveal. *Finally $\mathscr{A}$ halts, and* wins *if there exists an instance π_i^m such that all the following conditions occur:*

– π_i^m has ended in an accepting state: $\pi_i^m.\alpha = 1$, with partner $\pi_i^m.\mathsf{pid} = P_j \in \mathscr{MW} \cup \mathscr{S}$;

– No party in π_i^m.PSet (partnered to π_i^m) is corrupted;

– There exists no session π_j^n partnered with π_i^m ($\pi_j^n \notin \pi_i^m$.InstSet).

The adversary's advantage is its success probability.

Definition 4. ***(Channel Security).*** *For* channel security, *after setup, $\mathscr{A}$ is given access to the new* NewSession *oracle and to* Send, Corrupt, Reveal, Encrypt, *and* Decrypt. *Whenever a new instance π_i^m is created via* NewSession, *the attribute storing the secret bit used in* Encrypt, *denoted π_i^m.b, is chosen uniformly at random. Finally, $\mathscr{A}$ halts, outputting a tuple consisting of an instance π_i^m for $P_i \in \mathscr{C}$ and a guess-bit d. We say $\mathscr{A}$ wins if the following conditions occur simultaneously:*

– $\pi_i^m.\mathsf{b} = d$;

– No Corrupt *query was made on any party in π_i^m.PSet;*

– No Reveal *query is made on instances in π_i^m.InstSet.*

$\mathscr{A}$'s advantage is defined as $|p_{\mathscr{A}} - 1/2|$, where $p_{\mathscr{A}}$ is $\mathscr{A}$'s success probability.

Configuration soundness. We now describe a new security property capturing configuration soundness with respect to the input of NewSession. *Configuration soundness* has three components, formalized separately: (1) the soundness of the configuration w.r.t. fully-honest partners; (2) the soundness of the access-control-key distribution in the presence of corrupted middleboxes (and honest endpoints); (3) the soundness of the key-distribution even for a corrupted endpoint. We do not consider security against a collusion between endpoints and middleboxes: as soon as such a collusion occurs, we can guarantee practically no security for the middleboxes situated between the two colluding partners. Instead, the security level becomes somewhat equivalent to that of a smaller handshake, in which the colluding middlebox acts as the endpoint.

The three games begin with the setup described above. Then, $\mathscr{A}$ is given access to the new NewSession oracle and also to Send, Corrupt, Reveal, Encrypt, and Decrypt.

Definition 5. ***(Config. soundness: partnering).*** *The game is played as above. The adversary ends by outputting a master instance π_i^m, with $P_i \in \mathscr{C}$, ending in an accepting state with respect to its configuration,* i.e., *$\pi_i^m.\mu = 1$. Let $\pi_j^n \in \pi_i^m$.InstSet be such that $P_j \in \mathscr{S}$, $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid}$ and $\pi_j^n.\mu = 1$. We write* HConfig *for the configuration input to the* NewSession *query that outputs π_i^m. The adversary wins if the following conditions occur simultaneously.*

I. No Corrupt *query was made on parties in π_i^m.PSet;*

II. No Reveal *query is made on instances in π_i^m.InstSet.*

III. One of the following conditions occurs:

a. The master configuration π_i^m.m.config $\neq \pi_j^n$.m.config, or π_i^m.m.config $= \pi_j^n$.m.config $\neq$ HConfig;

b. The cumulative configuration of middlebox instances $\pi_k^\ell \in \pi_i^m$.InstSet differs from π_i^m.m.config or HConfig;

c. There is an instance $\pi_k^\ell \in \pi_i^m$.InstSet such that π_k^ℓ.bid $\notin \pi_i^m$.m.bid;

d. There is an instance $\pi_k^\ell \in \pi_i^m$.InstSet s.t. $\pi_k^\ell.\beta = 0$.

Condition (III.d) sometimes implies (III.c): it does for our protocol. However, this is not always true: the binding of all instances $\pi_k^\ell \in \pi_i^m$.InstSet could be correct, but there may exist an instance $\pi_k^\ell \in \pi_i^m$.InstSet that rejects that configuration, that is $\pi_j^n.\beta = 0$.

[5]If $\ell_i = 1$, the output coincides with what would be returned by the standard NewSession query for each participant P_i.

Definition 6. ***(Config. soundness: malicious*** MW***).*** *The game is played as above. The adversary will output, in a test phase, a tuple consisting of an instance* π_i^m *with* $\pi_i^m.\mu = 1$ *and a context* ctxt*, such that* $P_i \in \mathscr{C}$ *and there exists an instance* $\pi_j^n \in \pi_i^m.\mathsf{InstSet}$ *such that* $P_j \in \mathscr{S}$ *and* $\pi_i^m.\mathsf{sid} = \pi_j^n.\mathsf{sid}$*. The challenger will output either the true context key* K_{ctxt} *in* $\pi_i^m.\mathsf{m.ctxt.keys}$ *or a random key of the same length, according to a hidden bit b. Then* $\mathscr{A}$ *may continue querying oracles, finally outputting a bit d, winning if the following occurs simultaneously:*

– $d = b$*;*
– All instances $\pi_i^{\cdot} \in \pi_i^m.\mathsf{InstSet}$ *end in an accepting state with respect to the configuration,* i.e.*,* $\pi_i^{\cdot}.\beta = 1$*;*
– No Corrupt *query was made on any* $P_k \in \pi_i^m.\mathsf{PSet}$ *s.t.* ctxt *is registered as a permission for* P_k *in* $\mathsf{acl}_{\mathsf{HConfig}}$*;*
– No Reveal *query is made on instances in* $\pi_i^m.\mathsf{InstSet}$.

Finally, we guarantee that honest middleboxes accepting the configuration cannot distinguish from random the keys they are not entitled to, even if one endpoint is malicious.

Definition 7. ***(Config. soundness: malicious endpoint).*** *The game is played as above, and* $\mathscr{A}$ *eventually halts. It wins if there exists a client or server instance* π_i^m *storing a configuration* $\pi_i^m.\mathsf{m.config}$ *s.t. the following conditions apply simultaneously.*

– All instances $\pi_j^{\cdot}$ *for* $P_j \in \mathscr{MW} \cap \pi_i^m.\mathsf{PSet}$ *accepted the configuration;*
– Parties $P_j \in \mathscr{MW} \cap \pi_i^m.\mathsf{PSet}$ *are uncorrupted;*
– No Reveal *query is made on instances in* $\pi_i^m.\mathsf{InstSet}$.
– One of the following two conditions holds:

- *If* Corrupt *was queried on a server* $P_k \in \{\mathscr{S} \cap \pi_i^m.\mathsf{m.config}\}$ *(with* $P_k \neq P_i$*), then the following holds simultaneously: (a)* $P_i \in \mathscr{C}$*; (b) there are instances* $\pi_i^\ell, \pi_k^s \in \pi_i^m.\mathsf{InstSet}$ *with* $\pi_i^\ell.\mathsf{sid} = \pi_k^s.\mathsf{sid}$*; (c) all instances* $\pi_i^t \in \pi_i^m.\mathsf{InstSet}$ *ended in an accepting state w.r.t. the configuration,* i.e.*,* $\pi_i^t.\mu = 1$*; (d) there exists* $P_j \in \mathscr{MW} \cap \pi_i^m.\mathsf{PSet}$ *and some instance* $\pi_j^z \in \pi_i^m.\mathsf{InstSet}$ *with* $\pi_i^\ell.\mathsf{m.perm} \neq \pi_j^z.\mathsf{c.perm}$.
- *If no* Corrupt *query is made on the server* $P_k \in \{\mathscr{S} \cap \pi_i^m.\mathsf{m.config}\}$*, then the following holds simultaneously: (a)* $P_k = P_i$ *(the party* P_i was *the handshake server); (b) all instances* $\pi_i^z \in \pi_i^m.\mathsf{InstSet}$ *accepted the configuration (*$\pi_i^z.\mu = 1$*); (c) there exists* $P_j \in \mathscr{MW} \cap \pi_i^m.\mathsf{PSet}$ *and an instance* $\pi_j^t \in \pi_i^m.\mathsf{InstSet}$ *s.t.* $\pi_i^m.\mathsf{m.perm} \neq \pi_j^t.\mathsf{c.perm}$.

IV. An ACCE-AP-Secure Design

We now propose a protocol that is provably ACCE-AP secure. In fact, we introduce a generic *design*, to be instantiated with one or multiple accountable proxies. For one middlebox, this yields Figure 4; a 2-middlebox case is presented in the full version. In Appendix B, we also depict how to optimally apply our design to TLS 1.3.

A. The ACCE-AP*-secure Protocol Construction* Π

We denote by Π a generic secure-channel establishment protocol with accountable proxies, like the one in Figure 4. We use a modular construction, with a careful composition of independent, unmodified 2-party protocols guaranteeing slightly more than 2-party SACCE and respectively ACCE-security.

Our protocol consists of three phases: 2-ACCE, binding, and access-control.

2-ACCE Phase. This phase consists of running a number of parallel 2-SACCE and ACCE sessions, thus establishing channel and *exported* keys. Although not standardly output by AKE protocols, exported keys feature in many real-world AKE protocols, including TLS 1.3 (which standardizes an exporter secret)[6].

Each party in the protocol starts the following instances:

Transport instances. These instances are run by each entity with its direct neighbor(s): the endpoints only open one such instance, whereas each middlebox opens two. After successful completion, the transport sessions are used to securely exchange binding and access-control information. They also serve as an outer channel for record-layer message-exchanges.

Master instance. Only the client and the server open one such instance (irrespective of how many proxies take part in the handshake). In this session, the configuration and access control data are agreed upon, and the read/write keys are computed.

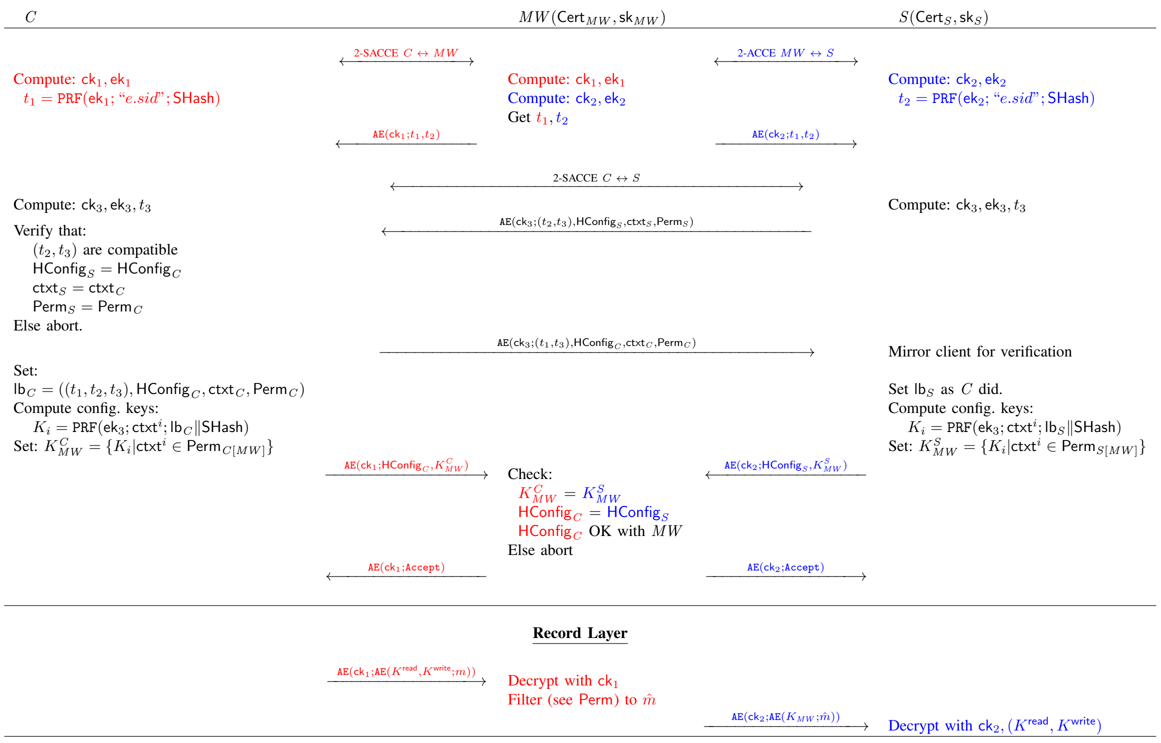
Key-transfer instance. Finally, each endpoint must open one instance to each middlebox it is not directly connected to via a transport session. Key-transfer instances are used to provide access-control keys and confirm the configuration with the middlebox so that the latter may accept or reject the handshake.

For the 1-middlebox case in Figure 4, the key transfer is done in the transport sessions: the client, the middlebox, and the server each open precisely two instances, yielding two transport sessions and the master session. For two middleboxes, each party opens three instances: the endpoints create one transport instance (which also serves to transfer keys to the closest proxy), one key-transfer instance (to the farthest middlebox), and the master instance. Each proxy opens two transport sessions (one to the neighboring endpoint, one to the second middlebox) and one key-transfer instance (to the non-neighboring endpoint).

Note that each instance will first need to finish in an accepting state with respect to authentication before the binding phase proceeds.

Binding Phase. We bind sibling instances together by using the exported keys output in each of the parallel sessions.

[6]The TLS 1.2 key exporter construction [11] is insufficient: we will need our exported keys to be unique (for instance, they could depend on the full session hash, not just the nonces).



C
$MW(\mathsf{Cert}_{MW}, \mathsf{sk}_{MW})$
$S(\mathsf{Cert}_S, \mathsf{sk}_S)$
2-SACCE $C \leftrightarrow MW$
2-ACCE $MW \leftrightarrow S$
Compute: $\mathsf{ck}_1, \mathsf{ek}_1$
$t_1 = \mathtt{PRF}(\mathsf{ek}_1; \text{“}e.sid\text{”}; \mathsf{SHash})$
Compute: $\mathsf{ck}_1, \mathsf{ek}_1$
Compute: $\mathsf{ck}_2, \mathsf{ek}_2$
Get t_1, t_2
Compute: $\mathsf{ck}_2, \mathsf{ek}_2$
$t_2 = \mathtt{PRF}(\mathsf{ek}_2; \text{“}e.sid\text{”}; \mathsf{SHash})$
$\mathtt{AE}(\mathsf{ck}_1; t_1, t_2)$
$\mathtt{AE}(\mathsf{ck}_2; t_1, t_2)$
2-SACCE $C \leftrightarrow S$
Compute: $\mathsf{ck}_3, \mathsf{ek}_3, t_3$
Compute: $\mathsf{ck}_3, \mathsf{ek}_3, t_3$
$\mathtt{AE}(\mathsf{ck}_3; (t_2, t_3), \mathsf{HConfig}_S, \mathsf{ctxt}_S, \mathsf{Perm}_S)$
Verify that:
(t_2, t_3) are compatible
$\mathsf{HConfig}_S = \mathsf{HConfig}_C$
$\mathsf{ctxt}_S = \mathsf{ctxt}_C$
$\mathsf{Perm}_S = \mathsf{Perm}_C$
Else abort.
$\mathtt{AE}(\mathsf{ck}_3; (t_1, t_3), \mathsf{HConfig}_C, \mathsf{ctxt}_C, \mathsf{Perm}_C)$
Mirror client for verification
Set:
$\mathsf{lb}_C = ((t_1, t_2, t_3), \mathsf{HConfig}_C, \mathsf{ctxt}_C, \mathsf{Perm}_C)$
Compute config. keys:
$K_i = \mathtt{PRF}(\mathsf{ek}_3; \mathsf{ctxt}^i; \mathsf{lb}_C \| \mathsf{SHash})$
Set: $K^C_{MW} = \{K_i | \mathsf{ctxt}^i \in \mathsf{Perm}_{C[MW]}\}$
Set lb_S as C did.
Compute config. keys:
$K_i = \mathtt{PRF}(\mathsf{ek}_3; \mathsf{ctxt}^i; \mathsf{lb}_S \| \mathsf{SHash})$
Set: $K^S_{MW} = \{K_i | \mathsf{ctxt}^i \in \mathsf{Perm}_{S[MW]}\}$
$\mathtt{AE}(\mathsf{ck}_1; \mathsf{HConfig}_C, K^C_{MW})$
$\mathtt{AE}(\mathsf{ck}_2; \mathsf{HConfig}_S, K^S_{MW})$
Check:
$K^C_{MW} = K^S_{MW}$
$\mathsf{HConfig}_C = \mathsf{HConfig}_S$
$\mathsf{HConfig}_C$ OK with MW
Else abort
$\mathtt{AE}(\mathsf{ck}_1; \mathtt{Accept})$
$\mathtt{AE}(\mathsf{ck}_2; \mathtt{Accept})$
Record Layer
$\mathtt{AE}(\mathsf{ck}_1; \mathtt{AE}(K^{\mathsf{read}}, K^{\mathsf{write}}; m))$
Decrypt with ck_1
Filter (see Perm) to $\hat{m}$
$\mathtt{AE}(\mathsf{ck}_2; \mathtt{AE}(K_{MW}; \hat{m}))$
Decrypt with ck_2, $(K^{\mathsf{read}}, K^{\mathsf{write}})$

Figure 4: Our generic, ACCE-AP-secure protocol Π, illustrated with a single middlebox

Each session's *binding identifier* is output by a pseudorandom function (like HKDF) keyed with an exported key, on input a label concatenated with that session's hash. It is essential for configuration soundness that the middlebox and server can verify that they are talking to the same client (since the client instances have so far been independent). We force the client to prove its knowledge of its instance binders to the server, and give it the binder for the remaining sessions; this will enable the two endpoints to agree on configuration and access-control details for that handshake. Both endpoints must agree on a configuration and an access control list before proceeding to the next phase.

In Figure 5, the binding identifiers are the values t_1, t_2, t_3, computed from the exported keys and the transcript hashes, as the output of a PRF instance, *i.e.*, $t_i = \mathtt{PRF}(\mathsf{ek}_i; \text{“}e.sid\text{”}; \mathsf{SHash})$. Each values t_i $(i \in \{1, 2, 3\})$ is stored individually in an π.bid, while π.m.bid stores all three of them.

Access-Control Phase. Finally, the endpoints each compute the access control keys for the handshake by running the same PRF as before, keyed with the exported keys of the master session, on input a label describing the context, and the transcript hash. More precisely, for a context ctxt^i and permissions a, the label lb^P_i is a string $\mathsf{ctxt}^i | a$. The access control keys are computed as $K_i = \mathtt{PRF}(\mathsf{ek}_3; \mathsf{ctxt}^i; \mathsf{lb}^P_i \| \mathsf{SHash}))$. Each middlebox receives the handshake configuration, one set of keys from the client, and one from the server, over a direct, secure channel (a key-transfer or a transport session). The middlebox verifies that its view is consistent with the received configurations, and that the keys sent by the endpoints coincide. If either verification fails, the middlebox *rejects the configuration*. Otherwise, it accepts the configuration and sends a message to that effect to the endpoints. Only then do the endpoints end in an accepting state, as well.

B. Security analysis

We first discuss the security assumptions we make on our primitives. Then, we state the security of our generic design for accountable proxying, given in Figure 4; due to lack of space, we leave the full proofs of these to the full version.

Security assumptions. The security of our protocol will require a 2-SACCE secure authenticated key-exchange (AKE) protocol Π_1 between the client and the middlebox, a 2-ACCE secure AKE protocol Π_2 between the middlebox and the server, and a 2-SACCE secure AKE protocol Π_3 between the client and the server. Notably, Π_1 and Π_3 offer unilateral authentication, whilst Π_2 is mutually authenticated.

For $i \in \{1, 2, 3\}$, let Ψ_i denote the extension of Π_i to a protocol $\Psi_i(\Pi_i)$ that runs Π_i, outputting ck, but in addition outputs an export key ek. We require that the export keys output $\Psi_i(\Pi_i)$ be indistinguishable from random, and in addition, that no PPT adversary (with access to the usual 2-party oracles NewSession, Send, Reveal, Corrupt) can find two non-partnered sessions for which the output export keys are identical, *even* if $\mathscr{A}$ can corrupt all servers (for the SACCE protocols) and all parties (for the ACCE protocol). We call the latter property *pseudo-uniqueness of* ek.

Our main reason for this assumption on the export keys is our notion of binding. We need to ensure that each configuration yields a unique set of binders even when a legitimate party in the handshake misbehaves. Such a condition helps us bypass *e.g.*attacks of the triple-handshake kind, like those on mcTLS shown in Section II-C. Two important aspects should be mentioned, regarding the pseudo-uniqueness assumption: (1) it is non-standard in AKE security (though interestingly it holds for TLS 1.3); (2) we do not require this same condition for the channel keys ck.

We do not demand pseudo-uniqueness of channel keys since they play no part in binding. As we do not consider resumption, or other short-cuts needing stronger keys, we only need channel keys be pseudo-random.

The ACCE-AP-security of the 1-middlebox-version of Π. We provide two security theorems for the multicontext handshake proposed in Figure 4: Theorem 1 captures entity authentication and channel security, and Theorem 2 captures configuration soundness (in its three aspects). The proofs are described in the full version.

Theorem 1. *Let Π be protocol in Figure 4, for which Π_1, Π_2, Π_3 and Ψ_1, Ψ_2, Ψ_3 are defined as above. If Π_1, Π_3 are 2-SACCE secure, and Π_2 is 2-ACCE secure, then:*

– *Π guarantees ACCE-AP entity authentication and channel security;*

Theorem 2. *Let Π be the protocol in Fig. 4, for which Π_1, Π_2, Π_3 and Ψ_1, Ψ_2, Ψ_3 are defined as above. If: (1) Π_1, Π_3 are 2-SACCE secure and Π_2 is 2-ACCE secure; (2) Ψ_1, Ψ_2, Ψ_3 extend Π_1, Π_2, Π_3 to also yield export keys ek which are indistinguishable from random and pseudo-unique; (3) the KDF used to compute t_1, t_2, t_3 is a secure PRF; (4) the hash function used for the recurring session hash is collision-resistant, then:*

– *Π is configuration sound w.r.t. partnering, malicious middleboxes, and malicious endpoints.*

These theorems informally guarantee 4 properties: a network adversary cannot convince an (honest) party it is part of a different configuration than is actually the case; all access-control keys are correctly distributed (middleboxes get the keys they are entitled to, and no collusion of malicious middleboxes can learn anything about keys it is not entitled to know); middleboxes can detect malicious behavior in one endpoint (with respect to the permissions); no network adversary can learn anything about the messages sent at the record layer of our multicontext handshake.

The security of the handshake, however, makes no guarantee w.r.t. the integrity and authenticity of messages sent at the proxied record layer. The secure-channel statement guarantees the integrity and authenticity of messages, but only within a two-party session. To capture integrity on the end-to-end record layer, we need to consider the *functionalities* and *permissions* of the middleboxes. We do so next.

Record-layer security. For a middlebox MW and a message fragment m (as given by the division into contexts), let $m' = MW(m)$ be the message m' obtained by the honest modification of m by MW. Note that m' could be the same as m, a correctly formatted message other than m, or it could also be an error message $\perp$.

For the one-middlebox case, we can prove the following result w.r.t. record-layer security. The case of several middleboxes will be discussed in the full paper, since it presents more problems.

Theorem 3. *For any handshake of the protocol in Figure 4, for all instances ending in an accepting state w.r.t. the configuration, for all contexts, consider a message-fragment m sent from one endpoint (which we will call* sender*) to the other endpoint (here called the* receiver*). Assume that the receiver receives a ciphertext c^*, for which it accepts the authentication of the message, decrypting it to a message fragment m^*. Then:*

– *If the middlebox is corrupted, but has read-only permissions for the context corresponding to fragments m, m^*, then $m^* = m$.*

– *If the sender is corrupted, but the receiver and middlebox are honest, then $m^* = MW(m)$.*

We can make no integrity and authenticity statement for a corrupted middlebox with write access. However, this is a flaw inherent to using read and write keys. The second guarantee in Theorem 3 is something no SSL inspection method, including mcTLS, has so far provided. Indeed, our double-encryption thwarts the middlebox-bypass attack we presented in Section II-C. This is a useful guarantee, as it ensures that honest proxies always perform their due tasks on messages sent to them, even by a dishonest endpoint.

Enhancing integrity. We could also make this integrity guarantee stronger, and ensure, like mcTLS, that a message that is sent unchanged from the honest sender to the honest receiver can be distinguished from a message that modified by a potentially-malicious middlebox with write access. This is easily done by adding a MAC, under, *e.g.*, a key derived from the export key of the master session under a new label. At the record layer, instead of just sending an authenticated encryption under the read and write key of a message, we could also add a MAC of that same message computed with the write key for that fragment. We choose to avoid this overhead, since the guarantees we can offer for the record layer are anyway limited.

C. Prototype Implementation

We built a proof-of-concept implementation of the ACCE-AP-Π protocol instantiated with TLS 1.3, as per depicted in Fig. 5. Our implementation is based on the miTLS [7] library, an implementation of TLS 1.3 (draft 23) in F* [37]. We prefer to use miTLS over more mainstream implementations, such as OpenSSL, for several reasons:

- miTLS is written in a modular way, with well-contained local state in the various modules (*e.g.*, the key schedule, negotiation module, handshake state-machine); this simplifies the writing of protocol extensions, in particular when multiple handshakes must be managed simultaneously, compared to implementations with large amount of ambient state. miTLS has been previously used for similar purposes within the FlexTLS tool [12];
- miTLS is designed to enable formal verification. The miTLS implementation of TLS 1.3 is under active development, but the security and correctness of many of its components have been formally proved, including the underlying cryptographic library [38] and the the record layer [5]. Currently, the miTLS implementation does not prove pseudo-uniqueness of exporter keys (which we rely on in our security proofs), but we believe that the key schedule contains all the necessary elements for such a proof, based on the PRF assumptions on HKDF and on the PRF-ODH assumption already in place for key indistinguishability, and that it is likely to be added in the future by the miTLS authors.

Our main change to miTLS is the addition of a new content type and record sub-protocol, which is used for the (t_1, t_3) and (t_2, t_3) exchanges and for transmitting K^C_{MW} and K^S_{MW}. Using a separate content type means that those messages are invisible to the miTLS handshake and do not interfere with its internal invariant and state machine (in particular, the transcript hashing and post-handshake messages). Similarly, these extra messages do not interfere with the flow of application data, following the multi-stream model of the miTLS record [5], and so do not interfere with the current application guarantees of miTLS.

Our changes are mostly contained to two files within miTLS: the top-level interface (TLS.fst) and the new middlebox sub-protocol (Middlebox.fst). Only minor changes are needed to the handshake to give access to the exporter secret-keyed PRF to the middlebox protocol. Our total changes consist of under 1000 lines of F*code, which is less than 5% of the miTLS protocol code. Our prototype implementation is available on Github as a branch of the miTLS repository.[7] We note that this prototype is restricted to a single middlebox whose certificate is known in advance to the client and server, and which has full read and write permissions.

To test our implementation, we use the simple HTTP client/server tool provided by miTLS. For the middlebox, we created a new application that exercises the new options added to the top-level interface TLS.fst. Our middlebox is configured to read and display the traffic, but otherwise performs no function.

We measured the performance of our code when using AES128-GCM record algorithm, HKDF-SHA256 for the key schedule, Curve25519 for the Diffie-Hellman key exchange, and ECDSA on the NIST P-384 curve with SHA-384 for the server and middlebox signatures. We measure a marginal increase in connection time (35ms $\rightarrow$ 38ms), but a significant throughput loss of about 40% (516MB/s $\rightarrow$ 299MB/s) in our experiment, compared to a direct connection between the client and server.

The latency increase is caused by the additional Diffie-Hellman computation at the middlebox, and disappears when PSK or 0-RTT is used. The decrease in throughput is primarily due to the re-encryption overhead, which is done sequentially in our prototype. We believe this overhead can be mitigated by parallelizing encryption and decryption on the middlebox.

V. Conclusions

In this paper, we proposed a new security definition, as well as a modular, provably-secure construction of an accountably-proxied secure channel. We hope that our design will provide a better example of how accountability can be achieved without harming the authenticity and integrity of the underlying channel. However, our formal model/definitions are complex, and –even so– we achieve limited record-layer guarantees in multi-middlebox setting. This illustrates the intuitive principle that the more middleboxes are interspersed between the client and the server, the weaker the security of incoming and outgoing messages becomes. If many (potentially-malicious) middleboxes are present, the endpoints may no longer rely on each of the proxies to inspect the traffic. As our analysis suggests, the most important middlebox should then be placed closest to the server to guarantee its functionality is observed.

Finally, our proposed protocol design is not meant to encourage widespread active proxying. Instead, we hope to have shown that, beyond the loss of end-to-end security inherent to proxying, it is difficult to construct a proxied handshake in a sound way. The integrity and authenticity properties of the record layer degrade quickly with the number and position of the corrupted middleboxes, and schemes with more than two proxies are not very efficient. As such, our results indicate that care should be taken before introducing a middlebox, and for sensitive end-to-end communications, they should preferably be eliminated.

Acknowledgments

P.-A. Fouque, K. Bhargavan, and C. Onete are grateful for the support of the ANR through grant 16 CE39 0012 (SafeTLS). K. Bhargavan was also funded by the ERC grant CIRCUS. I. Boureanu was in part funded by the Marie Skłodowska-Curie grant No 661362 under EU's Horizon 2020 programme.

[7]https://github.com/mitls/mitls-fstar/tree/middlebox

REFERENCES

[1] S. Alt, P. Fouque, G. Macario-Rat, C. Onete, and B. Richard. A cryptographic analysis of UMTS/LTE AKA. In *Proceedings of ACNS*, volume 9696 of *LNCS*, pages 18–35. Springer, 2016.

[2] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology — CRYPTO*, pages 232–249, 1993.

[3] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proceedings of S&P*, pages 483–502. IEEE, 2017.

[4] K. Bhargavan, I. Boureanu, P. Fouque, C. Onete, and B. Richard. Content delivery over TLS: a cryptographic analysis of Keyless SSL. In *Proceedings of Euro S&P*, 2017.

[5] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. Implementing and proving the tls 1.3 record layer. Appears in IEEE S&P (Oakland), 2017. Cryptology ePrint Archive, Report 2016/1178, 2016. http://eprint.iacr.org/2016/1178.

[6] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *Proceedings of IEEE S&P 2014*, pages 98–113. IEEE, 2014.

[7] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptopgrahic security. In *Proceedings of IEEE S&P 2013*, pages 445–469, 2013.

[8] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, pages 235–255, 2014.

[9] C. Brzuska, M. Fischlin, N. Smart, B. Warinschi, and S. Williams. Less is more: Relaxed yet composable security notions for key exchange. *International Journal of Information Security*, 12(4):267–297, 2013.

[10] C. Brzuska and H. kon Jacobsen. A modular security analysis of EAP and IEEE 802.11. In *Proceedings of PKC*, volume 10175 of *LNCS*, 2017.

[11] C. Brzuska, H. kon Jacobsen, and D. Stebila. Safely exporting keys from secure channels: on the security of EAP-TLS and TLS key exporters. In *EuroCrypt*, 2016.

[12] B. B. A. Delignat-Lavaud, N. Kobeissi, A. Pironti, and Karthikeyan Bhargavan. FLEXTLS: A tool for testing TLS implementations. In *Proceedings of USENIX WOOT 2015, best paper award*, 2015.

[13] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, pages 1197–1210, 2015.

[14] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. IACR ePrint archive, https://eprint.iacr.org/2016/081, 2016.

[15] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. Halderman, and V. Paxson. The security impact of HTTPS interception. In *Proceedings of NDSS*, 2017.

[16] M. Fischlin and F. Günther. Replay attacks on zero round-trip time: the case of the TLS 1.3 handshake candidates. In *Proceedings of Euro S& P*, pages 60–75. IEEE, 2017.

[17] P. Fouque, G. Macario-Rat, C. Onete, and B. Richard. Achieving better privacy for the 3gpp aka protocol. In *Proceedings of PETS (PoPETS)*. De Gruyter, 2016.

[18] R. Grahm. Extracting the SuperFish certificate. http://blog.erratasec.com/2015/02/extracting-superfish-certificate.html, 2015.

[19] F. Günther, B. Hale, T. Jager, and S. Lauer. 0-RTT key exchange with full forward secrecy. In *Advances in cryptology – EUROCRYPT*, volume 10212 of *LNCS*, pages 519–548. Springer, 2017.

[20] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *Advances in Cryptology – CRYPTO*, volume 7417 of *LNCS*, pages 273–293. Springer, 2012.

[21] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *Proceedings of ACM CCS 2015*, 2015.

[22] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. (de-)constructing TLS. IACR ePrint archive, report 020/2014, 2014.

[23] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. (De-)constructing TLS 1.3. In *Proceedings of Indocrypt*, volume 9462 of *LNCS*, pages 85–102. Springer, 2015.

[24] H. kon Jacobsen. A modular security analysis of eap and ieee 802.11. Ph.D. thesis, 2017.

[25] H. Krawczyk, K. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *Proceedings of CRYPTO 2013*, volume 8042 of *LNCS*, pages 429–448, 2013.

[26] H. Krawczyk and H. Wee. The OPTLS protocol and tls 1.3. In *Proceedings of Euro S&P*. IEEE, 2016.

[27] O. Levillain, B. Gourdin, and H. Debar. TLS record protocol: Security analysis and defense-in-depth countermeasures. In *Proceedings of ACM ASIACCS 2015*, pages 225–236. ACM, 2015.

[28] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu. Multiple handshakes security of TLS 1.3 candidates. In *Proceedings of S&P*, pages 485–505. IEEE, 2016.

[29] A. T. Ltd. Comparing approaches for web and DNS infrastructure security. goo.gl/PBD2N6, 2016.

[30] D. Migault, K. Ma, Ericsson, R. Rich, Akamai, S. Mishra, V. Communications, O. G. de Dios, and Telefonica. Lurk tls dtls use cases. https://tools.ietf.org/html/draft-mglt-lurk-tls-use-cases-02, 2016.

[31] P. Morrissey, N. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. In *Proceedings of ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 55–73, 2008.

[32] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. Steenkiste. And then there were more: Secure communication for more than two parties. In *the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 88–100, 2017.

[33] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, Diego R. López, K. Papagiannaki, Pablo Rodriguez Rodriguez, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of SIGCOMM 2015*, pages 199–212. ACM, 2015.

[34] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. TLS proxies: Friend or foe. In *Proceedings of IMC*, pages 551–557, 2016.

[35] T. Ormandy. Cloudflare reverse proxies are dumping uninitialized memory. https://bugs.chromium.org/p/project-zero/issues/detail?id=1139, 2015.

[36] K. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *Advances in Cryptology — ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer-Verlag, 2011.

[37] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43nd ACM Symposium on Principles of Programming Languages, POPL 2016*, pages 256–270, 2016.

[38] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. Appears in ACM CCS 2017. Cryptology ePrint Archive, Report 2017/536, 2017. http://eprint.iacr.org/2017/536.

Appendix A. Related Work

Two-party authenticated key-exchange protocols are an old and well-studied primitive [2] and various notions of channel security have been formalized and applied to TLS 1.2 [20], [25], [8], [22], [31], [36], [27] and 1.3 [13], [14], [19], [16], [3], [28], [23], [26], [21]. However, as Bhargavan et al. [4] showed in the context of TLS, and as Alt et al. [1] and Fouque et al. [17] – in the context of mobile networks, expanding two-party handshakes to include even a single, dedicated middlebox can expose the obtained channel to serious attacks. Our work reuses some parts of [4]'s security model, specifically the elegant way in which multiparty protocols can be viewed as compositions of 2-party AKE schemes; however, the scope of our paper is fundamentally different. The work of Bhargavan et al. centers around a particular type of caching middlebox, with all-or-nothing access rights to server contents. In this paper, our goal is to provide finer-grained access-rights (as defined through contexts) to both client- and server-side middleboxes.

Other approaches to delegated or proxied AKE protocols exist. Composition-centric approaches, such as those of Brzuska et al. [11], [10] and Jacobsen [24] capture the three-party handshake usually deployed in WLANs (also called the 4-way-handshake protocol). The main goals of that work is to prove that a handshake consisting of three separate executions of different AKE protocols, relying on different credentials, can still yield a secure channel. This is different from our case, for several reasons. The handshakes involved in the EAP protocol do not interfere with each other and can be treated independently, whereas in mcTLS and our protocol, keys are simultaneously established and distributed to multiple middleboxes. Furthermore, since our middleboxes have finer-grained access rights, record layer security in our setting is more subtle.

Our construction builds on the existing mcTLS protocol [33], which is a novel and a fundamentally different way of thinking about proxying TLS connections. In parallel to our work, a different variant of mcTLS, called middlebox TLS (mbTLS) [32], has been recently proposed. Like our protocol, mbTLS tries to keep more of the TLS handshake unchanged. However, mbTLS appears more focused on the network-architecture than on (formal) security guarantees. It also dedicates a significant part of its thrust to non-security properties (e.g., deployability, backwards compatibility). Different to our design, mbTLS looks at achieving certain security guarantees between two hops via the use of attestation, and at integrating the latter in the TLS handshake. But, unlike both mcTLS and mbTLS, we show how our design can be concretely instantiated with TLS 1.3, and we prove our proposal secure formally.

Appendix B. Instantiating our design with TLS 1.3

Fig. 5 shows how to optimally instantiate our ACCE-AP-secure design called Π, with TLS 1.3[8].

We presented our modular, ACCE-AP-secure design called Π for 1 middlebox, in an abstract manner, using generic authenticated key-exchange protocols. We will now show how this translates into a more concrete design, when the underlying authenticated key-exchange protocol used as the building block is TLS 1.3; to illustrate the computations therein, we will focus more on the server party S. I.e., one server-instance will "liaise" directly with the middlebox (in Figure 5, computations by this server-instance are written in blue). Another server-instance will "liaise" with the client (in Figure 5, computations by this server-instance are written in black). Similarly, one client-instance will "liaise" directly with the middlebox (written in red, on Figure 5), and one will "liaise" with the server (written in black, on Figure 5). For *each* instance, the normal computations for TLS1.3 take place: a handshake transport key htk is derived (in two stages: first, the TLS1.3 handshake secret hs is derived using $\mathsf{HKDF.extract}(\cdot)$ over the TLS1.3 early secret es and using as secret input *e.g.* the value $g^{yz'}$ produced by S's blue instance; second, the $\mathsf{HKDF.expand}(\cdot)$ is used over hs, a finished key fk is derived (fk is derived using $\mathsf{HKDF.expand}(\cdot)$ of the base-key), and finally a transport key tk and export keys ek are derived (as explained in the TLS 1.3 draft). As per TLS1.3, after each handshake transport key htk is derived, the TLS 1.3 handshake messages thereafter are encrypted using htk, and the finished messages are encrypted with fk. In Figure 5, we use the notation $\mathtt{AE}(\mathsf{htk};\cdot)$ to denote the authentication encryption of a message under the appropriate handshake traffic key. We can see that the ACCE part of our design, instantiated herein via TLS1.3 handshake, ends in each session –as expected– with the Finished messages. Clearly, the transport keys tk on Fig. 5 correspond to what in our generic ACCE-AP-secure construction Π was denoted as the ck keys.

Then, the *binding phase* can start, whereby different t_i (with $i \in \{1, 2, 3\}$) are produced by particular nodes of the communication and are sent across in a particular order, encrypted with the transport key. To this end, we write $\mathtt{AE}(\mathsf{tk};\cdot)$ for the authenticated encryption under a relevant traffic key tk. Like in our generic ACCE-AP-secure

[8]The newest draft of the TLS 1.3 handshake can be found at https://tools.ietf.org/html/draft-ietf-tls-tls13-21.

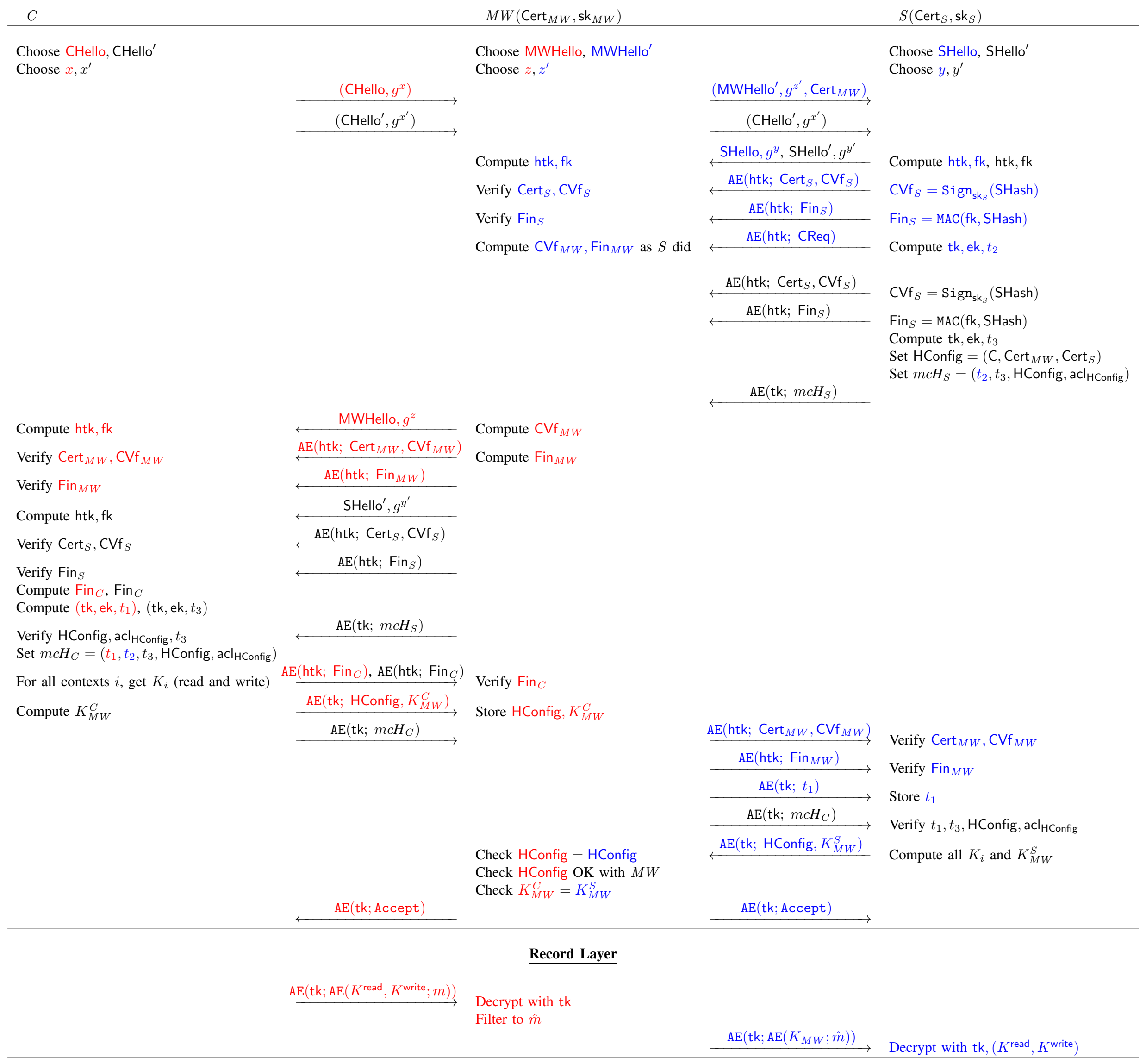

C
$MW(\mathsf{Cert}_{MW}, \mathsf{sk}_{MW})$
$S(\mathsf{Cert}_S, \mathsf{sk}_S)$
Choose CHello, CHello′
Choose x, x'
Choose MWHello, MWHello′
Choose z, z'
Choose SHello, SHello′
Choose y, y'
(CHello, g^x)
$(\mathsf{CHello}', g^{x'})$
$(\mathsf{MWHello}', g^{z'}, \mathsf{Cert}_{MW})$
$(\mathsf{CHello}', g^{x'})$
$\mathsf{SHello}, g^y, \mathsf{SHello}', g^{y'}$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Cert}_S, \mathsf{CVf}_S)$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_S)$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{CReq})$
Compute htk, fk
Verify $\mathsf{Cert}_S, \mathsf{CVf}_S$
Verify Fin_S
Compute $\mathsf{CVf}_{MW}, \mathsf{Fin}_{MW}$ as S did
Compute htk, fk, htk, fk
$\mathsf{CVf}_S = \mathtt{Sign}_{\mathsf{sk}_S}(\mathsf{SHash})$
$\mathsf{Fin}_S = \mathtt{MAC}(\mathsf{fk}, \mathsf{SHash})$
Compute $\mathsf{tk}, \mathsf{ek}, t_2$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Cert}_S, \mathsf{CVf}_S)$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_S)$
$\mathsf{CVf}_S = \mathtt{Sign}_{\mathsf{sk}_S}(\mathsf{SHash})$
$\mathsf{Fin}_S = \mathtt{MAC}(\mathsf{fk}, \mathsf{SHash})$
Compute $\mathsf{tk}, \mathsf{ek}, t_3$
Set $\mathsf{HConfig} = (\mathsf{C}, \mathsf{Cert}_{MW}, \mathsf{Cert}_S)$
Set $mcH_S = (t_2, t_3, \mathsf{HConfig}, \mathsf{acl}_{\mathsf{HConfig}})$
$\mathtt{AE}(\mathsf{tk};\ mcH_S)$
Compute htk, fk
Verify $\mathsf{Cert}_{MW}, \mathsf{CVf}_{MW}$
Verify Fin_{MW}
Compute htk, fk
Verify $\mathsf{Cert}_S, \mathsf{CVf}_S$
Verify Fin_S
Compute Fin_C, Fin_C
Compute $(\mathsf{tk}, \mathsf{ek}, t_1)$, $(\mathsf{tk}, \mathsf{ek}, t_3)$
Verify $\mathsf{HConfig}, \mathsf{acl}_{\mathsf{HConfig}}, t_3$
Set $mcH_C = (t_1, t_2, t_3, \mathsf{HConfig}, \mathsf{acl}_{\mathsf{HConfig}})$
For all contexts i, get K_i (read and write)
Compute K^C_{MW}
$\mathsf{MWHello}, g^z$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Cert}_{MW}, \mathsf{CVf}_{MW})$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_{MW})$
$\mathsf{SHello}', g^{y'}$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Cert}_S, \mathsf{CVf}_S)$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_S)$
$\mathtt{AE}(\mathsf{tk};\ mcH_S)$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_C)$, $\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_C)$
$\mathtt{AE}(\mathsf{tk};\ \mathsf{HConfig}, K^C_{MW})$
$\mathtt{AE}(\mathsf{tk};\ mcH_C)$
Compute CVf_{MW}
Compute Fin_{MW}
Verify Fin_C
Store $\mathsf{HConfig}, K^C_{MW}$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Cert}_{MW}, \mathsf{CVf}_{MW})$
$\mathtt{AE}(\mathsf{htk};\ \mathsf{Fin}_{MW})$
$\mathtt{AE}(\mathsf{tk};\ t_1)$
$\mathtt{AE}(\mathsf{tk};\ mcH_C)$
$\mathtt{AE}(\mathsf{tk};\ \mathsf{HConfig}, K^S_{MW})$
Verify $\mathsf{Cert}_{MW}, \mathsf{CVf}_{MW}$
Verify Fin_{MW}
Store t_1
Verify $t_1, t_3, \mathsf{HConfig}, \mathsf{acl}_{\mathsf{HConfig}}$
Compute all K_i and K^S_{MW}
Check $\mathsf{HConfig} = \mathsf{HConfig}$
Check HConfig OK with MW
Check $K^C_{MW} = K^S_{MW}$
$\mathtt{AE}(\mathsf{tk}; \mathtt{Accept})$
$\mathtt{AE}(\mathsf{tk}; \mathtt{Accept})$
Record Layer
$\mathtt{AE}(\mathsf{tk}; \mathtt{AE}(K^{\mathsf{read}}, K^{\mathsf{write}}; m))$
Decrypt with tk
Filter to $\hat{m}$
$\mathtt{AE}(\mathsf{tk}; \mathtt{AE}(K_{MW}; \hat{m}))$
Decrypt with tk, $(K^{\mathsf{read}}, K^{\mathsf{write}})$

Figure 5: Instantiating our ACCE-AP-secure construction Π, instantiated with TLS 1.3

construction Π, we use a PRF keyed on the relevant ek and applied over the hash of the session thus far to compute the t, binding values, *i.e.* the binder t_2 calculated by the server for the session it has with the middlebox is done as $t_2 = \mathtt{PRF}(\mathsf{ek}; \text{“}e.sid\text{”}; \mathsf{SHash})$. Similarly, the server computes a binder t_3 for the session it has with the client. All the configuration, the permissions and the binders t_1 and t_2 are placed by the server in mcH_S, which is sent (via the MW) encrypted with the transport key tk that S now shares with the client. Similarly, after the handshake phase, the client calculates binder t_1 (for the session C has with the middleware), and binder t_3 (for the session C has with the server). As it receives mcH_S, the client checks the configuration, permissions and t_3 (as sent by S).

If no fault is encountered by C in the checks of the configurations and binder t_3 as received from S (via MW), then the phase of the *computation of the access keys* can begin. C will compute the access keys for the MW and send them across to the MW, encrypted with the transport key tk that C shares with MW, *i.e.* $\mathtt{AE}(\mathsf{tk};\ \mathsf{HConfig}, K^C_{MW})$ (recall that the color red simply denotes the session $C - MW$). C will also package all the 3 binders along with the configuration and permissions into mcH_C which he sends to S, encrypted with the transport key tk that C shares with S (via MW), *i.e.* $\mathtt{AE}(\mathsf{tk};\ mcH_C)$.

Upon receiving $\mathtt{AE}(\mathsf{tk};\ \mathsf{HConfig}, K^C_{MW})$ from C, the

MW can store the config and permissions as sent by C and retrieve the binders (notably t_1) from K^C_{MW}. To this end, recall from the main manuscript that K^C_{MW} is composed of a series of keys $K_i = \mathtt{PRF}(\mathsf{ek}_3; \mathsf{ctxt}^i; mcH_C \| \mathsf{SHash})$, one for each context i. Importantly, having retrieved t_1, the middleware can send it encrypted with the transport key it shares with the server to the server, *i.e.* $\mathtt{AE}(\mathtt{tk};\ t_1)$ (recall that the color blue simply denotes the session $S - MW$).

From $\mathtt{AE}(\mathtt{tk};\ mcH_C)$ and $\mathtt{AE}(\mathtt{tk};\ t_1)$, the server can store t_1 but also check that t_1 and t_3 as seen by C are the same as produced by himself (in the case of t_3) and received by the middlebox (in the case of t_1). From here on, if no fault was found by S, then he will compute the access keys at his end and send them to the middlebox, as done previously by C, but –of course– encrypted with the key $\mathtt{tk}$ for the S–MW session, *i.e.* $\mathtt{AE}(\mathtt{tk};\ \mathsf{HConfig}, K^S_{MW})$ (recall that the colour blue simply denotes the session $S - MW$). So, at this point the middleware can confront the configs, permissions and keys as received from C and from S, to check that they are the same. From here, upon success, the configuration acceptance messages can be sent and the application layer exchanges can begin thereafter.

Appendix C. AKE & ACCE Security

The AKE and ACCE models. We now briefly review the AKE and ACCE security notions, using the notations of Brzuska et al. [11].

Parties and instances. Both models are defined in the context of a set $\mathscr{P}$ of parties, which is formed of two disjoint subset, the client set $\mathscr{C}$, and the server set $\mathscr{S}$. Parties are associated with private keys sk and associated public keys pk, which are certified by certifying authorities. Each protocol session in which a party is involved describes an *instance* of that party; π_i^m denotes the m-th instance of party P_i. Each instance is associated with a set of attributes, the session-specific ones intrinsically pertaining to the instances, whereas the long-term ones being "inherited" by instances from their "owning" parties.

For the *ACCE* model, the attributes are as follows: 1) The **private key** $\pi_i^m.\mathsf{sk} := \mathsf{sk}_i$ and **public key** $\pi_i^m.\mathsf{pk} := \mathsf{pk}_i$ of an instance π_i^m of party P_i; 2) The **role** $\pi_i^m.\rho \in \{init, resp\}$ of P_i, *i.e.*, that of an *initiator* (the party that initiates the execution of the protocol) or of a *responder* (the party responding in that execution); 3) The **session identifier** $\pi_i^m.\mathsf{sid}$ of an instance; 4) The **partner identifier**, $\pi_i^m.\mathsf{pid}$ is either a party identifier P_j or, in unilateral authentication, it can be a label "Client"denoting the client with whom the party P_i believes to be communicating. 5) The **acceptance-flag** $\pi_i^m.\alpha$ is 1 or 0 denoting that party respectively accepts or rejects the partner's authentication; to begin with and prior to the end of the handshake, this attribute is set to $\perp$; 6) The **revealed bit** $\pi_i^m.\gamma$ is 1 or 0 denoting that party has had its session keys revealed or not; to begin with, 0 is assigned to this attribute; 7) The **channel-key**, $\pi_i^m.\mathsf{ck}$ is set to a non-null bitstring when π_i^m ends in an accepting state; prior to the successful end of the session, this attribute is set to $\perp$; 8) The **left-or-right** bit $\pi_i^m.\mathsf{b}$, sampled at random when the instance is generated and used in the key-indistinguishability and channel-security games; 9) The **transcript** $\pi_i^m.\tau$ of the instance, containing the suite of messages received and sent by this instance, as well as all public information.

A crux notion of the ACCE models is that of *partnering*: two instances π_i^m and π_j^n are *partnered* if their identifiers $\pi_i^m.\mathsf{sid}$ and $\pi_j^n.\mathsf{sid}$ are equal and non-$\perp$.

ACCE/AKE games and adversarial queries. In the ACCE model, a MiM adversary can interact with parties in concurrent or sequential session. His interaction is formally captured via a series of oracles. I.e., the $\mathsf{NewSession}(P_i, \rho, pid)$ oracle-query produces a new instance of P_i. Sending a message mes to such an instance π_i^m is modelled via the MiM issuing a $\mathsf{Send}(\pi_i^m, mes)$ query. The $\mathsf{Corrupt}(P_i)$ queries encapsulate the MiM learning party P_i's secret key. Querying $\mathsf{Reveal}(\pi_i^m)$ denotes the MiM finding out the channel keys for an accepting instance π_i^m.

In the AKE models, the MiM has access to all the above ACCE queries, as well as to the $\mathsf{Test}(\pi_i^m)$ oracle. On such a query, the output is either the real channel keys $\pi_i^m.\mathsf{ck}$ computed by the accepting instance π_i^m or some random keys of the same length.

To define ACCE/AKE security on top of a MiM adversary $\mathscr{A}$ having access to all the oracles above, we also need to employ the notion of session freshness.

Session freshness. A session π_i^m is *fresh* with intended partner P_j, if the following holds: 1) upon the last query of the adversary $\mathscr{A}$, the uncorrupted instance π_i^m has finished its session in an accepting state, with $\pi_i^m.\mathsf{pid} = P_j$; 2) P_j is uncorrupted; 3) no Reveal query was made for the π_i^m, π_j^n.

AKE Entity Authentication (EA). In the entity authentication game, the adversary queries the $\mathsf{NewSession}(P_i, \rho, pid)$, $\mathsf{Send}(\pi_i^m, mes)$, $\mathsf{Reveal}(\pi_i^m)$, and $\mathsf{Corrupt}(P_i)$ oracles and its goal is to make one instance, π_i^m, end in an accepting state, with partner ID P_j, which must be uncorrupted, such that no other unique instance of P_j partnering π_i^m exists. The adversary's advantage in this game is its winning probability.

AKE Key-indistinguishability (KI). In the KI game, $\mathscr{A}$ makes queries to the $\mathsf{NewSession}(P_i, \rho, pid)$, $\mathsf{Send}(\pi_i^m, mes)$, $\mathsf{Reveal}(\pi_i^m)$, and $\mathsf{Corrupt}(P_i)$) oracles and makes a single Test query for a *fresh* instance π_i^m. It wins if it can guess this party's correctly randomly-chosen bit $\pi_i^m.\mathsf{b}$. The adversary's advantage is the absolute difference between its winning probability and $\frac{1}{2}$. The AKE security of a protocol is given by the sum of the advantages an adversary has to break either of the two properties, EA or KI.

ACCE security. In the ACCE model, the adversary is slightly different to the AKE adversary: 1) it does not have access to the Test oracle; 2) it has access to the oracles $\mathsf{Encrypt}(\pi_i^m, l, M_0, M_1, H)$ and $\mathsf{Decrypt}(\pi_i^m, C, H)$, which simulate the access to the secure channel established by party instances. Intuitively, if the Decrypt oracle is queried on an adversarially created ciphertext (without using the Encrypt oracle), it will implicitly reveal the value of the bit b.

ACCE security is defined in terms of two games: the EA game presented as in the ACCE model, and the following game expressing the requirement of security for the established channel.

ACCE Security of the Channel (SC). In this game, the adversary $\mathscr{A}$ can use the $\mathsf{NewSession}(P_i, \rho, pid)$, $\mathsf{Send}(\pi_i^m, mes)$, $\mathsf{Reveal}(\pi_i^m)$, and $\mathsf{Corrupt}(P_i)$ oracles. For a fresh instance π_i^m, $\mathscr{A}$ must output, the bit $\pi_i^m.\mathsf{b}$ of that instance. The adversary's advantage is the absolute difference between its winning probability and $\frac{1}{2}$.

ACCE-security for TLS. It is known that TLS 1.2 in DHE mode is (S)ACCE secure with unilateral authentication and ACCE secure if the mutual authentication mode is used [20], [25]. A recent compositional result by Brzuska et al. [11] describes the construction of AKE-secure export keys from an ACCE-secure key-exchange protocol such as TLS 1.2. We note, however, that these results only prove the security of fresh sessions, in which both partners are honest and legitimate. In particular, the fact that the sessions keys are computed only on the nonces (and not, *e.g.*, over the entire session transcript incl. the exchanged certificates) implies a lack of uniqueness. In particular, a malicious, but legitimate server can force two different sessions (one with a client and another with an honest server) to share keys [6]. In addition, moreover, the client may also be fooled, by a malicious server, into thinking it is negotiating a handshake with a different, honest server. In our case, such attacks are particularly relevant, since we wish to compose handshakes, in a scenario in which some of the participants *may* be malicious. This weakness extends to the export-key construction of Brzuska et al. [11].

For TLS 1.3 (up to and including draft 10), Dowling et al. [14] have proved the (multi-stage) AKE security of the keys output in TLS 1.3. Although the protocol has changed since, it can still be inferred that the full mode of TLS 1.3 provides the same level of security for the obtained keys. Note that by construction, TLS 1.3 offers the possibility of computing export keys, and the proof of Dowling et al. also postulates their indistinguishability from random. As opposed to TLS 1.2, the TLS 1.3 handshake does provide the uniqueness of session keys, even in the presence of malicious, legitimate servers. This is because the TLS 1.3 key material is always computed on the entire session hash, including the authenticating information provided by the client and server.

On a different note, whereas in TLS 1.2 the server can simply choose DHE parameters as it wishes, in TLS 1.3 it is the client that chooses the DHE groups from amongst a list of possible groups that are considered secure. Although this does not impact our result, it does mean than our multicontext protocol is secure when instantiated with *any* of those groups. Should we wish to instantiate it with TLS 1.2, the security of the multicontext handshake would depend on which DHE parameters the server chooses.