Zero-Knowledge Middleboxes

Paul Grubbs Arasu Arun Ye Zhang Joseph Bonneau Michael Walfish

NYU Department of Computer Science, Courant Institute

Abstract. This paper initiates research on *zero-knowledge middleboxes* (ZKMBs). A ZKMB is a network middlebox that enforces network usage policies on encrypted traffic. Clients send the middlebox zero-knowledge proofs that their traffic is policy-compliant; these proofs reveal nothing about the client's communication except that it complies with the policy. We show how to make ZKMBs work with unmodified encrypted-communication protocols (specifically TLS 1.3), making ZKMBs invisible to servers. As a contribution of independent interest, we design zero-knowledge proofs for TLS 1.3 session keys. We apply the ZKMB paradigm to several case studies, including filtering for encrypted DNS protocols. Experimental results suggest that performance, while not yet practical, is promising. The middlebox's overhead is only 2–5ms of running time per verified proof. Clients must store hundreds of MBs to participate in the protocol, and added latency ranges from tens of seconds (to set up a connection) to several seconds (for each successive packet requiring proof). Our optimized TLS 1.3 proofs improve the client's costs 6× over an unoptimized baseline.

1 Introduction and Motivation

A decades-old conflict in Internet architecture pits user privacy against network administration. Network administration is typically carried out by middleboxes inspecting traffic. Strong privacy, however, requires concealing information—both what and with whom users are communicating—from *all* intermediate parties, including the middleboxes.

Recent developments in encrypted DNS form an illustrative example. DNS queries from end hosts (e.g., "What is the IP address for `example.com`?") have traditionally been sent to resolvers operated by the local network administrator. This architecture arose for performance reasons; as a byproduct, the administrator can use the local resolver as a middlebox, monitoring DNS queries and enforcing network policies like content filtering. Network providers sometimes abuse this capability, for example, by redirecting users to ads via DNS responses [16, 20, 73]. Selling data derived from DNS queries to advertisers has also been legal in the US since 2017 [73]. Another alarming practice is targeted government surveillance [16, 60] using DNS.

In response, several protocols have emerged for encrypting DNS queries to remote resolvers outside the control of the local network administrator [68, 71, 76]. While the primary goal is privacy, encrypted DNS also bypasses any local network policies. For this reason, many providers, including commercial ISPs, have lobbied hard against encrypted DNS [29]. As an attempted compromise, Mozilla introduced a special canary domain that instructs clients to fall back to legacy DNS. But this is no compromise: it is simply handing the local network a (silent) veto over encrypted DNS.

Encrypted DNS represents a fraught *tussle* [30, 70] that we specifically want to confront and resolve. Thus, this paper's overarching question: *can we create mechanisms that achieve meaningful compromise?* Note that we do not take a philosophical position for or against middleboxes.[1] That debate is almost as old as the Internet architecture itself. Instead, we accept that policy-enforcing middleboxes are a fact of life today and likely to remain so. For example, adult content blocking is legally mandated for U.S. K–12 school networks [27, 61] and commercial U.K. ISPs [122]; these networks comply with their legal mandate by filtering DNS.

The privacy-vs-policy conflict in middleboxes has motivated prior research efforts (§9). These efforts either (like SGX-Box [64] and EndBox [56]) rely on trusted execution environments [36, 41, 57, 63, 82, 106, 123, 129] or (like BlindBox [115] and mcTLS [98]) require application-specific modifications to standard cryptographic protocols [45, 75, 83–86, 90, 100, 101, 107, 132, 134].

These works conflict with a requirement that we regard as essential: **compatibility**. This means, first, no changes to standard protocols and deployed servers. Protocols such as TLS and HTTPS have been carefully developed over several generations and are now widely deployed; it is thus impractical to presume ubiquitous change [33]. We do tolerate changes to middleboxes and the clients behind them, as these can happen incrementally. Compatibility also means the network is open to all, not just clients that have particular trusted hardware.

This paper initiates research on *zero-knowledge middleboxes (ZKMB)*. The approach is inspired and enabled by dramatic progress over the last decade in probabilistic proofs, including zero knowledge (ZK) proofs (§3, §9). Once purely a theoretical construct, they have received sustained interdisciplinary focus, including real-world deployment in blockchain applications. A ZK proof protocol allows a *prover* to convince a *verifier* of the truth of a statement (for example, that a given boolean formula is satisfiable) while keeping secret the basis of its argument (for example, the satisfying assignment itself). If the statement is false (for example, no satisfying input exists), then the verifier is highly unlikely to be convinced.

Under ZKMBs, clients and servers communicate with standard encryption protocols (HTTPS, for example); the scheme

[1] However, we want to be clear that this work is not about enabling censorship or surveillance. On the contrary, as discussed in Section 2, this work will make it harder to justify those activities.

is invisible to servers. The mechanics happen between clients and their local networks, preserving today's network architecture. Given a policy mandated by the middlebox, honest clients send only traffic that complies (by default, clients know the middlebox's policies, though ZKMBs accommodate proprietary policies; §7.4.1). The middlebox sees this traffic in encrypted form, and clients are required to include a ZK proof that the plaintext traffic indeed complies. The middlebox acts as a ZK verifier, allowing only traffic with valid proofs.

Challenges. There are three mutually exacerbating challenges in bringing ZKMBs to life. The first is preventing the client from circumventing policy by encrypting and sending one (non-compliant) plaintext while generating a proof about another (compliant) one. Equivocation of this kind is possible if the encryption protocol does not offer a *binding* property (meaning that, given a ciphertext, there is only one correct decryption). But encryption protocols don't need a binding property to be secure, and it has a performance cost, so modern encryption protocols like TLS 1.3 do not provide it [37,62,89]. Consequently, ZKMBs need to somehow extract novel security guarantees from existing protocols.

The second challenge is the formalism within which one writes a statement to be proved in zero knowledge: the circuit model of computation (§3). Computations generally have verbose representations in that formalism, creating unacceptably large overhead. Indeed, despite all of the recent progress (and hype) in ZK proofs, they are not usable as a turnkey solution. "Practical" proof projects all have application-specific design, not only in deciding how to express a given computation as a circuit but also in choosing a circuit-friendly computation in the first place (for example, hash functions with particular algebraic structure).

The last challenge stems from our compatibility requirement. Because we must work with legacy network protocols, we often have no choice about the computation. We have to implement circuit-unfriendly functionality in circuits.

Contributions and results. We introduce a modular pipeline for expressing a ZKMB circuit (§4), and apply it to three case studies (§6–§7): firewalling non-HTTPS traffic, blocklisting domains for encrypted DNS (addressing the earlier example), and allowlisting resolvers for oblivious DNS.

As a contribution of independent interest, we show how to prevent the client from equivocating in TLS 1.3 (§5). We start with a simple but new observation: the messages exchanged during the key agreement protocol, together with a given ciphertext, binds the client to a single plaintext. One way to exploit this observation is to re-run major parts of the key agreement protocol in a circuit (§5.1). That would entail elliptic-curve cryptography (with circuit-unfriendly parameters) as well as hashing the entire key agreement transcript with SHA-256 (which is likewise circuit-unfriendly). However, we exhibit a shortcut that removes these expensive operations from the circuit (§5.2). The enabling insight is that the server's authentication messages include a hash of the results of these operations; the circuit need not re-run them as long as it verifies this hash. We give a provable-security analysis of the baseline and the shortcut (Appx. C). An interesting feature of our approach is that, because the key is usually fixed for an entire TLS 1.3 session, the work of proof generation can be amortized (§5.3) across all encrypted traffic sent in a session.

Another aspect of our work is parsing legacy protocols in circuits. The naive approach, expressing a full parser in circuit form, would be too expensive. Instead, we carefully tailor the parsing to the policy, to identify just the needed tokens. We also have to adapt the parsing algorithms to the costs imposed by circuits. For example, we identify the first CRLF in an HTTP request using a loop that is carefully written to avoid certain branches (§6).

Our implementation of ZKMBs (§8) uses xJsnark [79] for compiling code to circuits and the Groth16 [59] proof protocol. As a consequence of Groth16, our implementation outputs 128-byte proofs which take 2–5ms to verify, regardless of the encoded policy. The salient performance issue is prover (client) time. A proof that accompanies the initiation of a TLS connection costs roughly 14 seconds to generate on a good CPU; proofs in successive packets cost 4–8 seconds to generate, depending on the policy. Also, the client must store hundreds of MB of data to participate in the proof protocol. These costs mean that ZKMBs aren't ready to be deployed today. However, our optimizations already yield meaningful improvements: for example, our shortcut TLS 1.3 circuit is 6× smaller than the baseline.

Future work (§10) includes handling fragmentation of traffic across multiple TLS records, and supporting additional middlebox configurations and functionality (scanning-style middleboxes, off-path middleboxes, and middleboxes with proprietary policies beyond what is outlined in §7.4.1). Other future work is interrogating tradeoffs (§9) among setup costs, trust, verifier performance, and prover performance, to identify the optimal ZK machinery for this context.

Despite the limitations of our work, we believe that the ZKMB approach is worthy of continued research: it expands the frontiers of network architectures, pushes privacy-enhancing technologies into new settings, and surfaces new applications of efficient zero-knowledge proofs (whose real-world impact so far has been substantial but on the other hand, limited to blockchains). In summary, our contributions are:

- A new application of ZK proofs;
- The ZKMB paradigm (§2–§4) and a modular framework for circuits in this setting;
- Sub-circuits for efficiently decrypting TLS 1.3 (§5);
- Three case studies (§6,§7) showing how ZKMBs can resolve tensions between network administration and user privacy, including for encrypted DNS; and
- Implementation and evaluation (§8). Source code is at `https://github.com/pag-crypto/zkmbs`

2 Overview, Model, and (Non)-Objectives

Our setting includes three principals: a *client*, a *server*, and a network. The client is attached to the network, which deploys an *on-path middlebox*, meaning that the client's traffic physically traverses the middlebox en route to any server. (We discuss other middlebox architectures later; §10.) Figure 1 depicts the principals, abstracting the entire network as the middlebox. The goal of the client is to communicate with the server via the middlebox, using an end-to-end encrypted channel to keep its data secret. The goal of the middlebox is to enforce some policy; as is typically the case with middleboxes, policy enforcement means dropping traffic that does not adhere to the policy. This policy can affect which servers and protocols the client can use, or it can affect communication content (for example, signatures of malware).

For clients, the ZKMB protocol should be incrementally deployable, and represent an *opportunistic privacy upgrade*. That is, clients without support for ZKMBs can continue to use the network—however, as happens today, the middlebox will block their encrypted traffic, forcing a downgrade to unencrypted protocols.

In the basic protocol flow in ZKMBs, the client downloads a network policy specification from the middlebox upon joining the network. (Looking ahead, in Section 7.4.1 we will show how policies can be kept secret from clients in some cases, with moderate overhead, and in Section 10 we discuss policy updates.) This specification defines a policy-relevant *scope*; for example, all traffic of a certain class. Then for each encrypted channel to a remote server that is within scope, the client submits *proof* that the contents of the channel comply with the network policy; the middlebox verifies it. Note that the middlebox has to be able to reliably identify in-scope traffic; we discuss how this can be done for our case studies (§6,§7). The server is unaware that a ZKMB protocol is even in use. We elaborate on the steps of a ZKMB protocol in Section 4.

Threat model, objectives, and non-objectives. We assume that parties cannot break cryptographic primitives. In particular, we assume that all parties are probabilistic polynomial-time adversaries, as is standard in cryptography.

ZKMBs should ensure that an adversarial middlebox learns nothing about the plaintext, with two exceptions. First, the middlebox learns whether the plaintext is policy-compliant. Second, if an existing unmodified protocol leaks information through timing or metadata—as is the case with encrypted DNS [23, 117, 118]—ZKMBs necessarily inherit that leak. We will also not defend against middlebox-server collusion.

ZKMBs should ensure that a client, abiding by standard protocols, complies with policy within the relevant network layer. For example, a client's traffic to its stated destination should be subject to the middlebox's policy. Of course, the client's true destination may be other than what is stated in packets, using VPNs, Tor [35], hidden proxies [133], "inner encryption," or steganography [131]. Detecting and blocking such circumvention is an arms race (see, for example, the progression of obfuscated Tor transport-layer encodings [43]).

We can't hope to build a protocol that defeats all circumvention, but more importantly we don't want to. We view ZKMBs as establishing a default that respects both policy enforcement and privacy, while retaining the ability of advanced users to circumvent the policy, at a comparable level of difficulty to circumventing traditional network filtering of legacy (unencrypted) traffic. Seeking to block all circumvention of network policy would cross a line from filtering into censorship. Note that this does *not* imply that ZKMBs are equivalent to an "honor system" model with no proofs, in which clients download the middlebox's policy and simply exercise discretion about whether to abide by the policy. Such a design would explicitly disempower the provider from enforcing policy.

Why would providers adopt the ZKMB model? Our belief is that most network administrators *do not want to surveil*. Using ZKMBs, they can establish publicly that they are not surveilling (but are meeting their obligations to apply policy). Another motivation for our work is that, by demonstrating the plausibility of ZKMBs, we show that "policy enforcement requires plaintext" does not follow, removing an alleged rationale for surveillance.

Bad actors, such as repressive governments, likely will not deploy ZKMBs because they specifically want to surveil (and censor). With ZKMBs, however, non-surveilling networks can prove themselves as such.

Costs and scenarios. We evaluate the cost of deploying a ZKMB protocol on several dimensions: client computation costs for proof generation, middlebox verification costs, communication costs to send proofs from client to middlebox, and setup costs for clients to download parameters (defined below) when joining the network. Overall connection latency overhead is the sum of computation time for clients and middleboxes as well as time to transfer proofs. We consider connection latency to be the primary barrier to practical deployment; hence minimizing all three components is essential.

An important design consideration is who generates the public parameters (the SRS; §3) necessary for the ZK proof system. Some ZK proof schemes require no trusted setup [8, 22, 112, 113] but have larger proofs that take longer to verify. Given our desire to minimize connection latency (subject to maintaining compatibility with existing protocols), we rely on a proof system with a trusted setup procedure [59]. This trusted setup could be run by each middlebox, with clients downloading the public parameters upon joining the network. However, this may impose large overhead on clients when joining a new network. Alternately, this trusted setup could be run once (for example, by browser or OS vendors) for specific policies and pre-loaded onto clients. This adds a trust requirement: if setup is compromised, it will undermine

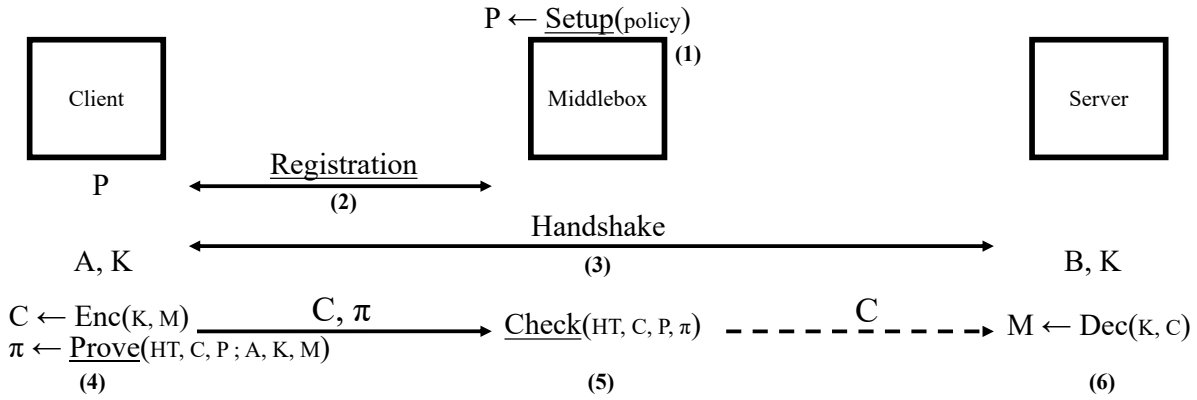

Figure 1: Architecture of a zero-knowledge middlebox. All notation is defined in Section 4.

soundness of the proof system and enable policy circumvention via forged proofs. However, subverting trusted setup (regardless of who executes it) does not affect privacy [48].

3 Background on Zero-Knowledge Proofs

This section gives necessary context on zero-knowledge (ZK) proofs. Our treatment is simplified and tailored to a particular strand of implemented ZK proof pipelines. We discuss specific systems further in Section 9.

A ZK proof protocol surrounds a given *computation* (or statement) $\mathcal{C}$, a *verifier* $\mathcal{V}$, and a *prover* $\mathcal{P}$. We formulate the computation as a function $\mathcal{C}(X;W)$ that produces output Y. Each of X, Y, W are vectors of variables. X and Y are known as public input and output variables, respectively; W is known as the secret input, or *witness*. The semicolon between X and W delineates the public and the secret input.

In a ZK proof protocol run, $\mathcal{V}$ starts with a pair (x, y). $\mathcal{P}$ produces a short certificate, or *proof*, that convinces $\mathcal{V}$ that (x, y) are valid according to $\mathcal{C}$. "Valid" means that there exists a w such that $y = \mathcal{C}(x; w)$, and furthermore that $\mathcal{P}$ knows such a w. As an example in our context, imagine that y is yes-or-no, x is encrypted data, w is the corresponding plaintext plus auxiliary information needed to decrypt, and $\mathcal{C}$ embeds decryption logic and policy logic. Semantically, the protocol convinces $\mathcal{V}$ that the (public) putative yes-or-no is consistent with the (secret) plaintext that corresponds to the (public) encrypted data.

For many modern ZK constructions, $\mathcal{C}$ is expressed as a generalization of *arithmetic circuits*, called R1CS [52]. For convenience, we call this formalism "circuits". Circuits bring challenges [18, 32, 102, 104, 114, 126]. There is no notion of state (unlike in hardware circuits) nor a program counter. To encode a loop, one unrolls it—to the worst-case number of iterations. Similarly, conditional statements pay for all branches. Bitwise operations and order comparisons are verbose. RAM is costly [10, 19, 79, 103, 126].

Our work uses the QAP-based [52] proof protocol Groth16 [59], which is a non-interactive zero knowledge proof, or NIZK [14, 46] (Groth16 is a kind of NIZK called a zkSNARK [13, 52].) For our purposes, a NIZK is a tuple of (possibly probabilistic) algorithms (ZKSetup, ZKProve, ZKVerify) that depend on $\mathcal{C}$:

- $\sigma \leftarrow \mathsf{ZKSetup}(\mathcal{C})$. This is run by $\mathcal{V}$ or a party that $\mathcal{V}$ trusts. σ is called a structured reference string, or *SRS*, and is a necessary aid to both $\mathcal{P}$ and $\mathcal{V}$. The SRS is generated once for $\mathcal{C}$ and reused over different (x, y, w).
- $\pi \leftarrow \mathsf{ZKProve}(\mathcal{C}, \sigma, (x, y), w)$. π is referred to as the proof.
- $\texttt{accept/reject} \leftarrow \mathsf{ZKVerify}(\mathcal{C}, \sigma, (x, y), \pi)$.

If $\mathcal{P}$ has a legitimate witness w for a particular (x, y), $\mathcal{P}$ can produce π that makes ZKVerify accept. If the statement is not valid (that is, there is no w for which $y = \mathcal{C}(x; w)$) or $\mathcal{P}$ does not know a w for which $y = \mathcal{C}(x; w)$, then the probability (over random choices by ZKSetup and/or ZKVerify) that $\mathcal{V}$ accepts is negligible.

In Groth16, the proof length, $|\pi|$, is short, and ZKVerify runs in nearly constant time with good concrete costs. However, ZKProve and ZKSetup are relatively expensive (§8).

4 ZKMB Protocol

In this section we define a ZKMB protocol, as represented in Figure 1. A ZKMB augments a base *secure channel* protocol, SChan. In our case studies, this is TLS, but we define it abstractly as $\mathsf{SChan} = \langle \mathsf{Handshake}, \mathsf{Enc}, \mathsf{Dec} \rangle$. Handshake is an interactive protocol between the client and server which establishes a shared symmetric key K. Given K, Enc and Dec are used by the client and server to encrypt and decrypt messages, respectively. A ZKMB protocol will not modify SChan.

A ZKMB protocol is a tuple of algorithms: $\mathsf{ZKMB}[\mathsf{SChan}] = \langle \mathsf{Setup}, \mathsf{Registration}, \mathsf{Prove}, \mathsf{Check} \rangle$

each of which are interactive between the client and middlebox. Our presentation assumes that the ZKMB protocol uses an underlying ZK proof protocol, however this interface could potentially work with other cryptographic primitives (e.g. secure multi-party computation).

The Setup procedure (item (1) in Figure 1) is run once at the beginning of some era; it is not re-run for each client. Setup takes as input a description of the network security policy and outputs policy metadata P. The metadata in P may include the SRS (§3) for the underlying ZK proof scheme (if not preloaded by the client) and possibly some additional public inputs to the proof such as a list of blocked domains.

The Registration procedure (item (2) in Figure 1) is an interactive algorithm run by a client and the middlebox to give the client the policy metadata P upon joining the network. We expect this to piggy-back on network-layer bootstrap protocols, e.g. using DHCP [39] extensions [40].

The client runs SChan.Handshake to open a secure connection to a server (item (3) in Figure 1). When the client wants to send a message M to the server, the client first checks that M is actually policy compliant; if it isn't then no proof can be generated so the client will decline to send M to the server and report an error to the user. Otherwise, the client first runs $\mathsf{SChan.Enc}(K, M)$ to obtain a ciphertext C, then runs the interactive, randomized Prove algorithm (both in item (4) in Figure 1) to prove that M is allowed by the policy. Prove takes as public input the handshake transcript HT, policy metadata P, and ciphertext C, and as private input the randomness A generated by the client during the handshake, the channel secret K, and message M. (We have abstracted away the details of the value A in this diagram; for concreteness, one can imagine A being a Diffie-Hellman secret as in TLS 1.3.)

Prove produces a proof π (using the underlying ZKProve algorithm) which the client sends to the middlebox along with the ciphertext C. The circuit that the ZK proof works over (§3) comprises three conceptual parts:

1. *channel opening*: Verify that the (hidden) message M is what results from decrypting the ciphertext C under the (hidden) key K, and that K is consistent with HTand A.

2. *parse-and-extract*: this is a translation layer between the network protocol wire format of M and the input format of the policy check component of the circuit. In our case studies, parse-and-extract will generally extract the policy-relevant substring of a network packet (call that string D), while checking that some syntactic requirements are met.

3. *policy check*: check that D satisfies the policy component of the circuit included within P.

Terminology note: We will often, in the sections ahead, refer to these parts as individual circuits. However, they are truly *sub*-circuits. From the perspective of the ZK proof protocol (§3), the combination of the three parts is a single circuit C, with a single associated SRS. Section 10 discusses extensions to this model.

After receiving π from the client, the middlebox runs the Check procedure (item (5) in Figure 1). This procedure takes as input the outputs of Prove, C and π, along with the transcript of SChan.Handshake, and invokes the underlying ZKVerify algorithm to output T/F. The result determines whether the middlebox forwards the client's encrypted traffic C to the server, which consumes the message using its unmodified SChan.Dec (step (6) in Figure 1)

Some ZKMB applications may require a single proof per channel opening. Others may require ongoing proofs as multiple messages are sent, in which case the client and middlebox can repeat steps (4)–(6) for the duration of the session. Some proving costs can be amortized across multiple messages sent in the same channel, as discussed in §5.3. Note that our formalism only considers proofs about client-to-server traffic. Proofs about server-to-client traffic could easily be supported, though they are not necessary in our motivating applications.

5 Channel Opening for TLS 1.3

This section describes a channel opening sub-circuit for the case where SChan is version 1.3 of the Transport Layer Security protocol (TLS) [109]. This sub-circuit will be used by all of our case studies (§6–§7). We focus on TLS 1.3, the newest version, because of its increasingly widespread deployment.

The challenge is that neither of the symmetric encryption schemes supported by TLS 1.3—AES-GCM and ChaCha20/Poly1305—are *committing* [62]. Using known equivocation attacks [37, 62, 89], the client could create a ciphertext with two decryptions: one that obeys the policy, and one that violates it. Then the client could run Prove to generate a proof about the first plaintext, while the server instead receives the second one.[2]

We note that this issue does not exist for TLS 1.2 because it supports committing schemes (e.g., HMAC-then-CBC). Thus, for TLS 1.2, it would suffice for the channel-opening sub-circuit to decrypt the ciphertext and check that this decryption equals the plaintext supplied as secret input. Indeed, prior work in other contexts—data feeds in blockchains [135] and anonymous PKI [130]—built proofs about TLS 1.2 plaintexts this way. (See Section 9 for a more detailed comparison between these works and ours.)

To prevent equivocation for TLS 1.3, we begin by observing that it suffices to ensure that (a) the transcript of the TLS 1.3 handshake is committing to the session key (i.e., the key that the client uses to encrypt its messages to the server), and (b) the channel opening sub-circuit verifies that this session key is consistent with the handshake. In Appendix C,

[2]One might wonder whether the channel opening sub-circuit could simply check that the plaintext has the expected packet format for the application-layer protocol, such as DNS. However, even structured formats admit equivocation attacks, as Dodis et al. [37] demonstrated for images.

Theorem 1, we prove that handshake transcripts are indeed commitments to the session keys. (The proof is simple, but we are not aware of any prior proof of this fact.) The rest of this section gives an overview of TLS 1.3, a baseline channel opening sub-circuit, and a crucial optimization.

TLS 1.3 Overview. TLS 1.3 sessions involve a client, who initiates the connection, and a server. TLS 1.3 has two sub-protocols: a *handshake* protocol and a *record* protocol. The handshake protocol begins with the client constructing a "Client Hello" message CH. This message usually contains a random value N_c and an ephemeral Diffie-Hellman (DH) key share g^a. The client starts the handshake by sending CH to the server. The server responds with two message: first, a "Server Hello" message SH of its own, containing a random N_s and key share g^b; second, an encryption C_{EE} (roughly, under g^{ab}) of the server's authentication data: its certificate and both a signature σ and MAC SF (the "ServerFinished" value) of the transcript. Once the client decrypts and verifies σ and SF, it responds with the encryption of a MAC CF of its own — the "ClientFinished" value — which completes the handshake protocol. After the handshake protocol finishes, the client and server can use the record protocol to send data back and forth. The data is split into *records* and encrypted using one of two AEAD schemes: AES-GCM or ChaCha20/Poly1305.[3] (Readers will find some additional preliminaries on algorithms used in TLS 1.3 in Appendix A.)

5.1 Baseline Channel Opening Circuit

The baseline channel opening sub-circuit prevents the client from equivocating by using the client's DH secret a to re-derive the session key $CATK$ and compare it against the claimed value K. The resulting sub-circuit is depicted in pseudocode in Figure 2. Computations on solely public data are done outside the sub-circuit by the verifier. We omit some details of TLS 1.3's key schedule for simplicity.

The channel opening sub-circuit takes as public input the handshake transcript CH, SH, C_{EE}, and the ciphertext C. The client's private inputs are the claimed values of a, the key K, and the plaintext M. First, the sub-circuit decrypts the ciphertext C with K and checks that the result is M.[4] Second, it checks that the client's provided key K is output by the key schedule (the CATKeyVerif procedure) by re-running part of the key schedule and also checking the witness a against the client's public DH value.

Efficiency of the baseline. This sub-circuit is expensive: for two realistic parameter choices, the corresponding arithmetic sub-circuit has over 7.5 million and over 10 million multiplication gates. (More context on these numbers will be given in Section 8.) Some of these costs are inherent and stem from TLS 1.3's use of legacy cryptographic primitives like AES and SHA-256. For example, each hkdf.ext or hkdf.exp operation requires about 100,000 multiplication gates. However, the two most expensive parts of the sub-circuit are (1) the two elliptic-curve scalar multiplications needed to derive DHE and check A, and (2) computing h_3, which requires both decrypting C_{EE} to get EEP and hashing the transcript. The two elliptic-curve operations together cost 2.4 million gates, while computing h_3 costs 4 million gates for a 3000-byte transcript. Deriving h_3 in the sub-circuit also requires the sub-circuit size to depend on the transcript length; thus, the sub-circuit must be padded to handle the largest possible transcript.

BaselineChanOpen$(CH, SH, C_{EE}, C; a, K, M)$:
$M' \leftarrow \mathsf{Dec}(K, C)$
$b \leftarrow$ CATKeyVerif$(CH, SH, C_{EE}; a, K)$
Return $b \wedge (M' = M)$

CATKeyVerif(CH, SH, C_{EE}, a, K):
$A \leftarrow \mathsf{GetDH}(CH)$
$B \leftarrow \mathsf{GetDH}(SH)$
$\mathsf{DHE} \leftarrow B^a$ // EC scalar multiplication
$HS \leftarrow \mathsf{hkdf.ext}(\mathsf{DHE})$
$SHTS \leftarrow \mathsf{hkdf.exp}(HS, H(CH||SH))$
$SHTK \leftarrow \mathsf{hkdf.exp}(SHTS, \text{"shtkey"})$
$EEP \leftarrow \mathsf{Dec}(SHTK, C_{EE})$// decrypt to get h_3 input suffix
$h_3 \leftarrow H(CH||SH||EEP)$// input for $CATK$ derivation
$dHS \leftarrow \mathsf{hkdf.exp}(HS, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf.ext}(dHS)$
$CATS \leftarrow \mathsf{hkdf.exp}(MS, h_3)$
$CATK \leftarrow \mathsf{hkdf.exp}(CATS, \text{"catkey"})$
$b \leftarrow g^a = A$ // EC scalar multiplication
Return $b \wedge CATK = K$

Figure 2: Baseline channel opening circuit. Certain details, such as label inputs to hkdf.exp and TLS record headers, are omitted.

5.2 Optimizing the Baseline

The core idea of the optimization is to re-purpose the SF value sent in the handshake as a commitment to two intermediate values of the key schedule: HS and the transcript hash h_7. We will show that this lets us eliminate elliptic-curve cryptography and the derivation of h_3 from the sub-circuit, while maintaining the binding security needed to prevent the client from equivocating about K.

Figure 3 shows pseudocode for this method. For simplicity, the figure presents a simplified variant of the optimization; we discuss the differences at the end of this subsection. It uses

[3] Per RFC 8446, only AES-GCM support is required. ChaCha20/Poly1305 is optional, but so widely used that support is de facto required. A third scheme, AES-CCM, is also optional; for both performance and security reasons, it is almost never supported [69].

[4] An equivalent formulation of the sub-circuit would not take M as input and instead deliver M' as output to the next sub-circuit in the pipeline.

ShortcutChanOpen$(CH, SH, C_{EE}^{\text{suff}}, C; HS, \mathsf{LCCV}, \mathsf{vsuff}, K, M)$:

$M' \leftarrow \mathsf{Dec}(K, C)$
$b \leftarrow$ CATKeyVerif-SC$(CH, SH, C_{EE}^{\text{suff}}, HS, \mathsf{LCCV}, \mathsf{vsuff}, K)$
Return $b \wedge (M' = M)$

CATKeyVerif-SC$(CH, SH, C_{EE}^{\text{suff}}, HS, \mathsf{LCCV}, \mathsf{vsuff}, K)$:

$SHTS \leftarrow \mathsf{hkdf.exp}(HS, H(CH||SH))$
$SHTK \leftarrow \mathsf{hkdf.exp}(SHTS, \text{“shtkey”})$
$fk_S \leftarrow \mathsf{hkdf.exp}(SHTS, \text{“fkey”})$
$h_7 \leftarrow f(\mathsf{LCCV}, \mathsf{PadLB}(\mathsf{vsuff}))$
$SF' \leftarrow \mathsf{HMAC}(fk_S, h_7)$
$SF \leftarrow \mathsf{Dec}(SHTK, C_{EE}^{\text{suff}})$ // At most 3 AES blocks.
$h_3 \leftarrow f(\mathsf{LCCV}, \mathsf{PadLB}(\mathsf{vsuff}||SF))$
$dHS \leftarrow \mathsf{hkdf.exp}(HS, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf.ext}(dHS)$
$CATS \leftarrow \mathsf{hkdf.exp}(MS, h_3)$
$CATK \leftarrow \mathsf{hkdf.exp}(CATS, \text{“catkey”})$
Return $SF' = SF \wedge CATK = K$

Figure 3: The shortcut channel opening circuit. Certain details, such as label inputs to hkdf.exp and TLS record headers, are omitted.

two functions we have not yet defined: f refers to the SHA-256 compression function and PadLB the padding function used for the last block of SHA-256. We call this the “shortcut” channel opening or ShortcutChanOpen. Its public input is the client and server Hello messages and the ciphertext C, but only the last 32 bytes of C_{EE}, which we denote C_{EE}^{suff}. Its private input is the handshake secret HS, the key K, and message M. It also takes values LCCV and vsuff which are, respectively, the last intermediate compression function output in common between the hashes h_7 and h_3, and the last full block input to the compression function to derive h_7.

The optimization happens in CATKeyVerif-SC, which works as follows. It first derives $SHTK$, which we used in CATKeyVerif to decrypt C_{EE}. Here we use it to decrypt only the last 32 bytes, C_{EE}^{suff}, containing the finished value SF. It then uses the chaining value LCCV and vsuff to complete the derivation of h_7, re-derives SF', and then appends SF to vsuff and applies f to recompute h_3. With HS and h_3, CATKeyVerif-SC re-derives $CATK$, checks it against K, and also checks that SF obtained from the ciphertext is the same as the re-computed one SF'.

This channel opening sub-circuit eliminates elliptic-curve operations and the expensive h_3 derivation. It can do this securely because it re-computes and checks SF against the one obtained from C_{EE}^{suff} — the server's finished value acts as a commitment to both HS and the hash h_7, which commits to the transcript. Because the inputs of h_7 and h_3 share a long common prefix, most of the chaining values output by f while deriving them will be the same. The sub-circuit takes as input the last common one (hence LCCV is the “last common chaining value”) to quickly re-derive both h_7 and h_3 with only one f call each. The sub-circuit checks SF to ensure that the prover did not lie about LCCV and vsuff.

This description of ShortcutChanOpen made a few simplifications. First, we assumed $(\mathsf{len}(EE) - 36) \bmod 64 \leq 48$, which means both h_7 and h_3 require only one more call to f. If this is not the case, the sub-circuit may need to do two compression function calls to compute h_3 and possibly h_7. Second, we give vsuff as a witness to the sub-circuit, but in our implementation and analysis of ShortcutChanOpen, we recompute vsuff by decrypting a few additional blocks of C_{EE} in the sub-circuit. Our security analysis suggests recomputing vsuff is not strictly necessary, but its concrete cost is fairly low and it may improve the concrete security of ShortcutChanOpen somewhat. Appendix C further discusses this point.

5.3 Security and Discussion

Appendix B describes another channel opening for data sent in a 0-RTT resumption handshake. Appendix C contains a provable-security analysis of the security of BaselineChanOpen and ShortcutChanOpen, showing that both prevent the client from equivocating about M or K during an interaction with an honest server.

Amortizing channel opening. Since some policies will apply to every packet the client sends, it will need to open every TLS record. Done naively, the client can regenerate the full proof for every new packet sent. The middlebox would have to keep track of the number of ciphertexts sent from the client to the server, so that it knows the correct sequence number for the ciphertext. (In TLS 1.3, a sequence number is exclusive-ORed into the AEAD's nonce before decryption.)

A much more efficient way to do channel opening would be to take advantage of the fact that in TLS 1.3, the client's key $CATK$ is usually fixed for the lifetime of a session. Thus, the cost of the key consistency check CATKeyVerif-SC in the channel opening sub-circuit can be amortized over every message that the client sends: at the beginning of the TLS session, the client computes and sends to the middlebox a short commitment to its session key (e.g. its hash h for some hash function H) and then proves that the committed value is consistent with the handshake using this CATKeyVerif-AM sub-circuit, which is essentially proving equality of two committed values:

CATKeyVerif-AM$(CH, SH, C_{EE}^{\text{suff}}, C, h; HS, \mathsf{LCCV}, \mathsf{vsuff}, K)$:

$h' \leftarrow H(K)$
$b \leftarrow$ CATKeyVerif-SC$(CH, SH, C_{EE}^{\text{suff}}, HS, \mathsf{LCCV}, \mathsf{vsuff}, K)$
Return $b \wedge (h' = h)$

AMChanOpen$(C, h; K, M)$:

Return $M = \mathsf{Dec}(K, C) \wedge h = H(K)$

The client can then use the above AMChanOpen subcircuit per-packet channel opening. For long-lived sessions, the amortized cost of channel opening would likely be lower than running committing AE decryption for every packet, since only one cryptographic pass would be required per packet versus at least two for (say) HMAC-then-CBC.

6 HTTP Firewalling

This section describes our first of three case studies of zero-knowledge middlebox protocols: verifying that outgoing encrypted connections contain HTTP packets.

Context and motivation. Many networks wish to block outbound connections that are not HTTP or HTTPS [111]. The standard enforcement mechanism is *port blocking*: discard all outbound traffic where the destination port is not equal to 80 or 443, the HTTP and HTTPS default ports. Port blocking is simple, does not require inspecting traffic directly, and imposes low latency.

However, it is a blunt instrument that leads to overblocking and underblocking. It overblocks (i.e. disallows legitimate connections) because it does not allow HTTP(S) connections to be established on non-default ports. It underblocks because it allows outbound non-HTTP(S) connections that use port 80 or 443. This case study outlines a ZKMB protocol that implements a much more precise outbound HTTPS firewall, by having clients prove all their outbound TLS connections contain HTTP requests.

Proof Details. The channel opening procedure will output the plaintext TLS record M. For simplicity, this case study will describe parse-and-extract and policy check circuits for a very specific check: that the request uses version 1.1 of HTTP.

The first line of any HTTP request must end with the version — `HTTP/1.1` indicates version 1.1 — and lines are delimited with the two-byte sequence `\r\n`. (For reference, see Section 7.3 for an example HTTP request.)

To check the version, the parse-and-extract circuit just needs to output last eight bytes prior to the first `\r\n` in the request. As depicted in the pseudocode in Figure 4, we implement this in a few steps: first, the circuit computes the index of the first `\r\n` using string matching (the MatchFirstCRLF procedure). The value i is computed in the circuit, rather than letting the prover specify it as an input, to ensure i is indeed the *first* `\r\n`, and not a later one — even if the circuit explicitly checked for a `\r\n` at the prover-specified index, a request containing the string `HTTP/1.1\r\n` at the end of some later line could be used to bypass the policy. We implemented MatchFirstCRLF as a linear scan over the request to minimize dynamic memory accesses. Second, the circuit compares i to the length ℓ. Since the request must be padded out to the largest possible length, this check ensures the prover does not insert `\r\n` into the padding. (Note that ℓ is public.)

```
HTTPFirewallPE(M, ℓ):
i ← MatchFirstCRLF(M)
b ← i < ℓ
LE ← M[i−8 : i−1]
If b then Return LE
Return ⊥

MatchFirstCRLF(M):
prev ← M[0] ; notfound ← 1 ; first_ind ← 0
For j ← 1 to ℓ_max:
  curr ← M[j]
  If prev||curr = \r\n then:
    first_ind +← notfound * (j − 1) ; notfound ← 0
  prev ← curr
Return first_ind
```

Figure 4: Parse-and-extract logic for HTTP firewalling.

Finally, if i is less than the message length, the circuit outputs bytes $i-8$ through $i-1$ of the input; else, it returns an error.

The policy-check circuit just outputs the result of comparing its input to the string `HTTP/1.1`. This is logically equivalent to the AND of eight clauses. To generalize to HTTP/2 traffic, this circuit can instead check that the start of the protocol stream has a request line that ends with `HTTP/2.0\r\n`.

7 ZKMBs for Encrypted DNS

Next we build a ZKMB that lets middleboxes block encrypted DNS queries, sent via DNS-over-TLS (DoT) or DNS-over-HTTPS (DoH), for domains on a pre-determined blocklist $\mathsf{BL} = \{D_1, D_2, \ldots, D_n\}$, which we assume is given to users during Registration.

After channel opening outputs the TLS plaintext M, the parse-and-extract step of Prove takes M as input and outputs the queried domain name D. The policy check step proves $D \notin \mathsf{BL}$. We also show how to keep the blocklist hidden from the client if necessary, and describe an extension for *allow*listing resolvers in Oblivious DoH.

This case study assumes the middlebox can distinguish encrypted DNS queries from other TLS traffic, and only requires proofs for the former. This can be done with a list of IP addresses of resolvers that support DoT/DoH.

7.1 Background: Encrypting DNS Queries

The Domain Name System (DNS) is the phone book of the Internet: it translates human-readable domain names (e.g., `example.com`) to machine-readable IP addresses (e.g. 142.250.190.14). While a user is browsing the web, their browser makes a DNS query to a resolver — a server that stores the name→IP mapping — to learn the IP address of

each site they visit.[5] This means a user's DNS queries reveal a great deal of information about their web browsing habits. As described in the introduction, DNS queries have historically been sent in the clear to a resolver controlled by the local network or Internet Service Provider (ISP).

Encrypting DNS queries with TLS 1.3 improves the privacy of DNS. There are two main standards for encrypted DNS: DoT [71] and DoH [68]. DoH has been supported by default for US users since mid-2020 by Mozilla [34], Google [5], and Apple [119]. Their browsers send users' queries to centralized encrypted DNS resolvers run by, e.g., Cloudflare [108] or Google [124]. As noted in the introduction, this arrangement prevents network operators from enforcing network policies at the DNS layer.

This has proven to be a major issue for network operators [29], and has impeded the deployment of DoT and DoH. The Google Chrome and Microsoft Edge browsers only allow "auto-upgrade": switching to DoH when the system's resolver is on a (small) list of resolvers known to support it. Where DNS encryption is the default, operator concern about filtering led to downgrade attacks being built into user-facing software to give networks the ability to disable DNS encryption without informing the user. For example, in the Firefox browser, DoH is enabled by default, but on browser startup the local network can selectively disable DoH by configuring its local DNS resolver to return an `NXDOMAIN` response to a (plaintext) DNS query for the canary domain `use-application-dns.net`. An early study found over 15% of networks utilizing this method to disable encrypted DNS [65]. Clearly, better solutions are needed than an all-or-nothing trade-off between privacy and network policy enforcement.

7.2 Parse-And-Extract for DoT

Request format. A DNS query canonically has two parts. A 12-byte *header* contains an identifier and metadata about the query. A variable-length *question* begins with a domain name, serialized in a length-label format (see below), and ends with four bytes indicating the question's type and class.

A domain name is a sequence of *labels* delimited by the period (full stop) character. For example, "www.example.com" has three labels: "www", "example", and "com". The serialization format of domain names is a sequence of length-label pairs, where each label is preceded by its length, ending with a single zero byte indicating the special "root" label at the top of the DNS hierarchy. The domain "www.example.com" is serialized as `3www7example3com0`.

Many extensions to this basic request format have been developed during the long lifetime of DNS, but they don't affect the invariant that we rely on, namely that the DNS question starts at the thirteenth byte of the request.

[5] For brevity, we present a greatly simplified picture of the DNS ecosystem and direct the interested reader to [70, 94, 95] for more details.

In DoT, DNS queries are sent in TLS records preceded by a two-byte length indicator [71]. Those bytes plus the twelve-byte header situate the serialized DNS question at the fifteenth byte of plaintext. The parsing task is thus to deserialize the bytes beginning at byte 15.

Deserializing DNS questions. The deserialization circuit need not handle arbitrary-length questions: domain names can be at most 255 bytes long. This allows us to deserialize with a single pass consisting of a fixed number of iterations over the request. Letting output be a length-255 array, and request be the request, our circuit is conceptually just an unrolled version of the following loop:

```
nlsi ← 0 // "next label start index"
stop ← 0
For i ← 0 to 255:
  If stop ≠ 1:
    If i ≠ nlsi:
      output[i] ← request[14 + i]
    If i = nlsi:
      If request[14 + i] = 0x00:
        stop ← 1
      Else:
        output[i] ← "."
        nlsi ← request[14 + i] + 1
Return output
```

The circuit just keeps track of the current index and current label length. When the current input byte is a length label, the circuit writes a period to the output; when the current byte is 0x00, the circuit stops copying bytes to the output. This is a somewhat unnatural approach; the advantage is that the accessed locations of request and output depend only on the loop index, thereby avoiding the overhead of random accesses (§3).

Our deserialization circuit has some limitations, and does not support all the features of a modern DNS query. One limitation is case-sensitivity: DNS questions are case insensitive to servers, but our circuit does not normalize the case of the domain name. The cost of doing this in the circuit would be very small. Our circuit also does not handle internationalized domain names (IDNs) [44, 72, 77]. DNS clients convert non-ASCII labels in domain names to an ASCII-compatible encoding called Punycode [31]. (Punycode handles case sensitivity issues as well [66, 67].) A naive solution would be to decode Punycode to some other encoding (e.g. UTF-8) in the circuit. However, this decoding algorithm is complicated and would likely incur expense in circuits, as it requires data-dependent memory accesses. A better solution would adapt the policy check to use Punycode as its string representation, if possible — this would obviate the need for Punycode-decoding in the circuit. We leave the details to future work.

7.3 Parse-and-Extract for DoH

Parse-and-extract for DoH is slightly more complex than DoT, because the query name is not guaranteed to begin at a fixed plaintext offset. DoH can use HTTP `GET` or `POST`; the circuit determines which it is by looking at the first four bytes of the plaintext HTTP request (for the strings "GET" or "POST").

DoH `POST`. In this case, the query is sent in the request body. Although the starting location of an HTTP request body is not fixed — it is determined by the header lengths — it is unambiguously delimited by the byte sequence "`\r\n\r\n`" [99]. Thus, the prover can give the location of this sequence as a witness to the circuit, which can check its validity (cf.§6).

```
GET /dns-query?dns=<b64 DNS query> HTTP/1.1\r\n
Host: cloudflare-dns.com:443\r\n
Accept-Encoding: identity\r\n
accept: application/dns-message\r\n
\r\n
```

DoH `GET`. An example DoH `GET` request is above. In a `GET`, the DNS query is the value of an HTTP query parameter, usually "dns". Its location is not fixed and is affected by the presence or absence of other query parameters. This is handled similarly to the `POST` case, verifying the starting location with the string "dns=" instead of "`\r\n\r\n`".

7.4 Policy Check for Domain Blocklisting

The policy check circuit takes as input a domain name (as output in period-delimited format by a parse-and-extract circuit) and verifies that it is not in a set of blocked domains. For this, we use Merkle tree non-membership proofs: by sorting the blocklist and computing a Merkle root, non-membership of a domain can be proved by exhibiting two elements that are adjacent in the Merkle tree and lexicographically bracket the queried domain [78]. We implemented this for exact-match blocklisting, using Poseidon [58] for circuit efficiency.

Handling wildcards. We would also like to support more complex blocklisting policies. Blocklisting tools have varied semantics, but mandatory sub-domain blocking appears very common. For example, if `example.com` is on the blocklist, any sub-domain like `foo.example.com` will be blocked. Essentially, each entry on the blocklist has an implied "*." in front of it, and our policy check circuit must prove that no suffix of the input domain is on the blocklist.

We handle this with a variant of the exact-match non-membership circuit. As above, our circuit takes two Merkle paths of adjacent list elements. However, our circuit assumes the strings in the list to have been reversed (e.g., the domain "example.com" is sorted as "moc.elpmaxe") before sorting and Merkle tree computation; reversing the strings ensures that a string's position in the list is determined by its suffix. The circuit checks the Merkle paths of the two adjacent list elements, and also checks that neither element is a proper prefix of the (reversed) input. Note that this last check respects ".", so that (e.g.) "bexample.com" is not blocked if "example.com" is on the blocklist.

7.4.1 Keeping the Blocklist Private

Above, we assumed the blocklist is visible to clients. This is good for transparency, since any user could in principle review the network's policy. However, a client-visible blocklist is infeasible if (e.g.) the blocklist is proprietary.

Here, we outline a potential extension to our protocol which can address this issue by allowing the middlebox to hide the blocklist from the client, while still letting the client prove that its queried domain D is not blocked. This can be achieved using an oblivious pseudorandom function (OPRF). For concreteness, we use the simple and efficient OPRF of Naor and Reingold [96], defined over a cyclic group $\mathbb{G}$ of order p: $F(x,m) := H(m)^x$ where $H: \{0,1\}^* \to \mathbb{G}$ and $x \in [1,\ldots,p]$.

Rather than sending the client the plaintext blocklist $\mathsf{BL} \leftarrow \{D_1,\ldots,D_n\}$ during Registration, the middlebox generates a random OPRF key x and gives the client $\mathsf{BL}_F \leftarrow \{F(x,D_1),\ldots,F(x,D_n)\}$. Then Prove begins with a first round where the client and middlebox $F(x,D)$ on the domain name D that the client intends to query, in three steps: first, the client generates a random key r and sends the middlebox $B \leftarrow F(r,D)$. Second, the middlebox replies with $BV \leftarrow B^x$. Finally, the client computes $F(x,D) \leftarrow BV^{1/r}$.

In the second round of Prove, the client generates and sends its proof. The circuit is as above, except it takes $F(x,D)$ and r as private input, and B and BV as public input. The circuit verifies a non-membership proof that $F(x,D) \notin \mathsf{BL}_F$ and uses r to "open" the OPRF transcript and prove that its DNS query in fact contains D: the circuit checks that $F(r,D) = B$ and that $BV^{1/r}$ was the value supplied to the non-membership proof.

The OPRF hides unqueried blocklist entries from the client. Also, the blinded value B hides the D from the middlebox, and the client cannot use different D in the first and second rounds: if H is a random oracle and discrete log is hard in $\mathbb{G}$, B is a commitment to D, and the circuit verifies that parse-and-extract outputs D and that $F(r,D) = B$.

The client learns $D \in \mathsf{BL}$ for only one D per execution, comparable to learning the blocklist of a filtering (non-ZK) middlebox by sending probe packets and seeing which are transmitted. However, in that case the middlebox learns which domains have been probed. With our approach, the middlebox can't tell which blocked domains clients have probed.

7.5 Oblivious DoH

Our third case study shows how a middlebox can verify that a client's DNS queries are destined for a filtered resolver, even when the client is using the *Oblivious DoH* (ODoH) protocol.

ODoH is an extension to DoH that enables stronger privacy by adding a *proxy resolver* between the client and *target resolver*: the client encrypts (using HPKE [6]) the actual DNS query with the target resolver's public key and sends it, along with the target's identity, to the proxy resolver via HTTPS. (More precisely, the target resolver's identity is sent as the value of the `targethost` query parameter.) This hides the client's identity from the target resolver. ODoH is motivated by the fact that in practice DoH resolvers are run by a few large companies like Google and Cloudflare.

ODoH is incompatible with the increasingly common practice of networks enforcing filtering by directing their users to a filtered third-party resolver: by design it prevents the local network from seeing the target resolver's IP address and thus prevents a non-ZK middlebox from blocking traffic to non-allowed target resolvers.

With a ZKMB, users can prove to the middlebox that their ODoH query is destined for a specific target resolver. For brevity, we sketch our circuit, omitting most details. Since the target resolver is sent in a query parameter, a modified version of the DoH parse-and-extract circuit outputs the `targethost` parameter. Then the policy check ensures equality between this string and the domain name of the filtered third-party resolver. To handle more than one third-party resolver, one could use Merkle-tree-based set membership proofs.

To handle ODoH domain blocklisting, the channel opening phase would have to nest a ZKMB-style circuit that decrypts and parses the HPKE ciphertext. Doing this securely is somewhat involved: to prevent the client from equivocating about the HPKE decryption key, the circuit must verify that the public key used by the client is in fact the correct key for the destination ODoH resolver identified in the request. This can be done by performing, in the circuit, a lookup against a public mapping between identifiers of ODoH resolvers and HPKE public keys. As a benefit, this approach hides the destination resolver from the middlebox. We leave the implementation and evaluation of ODoH blocklisting to future work.

8 Empirical Evaluation

Our empirical evaluation addresses three questions: (1) What is the baseline performance of channel opening and how effective are our optimizations? (§5), (2) How performant are our protocols for the case studies? (§6 and §7), and (3) Are the combined costs tolerable for real-world use? If not, how far from tolerable are they?

Implementation. We implemented all protocols described in Sections 5–7, except the private blocklist extension of §7.4.1. Our implementation framework is xJsnark, a high-level language for writing zero-knowledge proofs [79]. We use xJsnark's gadget library extensively, in particular its AES and SHA-256 gadgets and its efficient random-access memory type. The xJsnark system uses libsnark for all backend proof implementations; we use libsnark's implementation of the Groth16 [59] protocol with the BN128 curve as our backend.

Method	**Gates**$\times 10^5$	**Time (s)**	**SRS (MB)**
BaselineChanOpen	76.7	96	1200
ShortcutChanOpen	13.0	18	175
EarlyChanOpen	6.3	9	82
Amortized	1.9	4	27

Figure 5: Gate count, proof generation time, and CRS size for opening a 255-byte ciphertext with each channel opening circuit described in Section 5. Verification time was less than 5 milliseconds in all cases; verification key size was less than one megabyte.

Harness and testbed. We wrote an experiment harness in Python to generate inputs for circuits: transcripts of TLS sessions and network traffic like DNS queries and HTTP requests. We heavily modified the tlslite-ng library [121] to give us the ability to extract its internal state. We also used this harness to compute the Merkle paths for our blocklisting policy in Section 7. We run all our experiments using an Amazon EC2 instance `t3a.2xlarge`, with a (virtual) 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 gigabytes of RAM.

8.1 Microbenchmarks for Channel Opening

We performed microbenchmarks of the channel opening circuits described in Section 5, including the "amortized" channel opening circuit, where the client's key consistency check is simply verifying a hash (we use Poseidon) of the key. We measure the size (in multiplication gates) of the circuit output by xJsnark, and the proof generation time (median of 5 trials), SRS size, and verification time for opening a 255-byte ciphertext. For the BaselineChanOpen experiment, we use an upper bound of 3000 bytes on the size of C_{EE} (the server's extensions). From the (small) suite of the algorithms supported by TLS 1.3, we use AES-128-GCM [42, 93], SHA-256 [47], and ECDHE over the secp256r1 (NIST P-256 [74]) curve in all cases.

Figure 5 lists the results. The baseline protocol is the least efficient in all three metrics. ShortcutChanOpen improves on the baseline dramatically: proving time is 5× faster, and the circuit has 6× fewer gates. This shows that removing elliptic curve operations and the h_3 derivation produces substantial benefit. Without the roughly four-second cost of AES decryption, the time taken (14s) is roughly what CATKeyVerif-AM would take to produce a commitment to the session key. This means the one-time setup cost for a TLS session is around 14 seconds. EarlyChanOpen gives still more improvement, a factor of two improvement over ShortcutChanOpen circuit. The amortized channel opening is the most efficient in all three metrics, and is also the simplest: the only substantial cost is evaluating AES-128 16 times (on the same key).

Case Study	Gates$\times 10^5$	Time (s)	SRS (MB)
HTTP Firewall (§6)	15.1	22	207
DoT (§7.2)	13.9	21	197
DoH GET (§7.3)	15.9	23	235
ODoH (§7.5)	4.7	8	72

Figure 6: Gate count for Prove circuit, proof generation time, and SRS size for each case study. The ODoH results use the Amortized channel opening and the rest use the ShortcutChanOpen. More details on concrete parameters are given in the text.

8.2 Full benchmarks for case studies

Next we evaluate the performance of a full Prove implementation for each of our three case studies (§6 and §7). For each experiment, we use ShortcutChanOpen for channel opening, then apply the parse-and-extract and policy check circuits described in the corresponding section.

HTTP Firewalling (§6). For this experiment, we evaluated Prove on a 500-byte ciphertext. HTTP requests can be larger than this, but since this policy only concerns the first line of a given request, the prover only needs to decrypt a prefix of the ciphertext long enough to contain the first line.

The results are shown in the first row of Figure 6. For this case study, the dominant cost, by far, is channel opening: ShortcutChanOpen by itself costs 1.5 million gates, of which about 400,000 are for the AES computations needed to decrypt the ciphertext; the other two steps together take under 20,000 gates. The parse-and-extract and policy check steps are simple here, but even for more complex examples below we will see that channel opening is usually the dominant cost.

Domain Blocklisting for DoT/DoH (§7). For this experiment, we use the Python experiment harness described above to create the Merkle tree and compute the Merkle paths for the non-membership proof. We use the Poseidon [58] hash function in constructing the Merkle tree. Because of some subtleties in the default parameterization of Poseidon, our current implementation can only support 253-byte domain names, one byte less than the real limit of domain name length in DNS. We use a blocklist containing two million entries, obtained from an open-source online blocklist of adult-content domains [92], which we believe to comparable to the size of proprietary adult content domain blocklists [138]. Because the blocklist must be downloaded by the client during Registration, a practically-relevant question is the size of the blocklist. Compressed with gzip, it is just over seven megabytes, and its uncompressed size is fifty megabytes — thus, the cost of downloading and storing the blocklist is relatively small.

Creating the Merkle tree for this blocklist is expensive: it takes nearly two hours. The hash tree itself is over 500 megabytes (roughly two million 254-bit Poseidon hashes). Computing the Merkle tree is a one-time cost for the client; even still, two hours is impractical. The high cost here is likely because of our unoptimized Python implementation of Poseidon.

For our DoT experiment, we use a 255-byte ciphertext with ShortcutChanOpen; since domain names can only be 255 bytes, it is unlikely that a DoT ciphertext would be larger than this. Overall, the cost is 1.4 million gates and 21 seconds to generate a proof. ShortcutChanOpen proves to be the dominant cost in terms of circuit size: 1.3 million gates are needed for channel opening, including nearly 200,000 for decrypting a 255-byte ciphertext. Parse-and-extract and policy check take only 90,000 gates. Of note is the small size and time needed for the non-membership check: even for our fairly large blocklist, proving non-membership costs only 40,000 gates. This is because Merkle paths are compact and Poseidon's circuit is very small, costing only 250 gates to evaluate.

We next evaluate DoH GET with ShortcutChanOpen and a 500 byte ciphertext, to accommodate possible HTTP headers in addition to an at-most 255-byte domain query. The cost is slightly higher than the DoT circuit: 1.6 million gates and 23 seconds to generate a proof. ShortcutChanOpen is again the dominant cost, with 1.5 million gates for channel opening, including 400,000 for decrypting the 500-byte ciphertext. The parse-and-extract policy check costs about 100,000 gates.

Resolver Allowlisting for ODoH (§7.5). We evaluate our ODoH case study described in Section 7.5 with the amortized channel opening circuit for a 500-byte ciphertext. We choose this both to get a sense of the improvement of amortization, and because amortizing makes sense for ODoH: clients will likely have long-lived connections to a single proxy. Our current implementation of the policy check circuit only supports a one-entry allowlist, and cannot yet use Merkle proofs.

This experiment results in the best performance in all three metrics: the circuit is only around 470,000 gates, of which nearly 400,000 are AES decryption. Proving time is only about a third of the other case studies as well.

Summary. Our goal in doing these experiments was to understand how our ZKMB protocols perform, and whether their costs are tolerable for real-world use. Overall, our experiments indicate ZKMBs are not yet practical, primarily because proof generation times are still too high: our most efficient channel opening circuit for a full TLS 1.3 handshake takes eighteen seconds to generate. Subsequent per-packet proofs take between four and eight seconds to generate, even for our most efficient case study. SRS sizes are another practical barrier to deployment: up to 235 megabytes. However, these numbers represent a huge improvement over a naive implementation: the baseline channel opening circuit takes 96 seconds for proof generation, and its SRS is 1200 megabytes.

9 Related Work

Implementations of probabilistic proofs. Probabilistic proofs are fundamental in cryptography and complexity theory [2–4, 53–55]. At a high level, they allow a *verifier* to efficiently and probabilistically check a claim made by a *prover*. Over the last decade, and accelerating over the last few years, there has been intense interest in refining and implementing probabilistic proofs, with particular interest in proofs with zero-knowledge properties. See [120, 125, 128] for surveys.

Built systems generally involve a *back-end* (a probabilistic proof protocol) and a *front-end* (a mechanism for transforming from a high-level computation to a circuit). We use the Groth16 [59] back-end, summarized earlier (§3). Groth16 is built atop the seminal QAP formalism [52], which was initially implemented in Pinocchio [104] (see also BCTV [10]). All of these works have the qualitative performance described earlier (§3, §8). For relatively recent comparisons of Groth16 and other strands of back-ends, see Hyrax [127, §8] and Spartan [112, §9]. Subsequent work has not changed the qualitative costs and trade-offs, though there has been progress in relaxing setup assumptions [1, 8, 11, 22, 25, 26, 50, 87, 88, 112, 113, 127, 136] and extending the machinery to circuits and statements of greater size and scope via recursive proof composition [17, 21, 51, 80, 81].

Turning to front-end work, there are two main approaches: "ASIC" and "CPU" [125, 128]. We use the xJsnark [79] front-end, which is an "ASIC" approach, meaning that each high-level computation transforms to a custom circuit $\mathcal{C}$. A program compiler works over the high-level computation, going line-by-line to produce corresponding arithmetic circuit gates (or constraints), using various techniques to produce concise encodings when possible [18, 19, 28, 32, 102, 104, 105, 114, 126, 139]. A variant of this approach is to hand-design circuits by exploiting recent enhancements to arithmetization [15, 49]. In contrast, the "CPU" front-end approach encodes a "universal machine" in a circuit [8–10, 137]. This approach, while in principle facilitating programmability, results in performance that is not competitive (overhead of 50× [126]).

Applications of general-purpose ZK proofs have so far mainly surrounded cryptocurrency [110] (with exploding commercial interest in blockchain applications), though a recently launched program by DARPA, called SIEVE, seeks to develop applications beyond that sphere. To our knowledge, no prior work has applied this machinery to network security.

Middlebox architectures. Many works have proposed alternate middlebox architectures; see Sherry's dissertation for an overview [116]. Here, we restrict focus to work that, like ZKMBs, seeks to balance privacy and functionality; see mbTLS for a thoughtful framework [97]. Prior works do not meet one or both facets of our Compatibility requirement (§1):

TEE-based. ETTM [36] creatively re-imagines network architecture, placing middlebox functionality in a trusted virtual machine on the end host, which requires trusted hardware (to attest to, and boot, the hypervisor). Endbox [56] refines this vision, placing the middlebox code in a trusted execution environment (TEE), such as an SGX enclave, on end hosts. A variant of this architecture runs the middlebox code in an in-cloud TEE [64, 106, 123]; the motivation is enterprise-deployed middleboxes running on untrusted cloud platforms. These works and many others [41, 57, 63, 82, 129] rely on trusted enclaves.

Application-specific protocol modifications. Like ZKMBs, Blindbox [115] is based on advanced cryptographic tools, specifically Oblivious Transfer, circuit garbling, and searchable encryption. Blindbox encrypts all traffic with ordinary TLS and a separate, weaker encryption that allows the middlebox to compute obliviously. Blindbox requires changes to servers, which must check the consistency of the TLS contents and the weaker encryption, otherwise the client can equivocate (§5). Successive work in the same model includes Embark [85] and many others [45, 75, 83, 84, 86, 90, 100, 101, 107, 132, 134]. All of these proposals require server-side changes, which we regard as incompatible with widespread adoption.

Another approach is mcTLS [98], which extensively modifies TLS to reveal plaintext traffic to middleboxes, albeit smaller, application-defined parts of the plaintext. This privacy model is weaker than that offered by ZKMBs (and besides has been shown to be insecure [12]).

Proving properties of encrypted data. Some prior research shares a high-level goal with ZKMBs: proving something about the (hidden) plaintext of a (public) ciphertext. One such line of work develops verifiable encryption (VE) schemes [24], which are public-key encryption schemes that enable efficient generation of proofs for predetermined functions of plaintexts (for example, that the message is the discrete log of a public group element). Each function usually requires its own dedicated VE scheme. ZKMBs are conceptually similar to VE; however, VE is not compatible with existing protocols, so VE techniques are inapplicable to ZKMBs.

Two more recent papers share our compatibility requirement (§1). Wang et al. [130] build a certificate authority that can issue certificates to anonymous users. One step of their certificate issuance protocol involves proving that part of a TLS 1.2 record has a given hash value. They do not consider TLS 1.3, and the goals and techniques of their work are otherwise distinct from ours.

Zhang et al. [135] build a "decentralized oracle" (DECO) primitive that allows a user to reveal server responses to a non-inline third party. Their construction relies on the user and third party jointly running a TLS client in MPC. Clients have the option of proving statements about the responses using a ZK proof system instead of revealing them. In an appendix, they suggest having the third party act as an inline proxy, and show that this reduces the overhead of MPC.

Zhang et al. make a few important observations that prefigure some of the technical challenges in building ZKMBs:

first, as described in Section 5, they recognize the need for TLS records to commit to plaintexts. They also highlight the risk of the prover equivocating the message by lying about its structure. The authors address this via specialized subcircuits for extracting contents from JSON responses, essentially a special case of the parse-and-extract step in our proof pipeline. However, there are several crucial differences between their work and ours. First, the high-level goal of a decentralized oracle is to let the client prove statements about TLS responses from servers; in contrast, in ZKMBs, clients prove statements about their own traffic. Second, their ZK proof protocols rely on the prover and verifier having a secret-shared version of the TLS session key output from their MPC protocol. Finally, their focus throughout is on TLS 1.2. They heavily rely on TLS 1.2's HMAC-then-CBC cipher suite, which is not available in TLS 1.3.

10 Summary, Discussion, and Future Work

To summarize, this paper introduced ZKMBs (§2, §4), thereby identifying a novel application of general-purpose ZK proofs. Instantiating secure and efficient protocols, however, was easier said than done. The biggest challenge was representing enough of the TLS handshake in the circuit formalism (§3) to achieve the needed binding property (§5) but not so much that the circuit would inflate (which would destroy performance). Another challenge was embedding the specific case study functionality (§6–§7) in circuits.

In our experiments (§8), proof length is 128 bytes, and the verifier (middlebox) runs in 2–5ms. These quantities are inherited from Groth16 [59]. Two aspects of performance give us pause. First, the prover's (client's) running time: around 14 seconds to initiate a TLS connection and several seconds for any packets that call for proofs. On the other hand, this cost represents a 6× improvement over an unoptimized baseline, with more improvement expected in future work (see below).

Second, the SRS (§3) is large (hundreds of megabytes). This raises the question: how does the SRS get to clients? If each middlebox generates its own, clients have to download a large package. Another alternative is for a trusted entity (such as a browser vendor) to install on clients SRSes for appropriate, fixed circuits (§2). Thus, there is a tradeoff between costs and trust. (However, as stated in Section 2, even if the SRS is generated adversarially the client's privacy is not affected [48].) Notice that if the middlebox needs to update policy, pre-installation is not an option, but it might still be advantageous to have browser vendors disseminate SRS updates, perhaps with browser upgrades.

Several items in our paper are clear targets for future work: implementing the private blocklist technique (§7.4.1), supporting application requests that span several TLS records, and implementing functionality, such as IDS and malware scanning, that necessitates proofs per packet. Other future work is using TLS 1.3's ChaCha20/Poly1305 encryption as an alternative to AES-GCM; the ChaCha20 function's simpler structure may yield smaller channel opening circuits and reduced proof generation times.

Several larger extensions can also be explored. One is *circuit decomposition*, in which each of channel opening, parse and extract, and policy-check are separate circuits, each with a separate SRS. Each circuit would output a commitment, which would be opened in the successive circuit. This arrangement would allow the middlebox to run its own ZKSetup for the smaller circuits (parse-and-extract and policy-check), while trusted parties could install the infrequently-changing channel-opening circuit on clients. One systems question is what language the middlebox should use for expressing policy to clients, so that the client's networking stack can prevent non-compliant messages, and so that the client and middlebox can agree on the relevant circuits.

Another extension is alternate back-end proof protocols (§9). Universal SNARKs [25, 50, 91] are proof protocols in which a single SRS can be applied to different circuits of the same length. Using this machinery would ease the tradeoff between setup costs and trust (because now the setup has to be done once, without ongoing revision), at the possible cost of more circuit design work and larger proofs. Also intriguing are backends that achieve excellent prover overhead [1, 11, 88, 112, 136] and require no trusted setup, though at the expense of larger verifier running time and larger proofs.

An interesting direction is extending ZKMBs to other middlebox architectures. We have been presuming that the middlebox is located in the same network as the client. We expect the ideas here to translate naturally to settings in which the middlebox is placed differently, for example interposed in front of the server, or running "off-path" in the cloud (§9).

Finally, the most challenging task for future work is an implementation of a ZKMB middlebox that runs at the network's line rate. We expect this to require hardware acceleration of one kind or another.

Acknowledgements

The authors are indebted to Chris Wood for his suggestions and insights about DNS throughout this project. The authors thank Srikar Varadaraj for his help during the early stages of this work. The authors also thank Malavika Balachandran Tadeusz, Bill Budington, Richard Clayton, Henry Corrigan-Gibbs, Felix Günther, Anurag Khandelwal, Ian Miers, Eric Rescorla, Justin Thaler, and Pete Zimmerman for helpful questions and comments. This research was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, October 2017.

[2] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof Verification and the Hardness of Approximation Problems. *Journal of the ACM*, 45(3), 1998.

[3] Sanjeev Arora and Shmuel Safra. Probabilistic Checking of Proofs: A New Characterization of NP. *Journal of the ACM*, 45(1), 1998.

[4] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. Checking Computations in Polylogarithmic Time. In *ACM STOC*, 1991.

[5] Kenji Baheux. A safer and more private browsing experience with Secure DNS. *Chromium Blog*, May 2020.

[6] R.L. Barnes, K. Bhargavan, and C. Wood. Hybrid public key encryption, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hpke-04.

[7] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *IACR EUROCRYPT*, 2006.

[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://ia.cr/2018/046.

[9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *IACR CRYPTO*, 2013.

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Usenix Security*, August 2014.

[11] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *ACM CCS*, October 2020.

[12] Karthikeyan Bhargavan, Ioana Boureanu, Antoine Delignat-Lavaud, Pierre-Alain Fouque, and Cristina Onete. A formal treatment of accountable proxying over TLS. In *IEEE Security & Privacy*, 2018.

[13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, January 2012.

[14] Manuel Blum, Paul Feldman, and Silvio Micali. Non-Interactive Zero-Knowledge and its Applications. In *ACM STOC*, 1988.

[15] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Nearly linear-time zero-knowledge proofs for correct program execution. Cryptology ePrint Archive, Report 2018/380, 2018. https://ia.cr/2018/380.

[16] Stéphane Bortzmeyer. DNS Privacy Considerations. RFC 7626, 2015.

[17] Sean Bowe, Jack Grigg, and Daira Hopwood. Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://ia.cr/2019/1021.

[18] Benjamin Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, December 2012.

[19] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SOSP*, November 2013.

[20] Jon Brodkin. AT&T to end targeted ads program, give all users lowest available price. *Ars Technica*, 2016.

[21] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data from accumulation schemes. Cryptology ePrint Archive, Report 2020/499, 2020. https://ia.cr/2020/499.

[22] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *IACR EUROCRYPT*, 2020.

[23] Jonas Bushart and Christian Rossow. Padding Ain't Enough: Assessing the Privacy Guarantees of Encrypted DNS. In *USENIX FOCI*, 2020.

[24] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *IACR CRYPTO*, 2003.

[25] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *IACR EUROCRYPT*, 2020.

[26] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *IACR EUROCRYPT*, May 2020.

[27] Children's Internet Protection Act. 106th Congress Public Law 554, 2000.

[28] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Report 2021/651, 2021. https://ia.cr/2021/651.

[29] Catalin Cimpanu. UK ISP group names Mozilla 'Internet Villain' for supporting 'DNS-over-HTTPS'. *ZDNet*, 2019.

[30] D.D. Clark, J. Wroclawski, K.R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's internet. *IEEE/ACM Transactions on Networking*, 13(3), 2005. First appeared in SIGCOMM 2002.

[31] Adam Costello. Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA). RFC 3492, 2003.

[32] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Security & Privacy*, 2015.

[33] Xavier de Carné de Carnavalet and Paul C. van Oorschot. A survey and analysis of TLS interception mechanisms and motivations, 2020. arXiv 2010.16388, https://arxiv.org/abs/2010.16388.

[34] Selena Deckelmann. Firefox continues push to bring DNS over HTTPS by default for US users. *Mozilla Blog*, February 2020.

[35] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[36] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A scalable fault tolerant network manager. In *USENIX NSDI*, March 2011.

[37] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *IACR CRYPTO*, 2018.

[38] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *IACR Journal of Cryptology*, 2020.

[39] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[40] Ralph Droms and Steve Alexander. DHCP Options and BOOTP Vendor Extensions. RFC 2132, 1997.

[41] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *arXiv preprint arXiv:1706.06261*, 2017.

[42] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (AES). NIST FIPS 197, 2001.

[43] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *ACM IMC*, 2015.

[44] Patrick Faltstrom, Paul Hoffman, and Adam Costello. Internationalizing Domain Names in Applications (IDNA). RFC 3490, 2003.

[45] Jingyuan Fan, Chaowen Guan, Kui Ren, Yong Cui, and Chunming Qiao. Spabox: Safeguarding privacy during deep packet inspection at a middlebox. *IEEE/ACM Transactions on Networking*, 25(6), 2017.

[46] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *IACR CRYPTO*, 1986.

[47] Secure hash standard. NIST FIPS 180-2, 2002.

[48] Georg Fuchsbauer. Subversion-zero-knowledge snarks. In *IACR PKC*, 2018.

[49] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020. https://ia.cr/2020/315.

[50] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://ia.cr/2019/953.

[51] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. Cryptology ePrint Archive, Report 2021/529, 2021. https://ia.cr/2021/529.pdf.

[52] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *IACR EUROCRYPT*, 2013. https://ia.cr/2012/215.

[53] Oded Goldreich. Probabilistic proof systems – a primer. *Foundations and Trends in Theoretical Computer Science*, 3(1), 2008.

[54] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. *Journal of the ACM*, 62(4), 2015.

[55] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 1989.

[56] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *IEEE/IFIP DSN*, 2018.

[57] Deli Gong, Muoi Tran, Shweta Shinde, Hao Jin, Vyas Sekar, Prateek Saxena, and Min Suk Kang. Practical verifiable in-network filtering for DDoS defense. In *2019 IEEE ICDCS*, 2019.

[58] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *Usenix Security*, 2021.

[59] Jens Groth. On the size of pairing-based non-interactive arguments. In *IACR EUROCRYPT*, 2016.

[60] Christian Grothoff, Matthias Wachs, Monika Ermert, and Jacob Appelbaum. Toward secure name resolution on the Internet. *Computers & Security*, 77, 2018.

[61] Electronic Frontier Foundation & Online Policy Group. Internet blocking in public schools, 2003. https://www.eff.org/files/filenode/net_block_report.pdf.

[62] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *IACR CRYPTO*, 2017.

[63] Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han. A secure middlebox framework for enabling visibility over multiple encryption protocols. *IEEE/ACM Transactions on Networking*, 28(6), 2020.

[64] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017.

[65] Wes Hardaker. Measuring the size of Mozilla's DOH Canary Domain. https://www.isi.edu/~hardaker/news/20191120-canary-domain-measuring.html, 2019.

[66] Paul Hoffman and Marc Blanchet. Preparation of Internationalized Strings ("stringprep"). RFC 3454, 2002.

[67] Paul Hoffman and Marc Blanchet. Nameprep: A Stringprep Profile for Internationalized Domain Names. RFC 3491, 2003.

[68] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, 2018.

[69] Ralph Holz, Johanna Amann, Abbas Razaghpanah, and Narseo Vallina-Rodriguez. The era of TLS 1.3: Measuring deployment and use with active and passive methods. *arXiv preprint arXiv:1907.12762*, 2019.

[70] Austin Hounsel, Paul Schmitt, Kevin Borgolte, and Nick Feamster. Designing for tussle in (encrypted) DNS. *arXiv preprint arXiv:2002.09055*, 2020.

[71] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. Specification for DNS over Transport Layer Security (TLS). RFC 7858, 2016.

[72] Internationalized domain names. https://en.wikipedia.org/wiki/Internationalized_domain_name, 2021.

[73] Jacob Kastrenakes. It's official: your Internet provider can share your web history. *The Verge*, 2017.

[74] Cameron F Kerry and Patrick D Gallagher. Digital signature standard (DSS). NIST FIPS 186-4, 2013.

[75] Jongkil Kim, Seyit Camtepe, Joonsang Baek, Willy Susilo, Josef Pieprzyk, and Surya Nepal. P2DPI: Practical and Privacy-Preserving Deep Packet Inspection. *AsiaCCS*, 2021. https://ia.cr/2021/789.

[76] Eric Kinnear, Patrick McManus, Tommy Pauly, Tanya Verma, and Christopher A. Wood. Oblivious DNS Over HTTPS. Internet-Draft draft-pauly-dprive-oblivious-doh-06, Internet Engineering Task Force, 2021. Work in Progress.

[77] John Klensin. Internationalized Domain Names in Applications (IDNA): Protocol. RFC 5891, 2010.

[78] Paul C Kocher. On certificate revocation and validation. In *International conference on financial cryptography*. Springer, 1998.

[79] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xJsnark: a framework for efficient verifiable computation. In *IEEE Security & Privacy*, 2018.

[80] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. Poppins: A direct construction for asymptotically optimal zksnarks. Cryptology ePrint Archive, Report 2020/1318, 2020. https://ia.cr/2020/1318.

[81] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. Cryptology ePrint Archive, Report 2021/370, 2021. https://ia.cr/2021/370.

[82] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel software guard extension (Intel SGX). *arXiv preprint arXiv:1802.00508*, 2018.

[83] Shangqi Lai, Xingliang Yuan, Joseph K Liu, Xun Yi, Qi Li, Dongxi Liu, and Surya Nepal. Oblivsketch: Oblivious network measurement as a cloud service. In *ISOC NDSS*, 2021.

[84] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, and Dongxi Liu. Practical encrypted network traffic pattern matching for secure middleboxes. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[85] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *USENIX NSDI*, 2016.

[86] Hyunwoo Lee, Zach Smith, Junghwan Lim, Gyeongjae Choi, Selin Chun, Taejoong Chung, and Ted Taekyoung Kwon. maTLS: How to make TLS middlebox-aware? In *ISOC NDSS*, 2019.

[87] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. Cryptology ePrint Archive, Report 2020/1274, 2020. https://ia.cr/2020/1274.

[88] Jonathan Lee, Srinath Setty, Justin Thaler, and Riad Wahby. Linear-time and post-quantum zero-knowledge SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/030, 2021. https://ia.cr/2021/030.

[89] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In *USENIX Security*, 2021.

[90] Cong Liu, Yong Cui, Kun Tan, Quan Fan, Kui Ren, and Jianping Wu. Building generic scalable middlebox services over encrypted protocols. In *IEEE INFOCOM*, 2018.

[91] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *ACM CCS*, 2019.

[92] Chad Mayfield. my-pihole-blocklists/pi_blocklist_porn_all.list. https://github.com/chadmayfield/my-pihole-blocklists, 2021.

[93] David McGrew and John Viega. The galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[94] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, 1987.

[95] P. Mockapetris. Domain names - implementation and specification. RFC 1035, 1987.

[96] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *IEEE FOCS*, 1997.

[97] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[98] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. *ACM SIGCOMM Computer Communication Review*, 45(4), 2015.

[99] Henrik Nielsen, Jeffrey Mogul, Larry M Masinter, Roy T. Fielding, Jim Gettys, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, 1999.

[100] Jianting Ning, Xinyi Huang, Geong Sen Poh, Shengmin Xu, Jia-Chng Loh, Jian Weng, and Robert H Deng. Pine: Enabling privacy-preserving deep packet inspection on TLS with rule-hiding and fast connection establishment. In *ESORICS*, 2020.

[101] Jianting Ning, Geong Sen Poh, Jia-Ch'ng Loh, Jason Chia, and Ee-Chien Chang. PrivDPI: privacy-preserving encrypted traffic inspection with reusable obfuscated rules. In *ACM CCS*, 2019.

[102] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Report 2020/1586, 2020. https://ia.cr/2020/1586.

[103] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security*, 2020.

[104] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Security & Privacy*, 2013.

[105] Pequin: A system for verifying outsourced computations, and applying SNARKs. https://github.com/pepper-project/pequin.

[106] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *USENIX NSDI*, 2018.

[107] Geong Sen Poh, Dinil Mon Divakaran, Hoon Wei Lim, Jianting Ning, and Achintya Desai. A survey of privacy-preserving techniques for encrypted traffic inspection over network middleboxes. *arXiv preprint arXiv:2101.04338*, 2021.

[108] Matthew Prince. Announcing 1.1.1.1: the fastest, privacy-first consumer DNS service. *Cloudflare Blog*, April 2018.

[109] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[110] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Security & Privacy*, 2014.

[111] Calyptix Security. Egress Filtering 101: What it is and how to do it, 2015.

[112] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *IACR CRYPTO*, 2020.

[113] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zksnarks. Cryptology ePrint Archive, Report 2020/1275, 2020. https://ia.cr/2020/1275.

[114] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, August 2012.

[115] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[116] Justine M. Sherry. *Middleboxes as a Cloud Service*. PhD thesis, University of California, Berkeley, 2016.

[117] Haya Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *Workshop on Privacy in the Electronic Society*, 2014.

[118] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted DNS–> privacy? a traffic analysis perspective. *arXiv preprint arXiv:1906.09682*, 2019.

[119] Anthony Spadafora. Apple devices will get encrypted DNS in iOS 14 and macOS 11. *Tech Radar*, July 2020.

[120] Justin Thaler. Proofs, arguments, and zero-knowledge. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html, 2020.

[121] tlslite-ng: TLS implementation in pure Python, 2021.

[122] Amar Toor. UK to block all online porn by default later this year. *The Verge*, 2013.

[123] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, 2018.

[124] Marshall Vale. Google Public DNS over HTTPS (DoH) supports RFC 8484 standard. *Google Security Blog*, June 2019.

[125] Riad Wahby. Practical proof systems: implementations, applications, and next steps. https://www.youtube.com/watch?v=roRPgh4bQMI, https://www.pepper-project.org/simons-vc-survey.pdf, September 2019.

[126] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *ISOC NDSS*, February 2015. https://ia.cr/2014/674.

[127] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Security & Privacy*, 2018. https://ia.cr/2017/1132.

[128] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2), February 2015.

[129] Juan Wang, Shirong Hao, Yi Li, Zhi Hong, Fei Yan, Bo Zhao, Jing Ma, and Huanguo Zhang. TVIDS: Trusted virtual IDS with SGX. *China Communications*, 16(10), 2019.

[130] Liang Wang, Gilad Asharov, Rafael Pass, Thomas Ristenpart, and abhi shelat. Blind certificate authorities. In *IEEE Security & Privacy*, 2019.

[131] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the Tor anonymity system. In *ACM CCS*, 2012.

[132] Florian Wilkens, Steffen Haas, Johanna Amann, and Mathias Fischer. Passive, transparent, and selective TLS decryption for network security monitoring. *arXiv preprint arXiv:2104.09828*, 2021.

[133] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J Alex Halderman. Telex: Anticensorship in the network infrastructure. In *Usenix Security*, 2011.

[134] Xingliang Yuan, Huayi Duan, and Cong Wang. Assuring string pattern matching in outsourced middleboxes. *IEEE/ACM Transactions on Networking*, 26(3), 2018.

[135] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In *ACM CCS*, 2020.

[136] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE Security & Privacy*, 2020.

[137] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *IEEE Security & Privacy*, 2018.

[138] Pete Zimmerman. Private communication, February 2021.

[139] ZoKrates: A toolbox for zkSNARKs on Ethereum. https://github.com/Zokrates/ZoKrates.

A Additional Preliminaries

In this appendix, we include some additional background on the cryptographic primitives discussed in the main body. We will leave some notational details, such as input and output spaces, implicit unless they are needed.

Hash functions, HMAC and HKDF. A hash function H takes as input a variable-length string x and outputs a fixed-size digest y. A pair of inputs $x \neq x'$ so that $H(x) = H(x')$ is called a collision. A common way to build hash functions for variable-length messages is by starting with a *compression* function f whose input length is fixed, then iteratively applying it to each fixed-length block of the variable-length input (starting with some initial fixed input value). In this work, the main compression function f of interest is SHA-256's, which has $\{0,1\}^{256} \times \{0,1\}^{512} \to \{0,1\}^{256}$. To build SHA-256 from f, the input must first be padded to a multiple of 512 bits. We do not need to worry about the details of this; we abstract it away as the procedure PadLB. For notational convenience we may sometimes write $H[f](V_1, V_2)$ to mean iterating f with initial chaining value V_1 over every 512-bit block of V_2, which we assume is block-aligned.

The Hash-based Message Authentication Code, or HMAC, is (for our purposes) a kind of two-input hash function parameterized by a hash H. The first input x is fixed-length and often called a "key"; the second y is variable-length and often called the "message". If x is less than the block size of H (e.g. 512 bits for SHA-256) it is right-padded with zeros until it is the size of a block. For distinct constant strings opad and ipad, The HMAC function is

$$\mathsf{HMAC}[H](x,y) = H(x \oplus \mathsf{opad} || H(x \oplus \mathsf{ipad} || y)) \ .$$

HMAC is the basis of HKDF, the key derivation function used in TLS 1.3. HKDF is composed of two procedures: one which *extracts* entropy from input strings, which we denote $\mathsf{hkdf.ext}$, and one which *expands* extracted entropy into key material, which we denote $\mathsf{hkdf.exp}$. In TLS1.3, $\mathsf{hkdf.ext}$ takes two arguments (x,y) and simply outputs $\mathsf{HMAC}(x,y)$.

$\mathsf{hkdf.exp}$ takes 4 inputs: source key K, a label string lbl, context cnxt, and output length ℓ. (In this work, it is always the case that ℓ is less than the output length of the underlying hash function.) Define the labeling function $\mathsf{hkdflabel}$ as

$$\mathsf{hkdflabel}(\mathsf{lbl}, \mathsf{cnxt}, \ell) = \langle \ell \rangle_{16} || \langle \ell^1_{in} \rangle_{16} || \text{"tls13 "} || \mathsf{lbl} || \langle \ell^2_{in} \rangle_{16} || \mathsf{cnxt}$$

where ℓ^1_{in} and ℓ^2_{in} are the lengths of "tls13 "||lbl and cnxt in bytes. We compute $\mathsf{hkdf.exp}$ by

$$\mathsf{hkdf.exp}(K, \mathsf{lbl}, \mathsf{cnxt}, \ell) = \mathsf{HMAC}(K, \mathsf{hkdflabel}(\mathsf{lbl}, \mathsf{cnxt}, \ell) || \text{ 0x01})$$

If the fourth argument is left unspecified, it is assumed to be equal to the hash function's output length. As a convenience function below, we also define the function $\mathsf{DeriveTK}(s)$ to derive traffic keys and IVs from some input secret s by outputting the pair

$$(\mathsf{hkdf.exp}(s, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16), \mathsf{hkdf.exp}(s, \mathsf{lbl}_{\text{iv}}, h_\varepsilon, 12)) \ .$$

All $\mathsf{hkdf.exp}$ calls in TLS 1.3 take a *label* input describing the context of the derived value — for example, the label "c ap traffic" for *CATS*. To simplify the presentation below, we will abstract away the concrete strings, and just number the labels (e.g. lbl_6) following the numbering convention of [38] so that distinct labels have distinct numbers. See [109] for these labels' values.

GCM-CTR encryption. For a block cipher E with block size n, key K nonce N of $n-32$ bits and message M define the GCM-Counter encryption scheme GCM-CTR$[E](K,N,M,\ell_1,\ell_2)$ to be CTR-mode encryption of the substring of M between bytes ℓ_1 and ℓ_2, inclusive, but with the ith block of pad computed as $E(K,N||\langle 2+i\rangle_{32})$. We will usually omit the last two arguments, writing GCM-CTR$[E](K,N,M)$ to mean GCM-CTR$[E](K,N,M,1,\mathsf{len}(M))$. This CTR mode variant is how messages are encrypted in GCM, the AEAD scheme we will assume is used for TLS 1.3's record layer.

B Opening Early Data

In this appendix, we describe our channel opening circuit for data sent with the 0-RTT resumption feature of TLS 1.3. We first briefly explain how resumption handshakes work in TLS 1.3.

Resumption in TLS 1.3. As in previous versions, TLS 1.3 has special handling for handshakes that resume a previously-established session between a client and server. Such handshakes are called "resumption" handshakes, and are usually more efficient than non-resumption handshakes because they rely on the client and server having established a shared secret value psk for resumption during the previous session.

Resumption handshakes are different in two concrete ways that are important below: first, CH includes an extension that identifies psk. This extension ends with an HMAC of CH, keyed with a psk-derived key. This HMAC is called the "PSK binder". Second, in a resumption handshake the client can send encrypted data immediately after the CH, without waiting for the server's response. This data is usually called "early data", and resumption handshakes that include it are usually called "0-RTT".

The EarlyChanOpen protocol. We describe this channel opening circuit in Figure 7. We follow the pseudocode convention established in Section 5 and simplify HKDF inputs. The circuit takes as public input CH and the client's early data ciphertext C_{ED}, and as private input the PSK psk, the key K, and the plaintext M. The main idea is to use the PSK binder as a commitment to psk, analogous to the Finished value SF in ShortcutChanOpen.

The key consistency procedure CATKeyVerif-0RTT computes the relevant parts of the resumption key schedule, such as the binder key fk_B. Note that the two context hashes h_1 and h_5 are computed on different prefixes of CH: CH_{NE} is everything except for the final extension which identifies the PSK and contains the binder value; PSKExt is the beginning of the PSK extension.

This method of channel opening leads to the smallest circuit, but has some drawbacks: because it is only for resumption handshakes, it requires the client and server to have previously set up a session with a normal handshake. Another drawback is that it requires the server to accept early data. Early data support is optional in TLS 1.3 because early ciphertexts are susceptible to replay attacks. Nevertheless, some large web hosting providers (e.g. Cloudflare) support it in cases where replay attacks are not a threat, such as GET requests in HTTP.

EarlyChanOpen$(CH, C_{\mathrm{ED}}\,;\mathsf{psk},K,M)$:
$M' \leftarrow \mathsf{Dec}(K, C_{\mathrm{ED}})$
$b \leftarrow$ CATKeyVerif-0RTT(CH,psk,K)
Return $b \wedge (M' = M)$

CATKeyVerif-0RTT(CH,psk,K):
$ES \leftarrow \mathsf{hkdf.ext}(\mathsf{psk})$
$BK \leftarrow \mathsf{hkdf.exp}(ES,\text{"resbinder"})$
$fk_B \leftarrow \mathsf{hkdf.exp}(BK,\text{"fkey"})$
$CH_{NE}, \mathsf{PSKExt}, BD \leftarrow CH$
$h_5 \leftarrow H(CH_{NE}||\mathsf{PSKExt})$
$BD' \leftarrow \mathsf{HMAC}(fk_B, h_5)$
$h_1 \leftarrow H(CH_{NE})$
$ETS \leftarrow \mathsf{hkdf.exp}(ES, h_1)$
$EATK \leftarrow \mathsf{hkdf.exp}(ETS,\text{"tkey"})$
Return $BD' = BD \wedge EATK = K$

Figure 7: Channel opening circuit for early data sent in a 0-RTT resumption handshake.

C Security Analyses

In this appendix, we analyze the security of our channel opening circuits from Section 5. We begin with the baseline BaselineChanOpen. The security experiment, $\mathsf{TLS\text{-}BIND}_{BL}$ is depicted in Figure 8. It is (implicitly) parameterized by a group $\mathbb{G}$ of order p with generator g, a hash function H underlying both HMAC and HKDF, and a block cipher E. It also uses a value dES; this is a fixed 256-bit value resulting from running TLS 1.3's PSK key schedule with the all-zeros PSK.

Note that we make a few simplifications to omit details which are unimportant for us: we abstract away the format of TLS1.3 protocol messages with the GetDH and MakeServerHello functions; the former extracts the DH value from a Hello message, and the latter formats a SH per the spec, given a nonce and DH value. We also use a fixed string SC of length s in place of the server's Certificate and CertificateVerify messages. The semantics of these messages do not matter to us — the only thing about them that is relevant to our analysis is that their contents are out of the client's control.

The security experiment gives the adversarial client $\mathcal{A}$ oracle access to an oracle O that takes as input an adversarially-chosen CH and completes the server's part of the TLS 1.3 handshake — choosing a Diffie-Hellman secret b and nonce

$\underline{\mathsf{TLS\text{-}BIND}^{\mathcal{A}}_{BL}}$:

$a, CATK', CATN', CH, SH, C_{EE} \xleftarrow{\$} \mathcal{A}^{O}$
If $T[(CH, SH, C_{EE})] = \perp$ then Return 0
$(CATK, CATN) \leftarrow T[(CH, SH, C_{EE})]$
$b \leftarrow \mathrm{CATKeyVerif}(CH, SH, C_{EE}; a, CATK', CATN')$
Return $b \wedge (CATK, CATN) \neq (CATK', CATN')$

$\underline{\mathrm{CATKeyGen}(HS, CH, SH, C_{EE})}$:

$h_2 \leftarrow H(CH||SH)$
$SHTS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_5, h_2)$
$(SHTK, SHTN) \leftarrow \mathsf{DeriveTK}(SHTS)$
$EEP \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE})$
$h_3 \leftarrow H(CH||SH||EEP)$
$dHS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf.ext}(dHS, 0)$
$CATS \leftarrow \mathsf{hkdf.exp}(MS, \mathsf{lbl}_7, h_3)$
Return $\mathsf{DeriveTK}(CATS)$

$\underline{\mathrm{CATKeyVerif}(CH, SH, C_{EE}; a, CATK, CATN)}$:

$A \leftarrow \mathsf{GetDH}(CH)$
$B \leftarrow \mathsf{GetDH}(SH)$
$HS \leftarrow \mathsf{hkdf.ext}(dES, B^a)$
$CATK', CATN' \leftarrow \mathrm{CATKeyGen}(HS, CH, SH, C_{EE})$
Return $g^a = A \wedge CATK = CATK' \wedge CATN = CATN'$

$\underline{O(CH)}$:

$A \leftarrow \mathsf{GetDH}(CH)$
$N_S \xleftarrow{\$} \{0,1\}^{256}$
$b \xleftarrow{\$} \{1, \ldots, p\}; B \leftarrow g^b$
$SH \leftarrow \mathrm{MakeServerHello}(N_S, B)$
$HS \leftarrow \mathsf{hkdf.ext}(dES, A^b)$
$h_2 \leftarrow H(CH||SH)$
$SHTS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_5, h_2)$
$(SHTK, SHTN) \leftarrow \mathsf{DeriveTK}(SHTS)$
$fk_S \leftarrow \mathsf{hkdf.exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$
$h_7 \leftarrow H(CH||SH||SC)$
$SF \leftarrow \mathsf{HMAC}(fk_S, h_7)$
$EEP \leftarrow SC||SF$
$C_{EE} \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, EEP)$
$h_3 \leftarrow H(CH||SH||EEP)$
$dHS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf.ext}(dHS, 0)$
$CATS \leftarrow \mathsf{hkdf.exp}(MS, \mathsf{lbl}_7, h_3)$
$T[(CH, SH, C_{EE})] \leftarrow \mathsf{DeriveTK}(CATS)$
Return (SH, C_{EE})

Figure 8: Security experiment for binding security of the BaselineChanOpen protocol in Section 5. Notation is explained there and in the prose in this appendix. The string SC is an arbitrary fixed string.

N_S to put in the SH, then computing the key schedule up to the client's application keys. The oracle stores (CH, SH, C_{EE}) in a table, along with the client application keys that are output by the key schedule for this handshake. The oracle outputs the handshake messages SH and C_{EE}. (Note that we assume the client can compute the application keys itself using the key schedule, and do not return them from O.)

After querying O some number of times, $\mathcal{A}$ must output a secret DH value a along with a key $CATK'$ and IV $CATN'$ and one of the handshake transcripts stored in the table T. The adversary wins if CATKeyVerif returns 1, but the key/iv pair output by the adversary are *different* than the one stored in the table for this handshake. Now we can state and prove the security theorem for BaselineChanOpen.

Theorem 1 *Let $\mathcal{A}$ be an adversary in game $\mathsf{TLS\text{-}BIND}_{BL}$, as described above, making any number of queries to its oracle O. Then*

$$\Pr\left[\mathsf{TLS\text{-}BIND}^{\mathcal{A}}_{BL} \Rightarrow 1\right] = 0\,.$$

Proof: The proof of this theorem is extremely simple. Since the adversary's CH contains g^a, a perfectly binding commitment to its DH secret which is checked inside CATKeyVerif, the adversary cannot cause CATKeyVerif to output 1 except by using the DH secret a which was committed to in the oracle call which inserted the adversary's handshake (CH, SH, C_{EE}) into the table. Since fixing a and the handshake fixes the client keys, the adversary cannot win.

Note that in our security experiment, the client has no control over the randomness the server uses to generate its DH value and nonce. In fact, Theorem 1 holds even if the client can control the server's randomness, since the pair of DH values (B, A) is a binding commitment to the shared DH secret.

C.1 Analyzing ShortcutChanOpen

Next we analyze the security of the optimized ShortcutChanOpen circuit discussed in Section 5. Note that this appendix presents a slightly different variant of the protocol: CATKeyVerif-SC gets C_{EE} as input instead of just a suffix; importantly, it also does not take vsuff as an input, instead re-computing this from C_{EE}. The slight simplification in Section 5 makes the protocol easier to explain; ultimately, our analysis will show the difference between the two is unimportant. We define a binding security

experiment $\mathsf{TLS\text{-}BIND}^{\mathcal{A}}_{SC}$ in Figure 9. It is mostly similar to the experiment $\mathsf{TLS\text{-}BIND}_{BL}$, with a few important differences.

Theorem 2 *Let $\mathcal{A}$ be an adversary in game $\mathsf{TLS\text{-}BIND}_{SC}$ as described in Figure 9, where the compression function $f\colon \{0,1\}^n \times \{0,1\}^{2n} \to \{0,1\}^n$ is an ideal compression function and the block cipher $E\colon \{0,1\}^{n/2} \times \{0,1\}^{n/2} \to \{0,1\}^{n/2}$ is an ideal cipher. Let q_O be the number of queries $\mathcal{A}$ makes to its oracle O, q_f be the total number of queries made to f by $\mathcal{A}$ and in O, and q_E the total number of queries to E made by $\mathcal{A}$ and in O*

$$\Pr\left[\mathsf{TLS\text{-}BIND}^{\mathcal{A}}_{SC} \Rightarrow 1\right] = \frac{q_f^2}{2^n} + \frac{q_E^2}{2^{n/2}} + \frac{q_f q_E}{2^n} .$$

Proof: We use a game-hopping argument to bound the adversary's success probability. We assume wlog that $\mathcal{A}$ queries the f and E oracles on all inputs that will be used in CATKeyVerif-SC to verify its outputs. The first game, G_0, is depicted in Figure 9. It is a rewriting of $\mathsf{TLS\text{-}BIND}_{SC}$, with some syntactic changes: we have surfaced the oracle f as a parameter everywhere it is used in H, hkdf, or HMAC. We also changed how the hashes h_7 and h_3 are computed; rather than computing them with two calls to H, we instead explicitly compute the iteration of f on their common 64-byte block prefix, resulting in a chaining variable LCCV. This is the last chaining value in common between $H[f](CH||SH||SC)$ and $H[f](CH||SH||SC||SF)$. With this, we then compute both h_7 and h_3 by padding and hashing the respective suffixes of their inputs. Finally, we introduce another table, W (for "witnesses"), which keeps track of the HS and chaining value LCCV derived in O.

The second game, G_1, is the same as G_0 except collisions in the output of f (for distinct inputs) are disallowed. By a standard argument (q.v. [7]) the difference in the adversary's probability of success in G_0 and G_1 is

$$\left|\Pr\left[G_0^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[G_1^{\mathcal{A}} \Rightarrow 1\right]\right| \leq \frac{q_f^2}{2^n} .$$

Game G_2 is the same as G_1, except all wins where $HS_{\text{out}} = HS_O$ are disallowed. The only winning outputs possible in G_1 that are impossible in G_2 are those where $HS_{\text{out}} = HS_O$ but $\mathsf{LCCV}_{\text{out}} \neq \mathsf{LCCV}_O$. However, in G_1 these wins are already impossible: since collisions in f are disallowed, $\mathsf{LCCV}_{\text{out}} \neq \mathsf{LCCV}_O$ implies the h_7 obtained when running CATKeyVerif-SC on the adversary's output must be different than the one obtained in the oracle call that inserted (CH, SH, C_{EE}) into the table. This implies the value SF' must be different than SF as well. (Note that since SF is computed by decrypting C_{EE}, its value is the same in CATKeyVerif-SC and the corresponding O call.) Thus,

$$\left|\Pr\left[G_1^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[G_2^{\mathcal{A}} \Rightarrow 1\right]\right| = 0 .$$

In G_2, the adversary cannot win unless $HS_{\text{out}} \neq HS_O$. To finish the proof, we only need to upper-bound the probability of such wins. First, we transition to a game G_3, which is identical to G_2 except the ideal cipher E is replaced with a (keyed) random function on the same domain and range. We apply a standard bound on the difference in probability between a random permutation and a random function to get that

$$\left|\Pr\left[G_2^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[G_3^{\mathcal{A}} \Rightarrow 1\right]\right| = \frac{q_E^2}{2^{n/2}} .$$

In G_3, $HS_{\text{out}} \neq HS_O$ and collisions in the output of f are disallowed. These two facts imply that in CATKeyVerif-SC, the handshake traffic key and nonce $SHTK$ and $SHTN$ used to decrypt the last 32 bytes of C_{EE} to SF *must* be different than the ones that were used in the oracle call that inserted (CH, SH, C_{EE}) into the table. This is because the inputs to the first $\mathsf{hkdf}[f].\mathsf{exp}$ call must be different in CATKeyVerif-SC and that oracle call; since f's output cannot collide for distinct inputs, the handshake traffic keys must be different.

In G_3, each call to E on distinct inputs outputs a uniformly random string. This implies the pad derived in GCM-CTR$[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31, \ell_{EE})$ is also uniformly random, and the SF output is as well. For the adversary to win, this random SF value must have been previously output by some call to f — since the derived SF' is output by f, the check $SF = SF'$ can't be true unless SF was output by f. Since there are at most q_f such outputs, the probability of any SF hitting one is $q_f/2^n$. By a union bound over the q_E ideal cipher queries made by $\mathcal{A}$, we get that

$$\Pr\left[G_3^{\mathcal{A}} \Rightarrow 1\right] \leq \frac{q_E q_f}{2^n} .$$

Applying the triangle equality and summing the right-hand sides of each bound completes the proof. ∎

This analysis of ShortcutChanOpen in Section 5 favored simplicity over tightness; as a result, there are several ways it could be improved. One is the step which swaps block cipher outputs for random strings. This incurs a birthday-bound loss in the output size of the block cipher and preventing us from proving more than 64 bits of security when the block cipher is instantiated with AES. This level of security is generally considered too low for practical applications; however, we strongly suspect this loss in tightness is not inherent, and a more complex analysis could remove this term from the bound. Using the ChaCha20 function for encryption instead of AES would allow this term to be easily removed from the bound, provided the ChaCha20 function can be suitably modelled as an ideal random function.

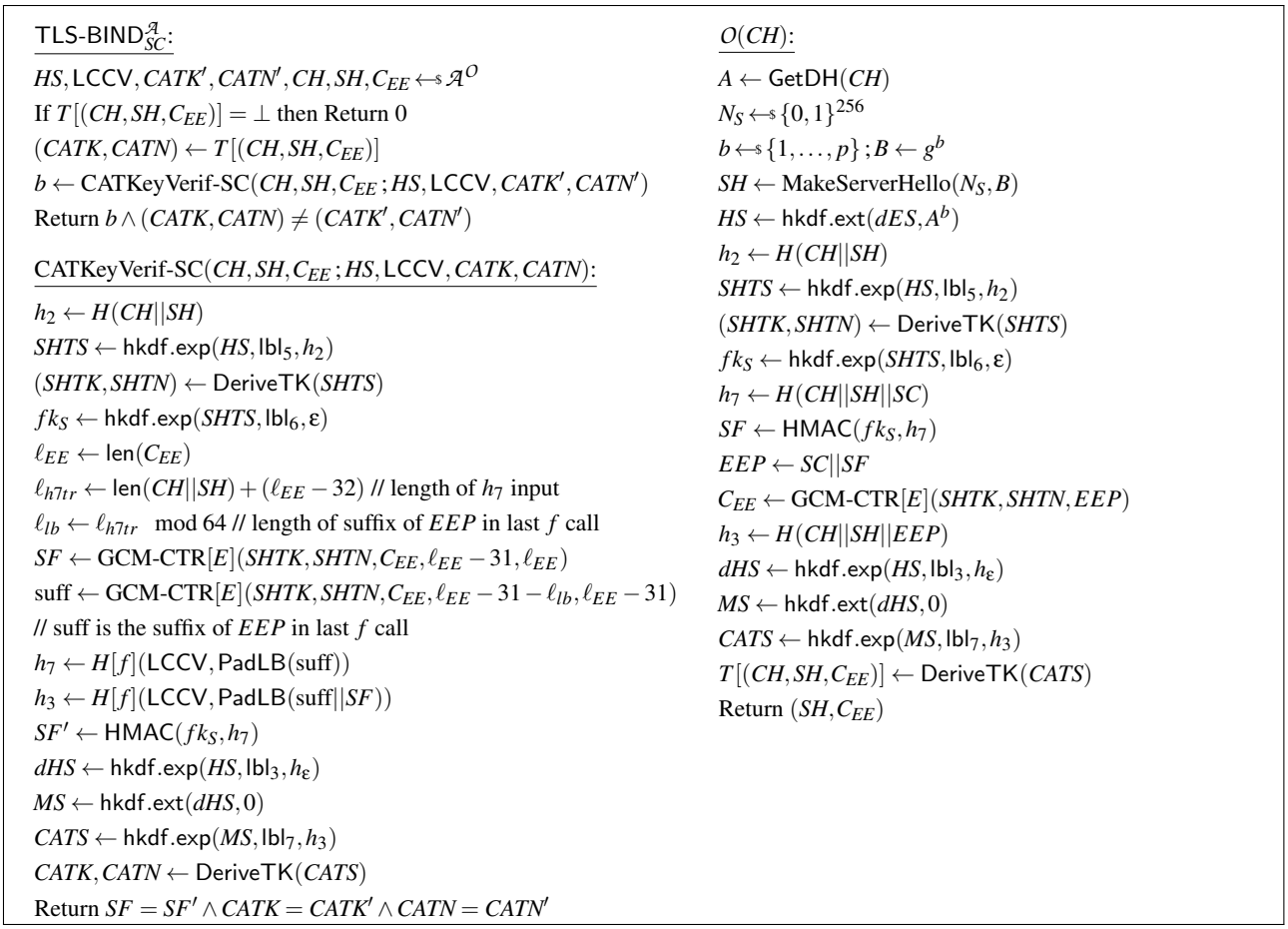

Figure 9: Security experiment for binding security of ShortcutChanOpen in Section 5. The string SC is an arbitrary fixed string.

Also significant is that our proof ignores one constraint on the adversary's winning probability. Namely, in the final upper bound on the win probability in G_3, we said that SF can hit *any* f output, but this is not exactly true: the value SF' depends both on the finished key fk_S and h_7, and h_7 depends on the string suff decrypted from C_{EE}. We suspect the bound could be tightened if this constraint were used in the analysis. An alternate interpretation of our analysis not needing the constraint implies that the suffix suff could be given as witness, instead of recomputed by decrypting C_{EE} in the circuit. Concretely, this would save roughly 60,000 multiplication gates by avoiding four AES computations. We leave this and other optimizations to future work.

Finally, we chose to use an idealized model of the underlying symmetric primitives in this analysis. This lessens its complexity somewhat, but limits its direct relevance to practical uses of our protocol, which would instantiate the ideal primitives using concrete algorithms like AES and SHA-256. A standard-model analysis would be somewhat more informative, but also more complex; we therefore leave it to future work.

C.2 Other Results and Discussion

In this appendix, we analyzed the security of our BaselineChanOpen and ShortcutChanOpen channel opening circuits. We omit analyses of our EarlyChanOpen circuit for early data, and our amortized AMChanOpen circuit. The latter is nearly trivial — collision-resistance of the hash prevents lying about the key. The former is more complex, but is substantially similar to our ShortcutChanOpen analysis above. (In fact, the analysis is strictly easier, since the PSK binder is sent in plaintext instead of encrypted, as the SF value is in the full handshake.)

Our security experiment in this section models quite closely the practical threat of a malicious client equivocating the session key: we give the adversary oracle access to, essentially, an honest server that completes as many handshakes as the adversary wants. However, it would be useful to prove security in a stronger model if possible: for example, one where the adversary can influence the server implementation, or the random values it chooses in the SH. Since these adversaries are outside our threat model, we leave this to future work.

$\underline{G_0^{\mathcal{A}}}$:

$HS_{\text{out}}, \mathsf{LCCV}_{\text{out}}, CATK_{\text{out}}, CATN_{\text{out}}, CH, SH, C_{EE} \leftarrow\!\!\$\ \mathcal{A}^{O,f,E}$
If $T[(CH,SH,C_{EE})] = \perp$ then Return 0
$(CATK_O, CATN_O) \leftarrow T[(CH,SH,C_{EE})]$
$HS_O, \mathsf{LCCV}_O \leftarrow W[(CH,SH,C_{EE})]$
$b \leftarrow \text{CATKeyVerif-SC}(CH,SH,C_{EE}; HS_{\text{out}}, \mathsf{LCCV}_{\text{out}}, CATK_{\text{out}}, CATN_{\text{out}})$
Return $b \wedge (CATK_O, CATN_O) \neq (CATK_{\text{out}}, CATN_{\text{out}})$

$\underline{\text{CATKeyVerif-SC}(CH,SH,C_{EE}; HS, \mathsf{LCCV}, CATK, CATN)}$:

$h_2 \leftarrow H[f](CH||SH)$
$SHTS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_5, h_2)$
$SHTK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$
$SHTN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{iv}}, h_\varepsilon, 12)$
$fk_S \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$
$\ell_{EE} \leftarrow \mathsf{len}(C_{EE})$
$\ell_{h7tr} \leftarrow \mathsf{len}(CH||SH) + (\ell_{EE} - 32)$
$\ell_{lb} \leftarrow \ell_{h7tr} \mod 64$
$SF \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31, \ell_{EE})$
$\text{suff} \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31 - \ell_{lb}, \ell_{EE} - 31)$
$h_7 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}))$
$h_3 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}||SF))$
$SF' \leftarrow \mathsf{HMAC}[f](fk_S, h_7)$
$dHS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf}[f].\mathsf{ext}(dHS, 0)$
$CATS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(MS, \mathsf{lbl}_7, h_3)$
$CATK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$
$CATN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{iv}}, h_\varepsilon, 12)$
Return $SF = SF' \wedge CATK = CATK' \wedge CATN = CATN'$

$\underline{O(CH)}$:

$A \leftarrow \mathsf{GetDH}(CH)$
$N_S \leftarrow\!\!\$\ \{0,1\}^{256}$
$b \leftarrow\!\!\$\ \{1,\ldots,p\}; B \leftarrow g^b$
$SH \leftarrow \text{MakeServerHello}(N_S, B)$
$HS \leftarrow \mathsf{hkdf}[f].\mathsf{ext}(dES, A^b)$
$h_2 \leftarrow H[f](CH||SH)$
$SHTS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_5, h_2)$
$SHTK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$
$SHTN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{iv}}, h_\varepsilon, 12)$
$fk_S \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$
$\mathsf{tr}_{h7} \leftarrow CH||SH||SC$
$\ell_{lb} \leftarrow \mathsf{len}(\mathsf{tr}_{h7}) \mod 64$
$fbs \leftarrow (\mathsf{len}(\mathsf{tr}_{h7}) - \ell_{lb})/64$ // #full 64-byte blocks in tr_{h7}
$\text{suff} \leftarrow \mathsf{tr}_{h7}[fbs * 64 : \mathsf{len}(\mathsf{tr}_{h7})]\ d \leftarrow \mathsf{ICV}_{\text{sha}}$
For $i \leftarrow 1$ to fbs:
  $d \leftarrow f(d, \mathsf{tr}_{h7}[(i-1) * 64 : i * 64])$
$\mathsf{LCCV} \leftarrow d$
$h_7 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}))$
$SF \leftarrow \mathsf{HMAC}[f](fk_S, h_7)$
$h_3 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}||SF))$
$EEP \leftarrow SC||SF$
$C_{EE} \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, EEP)$
$dHS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf}[f].\mathsf{ext}(dHS, 0)$
$CATS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(MS, \mathsf{lbl}_7, h_3)$
$CATK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$
$CATN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{iv}}, h_\varepsilon, 12)$
$T[(CH,SH,C_{EE})] \leftarrow (CATK, CATN)$
$W[(CH,SH,C_{EE})] \leftarrow (HS, \mathsf{LCCV})$
Return (SH, C_{EE})

Figure 10: Game G_0 of the proof of Theorem 2.