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 optimized zero-knowledge proofs for TLS 1.3 session keys.

We apply the ZKMB paradigm to several case studies. Experimental results suggest that in certain settings, performance is in striking distance of practicality; an example is a middlebox that filters domain queries (each query requiring a separate proof) when the client has a long-lived TLS connection with a DNS resolver. In such configurations, the middlebox's overhead is 2–5 ms of running time per proof, and client latency to create a proof is several seconds. On the other hand, clients may have to store hundreds of MBs depending on the underlying zero-knowledge proof machinery, and for some applications, latency is tens of seconds.

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 [15, 19, 75]. Selling data derived from DNS queries to advertisers has also been legal in the US since 2017 [75]. Another alarming practice is targeted government surveillance [15, 62] using DNS.

In response, several protocols have emerged for encrypting DNS queries to remote resolvers outside the control of the local network administrator [70, 73, 78]. 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 [24] 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, 72] 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, 63] and commercial U.K. ISPs [126]; 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 [66] and EndBox [58]) rely on trusted execution environments [37, 42, 59, 65, 84, 110, 127, 135] or (like BlindBox [119] and mcTLS [102]) require application-specific modifications to standard cryptographic protocols [47, 77, 86–88, 90, 94, 104, 105, 111, 140, 143].

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 assume changes to middleboxes and the clients behind them, but these can happen incrementally. Compatibility also means that the network is open to all, not just clients with 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

*Now at the University of Michigan.

[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.

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 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; §7.5). 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. 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 scheme is binding (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 [38,64,93]. 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 special 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, bind 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. 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. The enabling insight is that the server's authentication messages contain a hash of the results of these operations, and for security, it suffices for the circuit to re-run only the hash's last block. We analyze the security of the baseline and the shortcut (Appendix C). A feature of our approach is that it amortizes over an entire TLS connection, and can be extended to support early data sent in a 0-RTT resumption (Appendix B.3).

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 experimental evaluation of ZKMBs (§8) uses xJsnark [81] for compiling code to circuits and the Groth16 [61] proof protocol. As a consequence of Groth16, proofs are 128 bytes and 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 3–8 seconds to generate, depending on the policy. Also, the client must store hundreds of MB of data to participate in the proof protocol.

The foregoing implies that ZKMBs are close to practical in specialized settings. Notice that a long-lived TLS connection between client and server (to amortize the initiation cost), with proofs needed at most every few seconds, exactly fits blocklisting domains for encrypted DNS (§7). Of course, for many middlebox functionalities, our "v1 ZKMBs" are decidedly not practical. One avenue for future work (§10) is to interrogate tradeoffs (§9) among setup costs, trust, verifier performance, and prover performance, to identify the best ZK machinery for the context. For example, if there are not many clients behind a given middlebox, then we should use ZK proof protocols with far lower proving time, at the expense of verification time [1, 11, 44, 92, 116, 138, 139, 142, 145]. Another area of future work is expanding the scope of configurations (for example, to support off-path middleboxes) and functionalities (for example, to support scanning-style middleboxes).

Despite the limitations of our work, we believe that ZKMBs are worthy of continued research: the approach 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 *middlebox*. Middleboxes have tremendous variety in function, configuration, and architecture [101, 120]. In this work, we focus on the simplest case: a single, stateless middlebox on the local network, and hence on the path between the client and any server; Figure 1 depicts the basic arrangement. We expect the ZKMB paradigm to generalize (§9–§10).

The client's goal is to communicate with the server using an end-to-end encrypted channel. The middlebox's goal 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, checking for malware signatures).

The ZKMB protocol should be incrementally deployable: clients without support for ZKMBs should be able to use the network. However, as happens today, the middlebox will block their encrypted traffic, forcing a downgrade to unencrypted protocols. The ZKMB protocol thus represents an opportunistic privacy upgrade.

In the basic ZKMB protocol flow, the client downloads a network policy specification from the middlebox upon joining the network. (Looking ahead, in Section 7.5 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 sent over a certain protocol. Then, for each encrypted channel to a remote server that is within scope, the client submits a *proof* that the contents of the channel comply with the network policy; the middlebox verifies it. Note that the middlebox has to 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.

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. We also assume that the middlebox and server do not collude.

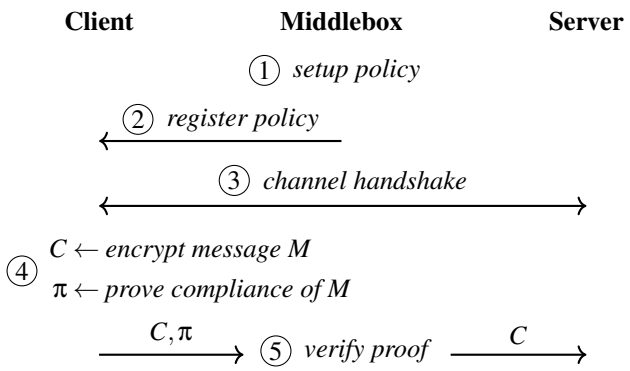

Figure 1: Structure of our ZKMB protocol. The client convinces the middlebox that the message M is policy-compliant without revealing it. The handshake and encryption algorithms are part of the underlying secure channel (such as TLS). Transcripts and secrets generated in the handshake may be inputs to the proving and verification steps.

The ZKMB protocol should ensure that an adversarial middlebox can't learn anything from ZKMB proofs beyond the fact that the encapsulated plaintext is policy-compliant. Of course, if an underlying protocol leaks information through timing or metadata—as is the case with encrypted DNS [22, 121, 122]—ZKMBs necessarily inherit that leak.

Another objective is compatibility with existing servers; a non-objective is to work with unmodified clients and middleboxes. We make these choices because modern browsers are empirically easier to update than web servers. For example, as of January 2022 only 51% of TLS servers supported TLS 1.3, which was finalized nearly four years prior [85], whereas 50% of Chrome users were already running Chrome 97, which was less than one month old [130]. That said, server-side changes would enable additional use cases and the ZKMB framework would still be motivated.

ZKMBs should ensure that any 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 [36], hidden proxies [141], "inner encryption," or steganography [137]. Detecting and blocking such circumvention is an arms race (see, for example, the progression of obfuscated Tor transport-layer encodings [45]).

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. 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? We believe 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). In fact, we hope that ZKMBs will ultimately demonstrate that "policy enforcement requires plaintext" does not follow, removing an alleged rationale for surveillance.

Performance considerations. We evaluate ZKMB performance 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 when joining the network. Connection latency is the sum of computation time for clients and middleboxes, plus time to transfer proofs; the primary performance goal is to minimize this quantity.

Some proof protocols have public parameters (an SRS; §3). Who generates the SRS? As discussed in Section 10, options include the middlebox, the browser vendor, or else using a back-end proof protocol that has no SRS.

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) $\mathbf{C}$, a *verifier* $\mathcal{V}$, and a *prover* $\mathcal{P}$. We formulate the computation as a function $\mathbf{C}(X;W)$ 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 $\mathbf{C}$. (We call protocols consisting of a single $\mathcal{P} \rightarrow \mathcal{V}$ message "non-interactive".) "Valid" means that there exists a w such that $y = \mathbf{C}(x;w)$, 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 secret auxiliary information needed to decrypt, and $\mathbf{C}$ embeds decryption logic and policy logic. Semantically, the protocol convinces $\mathcal{V}$ that the putative yes-or-no is consistent with the underlying plaintext that corresponds to the encrypted data.

For many modern ZK constructions, $\mathbf{C}$ is expressed as a generalization of *arithmetic circuits*, called R1CS [54]. For convenience, we call this formalism "circuits". Circuits bring challenges [17, 32, 106, 108, 118, 132]. 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, 18, 81, 107, 132].

Our work uses the QAP-based [54] proof protocol Groth16 [61], which is a non-interactive zero knowledge proof, or NIZK [13, 48] (Groth16 is a kind of NIZK called a zkSNARK [12, 54].) For our purposes, a NIZK is a tuple of (possibly probabilistic) algorithms (ZKSetup, ZKProve, ZKVerify) that depend on $\mathbf{C}$:

- $\sigma \leftarrow \mathsf{ZKSetup}(\mathbf{C})$. 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 $\mathbf{C}$ and reused over different (x,y,w).
- $\pi \leftarrow \mathsf{ZKProve}(\mathbf{C}, \sigma, (x,y), w)$. π is referred to as the proof.
- $\texttt{accept/reject} \leftarrow \mathsf{ZKVerify}(\mathbf{C}, \sigma, (x,y), \pi)$.

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 $y = \mathbf{C}(x;w)$) or $\mathcal{P}$ does not know a w for which $y = \mathbf{C}(x;w)$, 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: $\mathsf{ZKMB[SChan]} = \langle \mathsf{Setup}, \mathsf{Registration}, \mathsf{Prove}, \mathsf{Verify} \rangle$ 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 (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 *PMD*. The metadata in *PMD* 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.

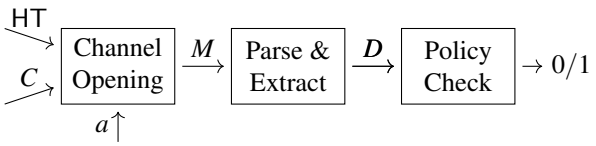

Figure 2: The parts of a ZKMB proof. HT = handshake transcript, C = ciphertext, a = client's handshake secret, M = plaintext message, D = policy-relevant substring.

The Registration procedure (② in Figure 1) is an interactive algorithm run by a client and the middlebox to give the client the policy metadata *PMD* upon joining the network. We expect this to piggy-back on network-layer bootstrap protocols, e.g. using DHCP [40] extensions [41].

The client runs SChan.Handshake to open a secure connection to a server (③ in Figure 1). When the client wants to send a message M to the server, the client first checks that M is policy-compliant; if it isn't then no proof can be generated so the client will decline to send M to the server and instead 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 ④ in Figure 1) to prove that M is allowed by the policy. Prove takes as public input the handshake transcript HT, policy metadata *PMD*, and ciphertext C, and as private input the secret randomness a generated by the client during the handshake. In TLS 1.3 for example, a is the client's Diffie-Hellman secret.

Prove produces a proof π (using the underlying ZKProve algorithm) which the client sends to the middlebox along with the ciphertext C. After receiving π from the client, the middlebox runs the Verify procedure (⑤ in Figure 1). This procedure takes as input C and π, along with the transcript of SChan.Handshake, and invokes the underlying ZKVerify algorithm to output $0/1$. The result determines whether the middlebox forwards the client's encrypted traffic C to the server, which decrypts using SChan.Dec.

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 ④–⑤ for the duration of the connection.

Framework for expressing ZK statements. The circuit that the ZK proof works over (§3) comprises three parts, as depicted in Figure 2:

1. *channel opening*: Given the handshake secret a and transcript HT, derive (hidden) key K and use it to decrypt ciphertext C and obtain (hidden) message M.
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.

$\mathsf{SimpleTLS.S2}(b, A, \mathsf{cert})$:	$\mathsf{SimpleTLS.C3}(a, (B, C_{\mathsf{cert}} \Vert SF))$:
1: $B \leftarrow g^b; SS \leftarrow A^b$	**1:** $SS \leftarrow B^a$
2: $FK \leftarrow \mathsf{KDF}(SS, 1)$	**2:** $FK \leftarrow \mathsf{KDF}(SS, 1)$
3: $HK \leftarrow \mathsf{KDF}(SS, 2)$	**3:** $HK \leftarrow \mathsf{KDF}(SS, 2)$
4: $h_1 \leftarrow H(A \Vert B \Vert \mathsf{cert})$	**4:** $\mathsf{cert} \leftarrow \mathsf{Dec}(HK, C_{\mathsf{cert}})$
5: $SF \leftarrow \mathsf{HMAC}(FK, h_1)$	**5:** $h_1 \leftarrow H(g^a \Vert B \Vert \mathsf{cert})$
6: $kh \leftarrow H(h_1 \Vert SF)$	**6:** $SF' \leftarrow \mathsf{HMAC}(FK, h_1)$
7: $K \leftarrow \mathsf{KDF}(SS, kh)$	**7:** $kh \leftarrow H(h_1 \Vert SF')$
8: $C_{\mathsf{cert}} \leftarrow \mathsf{Enc}(HK, \mathsf{cert})$	**8:** $K \leftarrow \mathsf{KDF}(SS, kh)$
9: Send $B, (C_{\mathsf{cert}} \Vert SF)$ to client	**9:** If $SF \neq SF'$ then Return $\perp$
10: Return K	**10:** Return K

Figure 3: The latter two steps of SimpleTLS.Handshake. The first step (SimpleTLS.C1) and all notation are defined in Section 5.

3. *policy check*: Check that D satisfies the policy in the metadata *PMD*.

Terminology: circuit vs. sub-circuit 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.

5 Channel Opening for Simplified TLS 1.3

This section discusses the core technical challenges of designing channel opening sub-circuits for modern secure channels. For ease of exposition, we present a simplified variant of TLS 1.3, which we call SimpleTLS. Appendix B presents our channel opening sub-circuits for TLS 1.3 in detail.

SimpleTLS begins with an interactive handshake SimpleTLS.Handshake that establishes a session key K. After the handshake, the client and server use an authenticated encryption scheme like AES-GCM or ChaCha20/Poly1305 to encrypt (Enc) and decrypt (Dec) messages with K.

SimpleTLS.Handshake comprises three algorithms $(\mathsf{C1}, \mathsf{S2}, \mathsf{C3})$, corresponding to TLS as (C1 = client hello, S2 = server reply, C3 = client finished). To initiate a channel, the client first runs SimpleTLS.C1, which samples $a \leftarrow\!\!\$\, \mathbb{Z}_p$, computes $A \leftarrow g^a$, and sends A to the server.

The server replies by generating its private exponent $b \leftarrow\!\!\$\, \mathbb{Z}_p$ and running SimpleTLS.S2 (left-hand side of Figure 3) on inputs b, A, and its server certificate cert. The server first computes the shared secret *SS*, then derives the *finished key FK* and the *handshake key HK* (lines 2–3). The former is used to authenticate h_1, a hash of the handshake, via the *server finished* value *SF*, and the latter is used to encrypt cert, so the server's identity is not revealed to eavesdroppers. The server then derives K from *SS* and a transcript hash (line 7), sends its reply to the client, and returns K.

SimpleBCO$(A, B, C_{\mathsf{cert}} || SF, C; a)$:

$K = \mathsf{SimpleTLS.C3}(a, (B, C_{\mathsf{cert}} || SF)$
$M \leftarrow \mathsf{Dec}(K, C)$
Return $(A = g^a)\ ?\ M\ :\ \perp$

SimpleSCO$(SF, C; SS, h_1)$:

$FK \leftarrow \mathsf{KDF}(SS, 1)$
$SF' \leftarrow \mathsf{HMAC}(FK, h_1)$
$kh \leftarrow H(h_1, SF')$
$K \leftarrow \mathsf{KDF}(SS, kh)$
$M \leftarrow \mathsf{Dec}(K, C)$
Return $(SF = SF')\ ?\ M\ :\ \perp$

Figure 4: Channel opening sub-circuits for SimpleTLS.

The client finishes by running SimpleTLS.C3 (right-hand side of Figure 3). It starts by deriving *SS*, *FK*, and *HK* (lines 1–3), then decrypting C_{cert}. Then, it derives *K*, checks *SF* (lines 5–8), and returns *K* only if the check passes.

Comparison to TLS 1.3. All the messages sent in SimpleTLS's handshake are also sent in a TLS 1.3 handshake. TLS 1.3 sends further values in its handshake—such as the server's signature. The value *SF*, while plaintext in SimpleTLS, is encrypted in TLS 1.3. The key schedule of SimpleTLS is substantially simplified compared to TLS 1.3; the operations that are relevant for explaining our optimizations, such as the derivation of the hash h_1, appear in both. Like SimpleTLS, TLS 1.3's key schedule derives *FK*, *HK*, and *K* from *SS*, but with more steps and inputs.

The challenge. We assume SimpleTLS (like TLS 1.3) supports only non-committing authenticated encryption schemes. For such schemes, it is possible to create a single ciphertext *C* with two valid (non-$\perp$) decryptions M_1, M_2 under two different keys K_1, K_2. Efficient algorithms for doing this are known [38, 64, 93]. Thus, the client could create a ciphertext with two decryptions M_1, M_2 where M_1 obeys the policy and M_2 violates it. Then the client could run Prove to generate a proof about M_1, and instead send M_2.[2] As a consequence, a naive channel opening solution, in which the sub-circuit takes *K* as a witness and verifies that decryption with *K* does not output $\perp$, is not secure.

A secure approach. Put differently, the problem with the naive approach is that nothing ensures that *K* (the key given as a witness to the proof) is the session key the server will use to decrypt. We prevent key equivocation in SimpleBCO (BCO is "baseline channel opening", top of Figure 4) by re-deriving the session key in the sub-circuit. The sub-circuit takes the handshake transcript and the client's ciphertext *C* as (public) input, and *a* as private input. The sub-circuit runs SimpleTLS.C3 using the public transcript and its private input to obtain key *K*, which it then uses to decrypt *C* and obtain *M*. Finally, it checks that *a* is consistent with the sent *A* and outputs the derived *M*.

Because *A* is a perfectly binding commitment to *a*, and *K* is determined only by *a* and the public transcript, this sub-circuit prevents equivocating about *K*. Our baseline channel-opening (BCO) sub-circuit for TLS 1.3 is described in Appendix B; it works similarly. In Appendix C we analyze BCO's security.

The SimpleBCO sub-circuit has very high proving costs. It requires two group operations: one to check $A = g^a$ and one to derive *SS* (line 1 of SimpleTLS.C3). It also requires decrypting (line 4) and hashing (line 5) the server's certificate (which could be a chain of multiple certificates, totaling thousands of bytes [35]). These costs are shared by BCO, our TLS 1.3 baseline, as well (§8.1).

A shortcut. The channel opening sub-circuit does not need to verify every step of SimpleTLS.C3, as the baseline does; for security, the sub-circuit need only check the correctness of *K*, which is derived from *SS* and *kh*. During the handshake, the server (line 5 of SimpleTLS.S2) sends *SF*, which is the MAC of two related values: the finished key *FK* and the hash h_1. By re-using *SF* as a check on the correctness of *SS* and h_1, the sub-circuit can take those arguments directly as input from the client instead of re-deriving them. This eliminates the two group operations and decrypting and hashing the certificate. Importantly, with this shortcut, the sub-circuit's size is independent of the handshake transcript length.

The pseudocode of the shortcut channel-opening (SCO) sub-circuit, SimpleSCO, is in Figure 4. Note that A, B, C_{cert} are not inputs, as SimpleSCO does not use them. We apply a very similar shortcut idea to reduce the cost of our channel opening sub-circuit for TLS 1.3, but the resulting sub-circuit is much more complicated due to details of the TLS 1.3 key schedule that SimpleTLS elides. We present this shortcut sub-circuit for TLS 1.3, SCO, in Figure 9 of Appendix B. In Appendix C we analyze the security of SCO, and show that it prevents the client from equivocating about *K*. We use a similar approach to build a sub-circuit for early data sent in a 0-RTT resumption in TLS 1.3. See Appendix B.3.

Amortization. Since *K* is fixed for the duration of a connection, deriving *K* can be done just once during the channel opening. This amortizes the work of deriving *K* across all ciphertexts sent. Specifically, after running SimpleTLS.C3 to finish a new SimpleTLS handshake, the client hashes *K* (call this hash h_{key}), then sends the middlebox the hash along with a proof that the hash's preimage is consistent with the handshake messages. This circuit consists of one hash computation, along with the operations not dependent on *C* in SimpleSCO. When the client sends a ciphertext *C*, the channel opening sub-circuit only needs to check that the decryption key's hash is equal to h_{key}. Appendix B.4 gives pseudocode

[2] One might wonder whether the channel opening sub-circuit could additionally 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. [38] demonstrated for images.

```
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 5: Parse-and-extract logic for HTTP firewalling.

for our amortized channel opening sub-circuit for TLS 1.3, and explains that, if we model the hash as a random oracle, the client cannot equivocate about K and h_{key} does not reveal K to the middlebox.

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 [115]. 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 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 traffic contains HTTP requests.

Proof Details. 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 5, 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.) 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

This section describes two more case studies.

One of them is a ZKMB that blocks 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. Sections 7.2 and 7.3 cover parse-and-extract for DoT and DoH, respectively. The policy check step (§7.4) proves $D \notin \mathsf{BL}$. We also show how to keep the blocklist hidden from the client (§7.5) if necessary.

The other case study is a ZKMB for *allow*listing resolvers in Oblivious DoH (§7.6).

These case studies assume that the middlebox requires proofs only for encrypted DNS queries, not all TLS traffic; the middlebox can distinguish encrypted DNS if given 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.[3] Thus, a user's queries reveal a great deal

[3] For brevity, we present a greatly simplified picture of the DNS ecosystem and direct the interested reader to [72, 98, 99] for more details.

of information about their web browsing habits. 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 [73] and DoH [70]. DoH has been supported by default for US users since mid-2020 by Mozilla [34], Google [5], and Apple [123]. Their browsers send users' queries to centralized encrypted DNS resolvers run by, for example, Cloudflare [112] or Google [129]. 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 [67]. 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.

In DoT, DNS queries are sent in TLS records preceded by a two-byte length indicator [73]. 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 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 approach is somewhat unnatural; the advantage is that the accessed locations of request and output depend only on the loop index, thereby avoiding overhead from 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) [46, 74, 79]. DNS clients convert non-ASCII labels in domain names to an ASCII-compatible encoding called Punycode [31]. (Punycode handles case sensitivity issues as well [68, 69].) 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`" [103]. 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 [80]. We implemented this for exact-match blocklisting, using Poseidon [60] 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.

7.5 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. (The value of this transparency has been observed elsewhere: for example, Mozilla's "trusted recursive resolver" policy requires DNS blocklists to be public [128].) 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 [100], 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.

7.6 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,

an extension to DoH which enables stronger privacy. ODoH adds a *proxy resolver* between the client and *target resolver*: the client encrypts (using HPKE [6]) the DNS query with the target resolver's public key and sends it, along with the target's identity, to the proxy resolver via HTTPS. (Specifically, 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 (§7.4).

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 (§5) and how effective are our optimizations?, (2) How well do our protocols perform for the case studies? (§6–§7), and (3) How tolerable or intolerable are the costs for real-world use?

Implementation. We implemented channel opening circuits for TLS 1.3, following the approach described for SimpleTLS in Section 5. Appendix B provides details for our full TLS 1.3 versions. We implemented parse-and-extract and policy-check circuits for all of the applications described in Sections 6–7, except the private blocklist extension of 7.5. Our implementation does not handle requests that are split across multiple TLS records. We did not implement a full ZKMB system, just the ZKMB circuits.

	Gate Counts ($\times 10^4$)					
Method	**EC**	**Hash**	**Dec**	**Total**	**Time**	**SRS**
BCO (§B.1)	274	255	219	748	94.0s	1.18GB
SCO (§B.2)	-	97	14	111	16.5s	149MB
ECO (§B.3)	-	61	-	61	8.6s	81MB
ACO^{AES} (§B.4)	-	0.03	19	19	4.4s	27MB
ACO^{Cha} (§B.4)	-	0.03	9	9	1.4s	14MB

Figure 6: The multiplication gate count, proof generation time, and SRS size for each channel opening sub-circuit described in Appendix B, operating on a 255-byte ciphertext. Costs are broken down into the elliptic curve (EC), hashing, and decryption (Dec) steps. $ACO^{AES/Cha}$ means Amortized Channel Opening, with AES-128-GCM or ChaCha20/Poly1305. Verification time is always less than 5 ms; verification key size is less than 1 MB.

Our implementation framework is xJsnark, a high-level language for writing zero-knowledge proofs [81]. 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 [61] protocol with the BN128 curve.

Harness and testbed. We wrote an experiment harness in Python to generate inputs for circuits, including simulated transcripts of TLS connections and DNS queries. We heavily modified the tlslite-ng library [125] to give us the ability to extract its internal state. We also used this harness to compute Merkle paths for our blocklisting policy in Section 7. We ran all our experiments using an Amazon EC2 instance `t3a.2xlarge`, using all cores of an 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 gigabytes of RAM.

8.1 Microbenchmarks for Channel Opening

Figure 6 provides microbenchmarks for our TLS 1.3 channel opening circuits. We measure the size (in gates) of the circuit output by xJsnark, the proof generation time (median of 5 trials), and SRS size for opening a 255-byte ciphertext. To evaluate the baseline channel opening circuit (BCO), we use an upper bound of 3000 bytes on the size of C_{EE} (the server's extensions). We use SHA-256 [49] and ECDHE over the secp256r1 (NIST P-256 [76]) curve in all cases; for decryption, we use either AES-128-GCM [43, 97] or ChaCha20/Poly1305 [89].

Figure 6 splits the results into the one-time cost of verifying the session key (§5) in the top three rows and the per-packet decryption cost in the bottom two rows. The shortcut channel opening circuit (SCO) improves on the baseline dramatically: proving time is 5× faster, and the circuit has 6× fewer gates.

Case Study		Gate Counts ($\times 10^4$) Channel Opening		Parse & Extract		Policy Check		Total	Time	SRS
Firewall	(§6)	SCO/500	149.0	Find First CRLF	0.9	Match HTTP	1.2	151.1	21.0s	208MB
DoT	(§7.2)	ACO^{Cha}/255	9.0	Extract DoT Label	5.5	Merkle Testing	4.0	19.5	3.1s	32MB
DoH `GET`	(§7.3)	ACO^{AES}/500	38.0	Extract DoH Label	7.5	Merkle Testing	4.0	49.5	6.8s	75MB
ODoH	(§7.6)	ACO^{AES}/500	38.0	Parse `targethost`	1.5	Match Resolver	8.5	48.0	6.4s	61MB

Figure 7: Gate counts, proving time, and SRS size for our case studies. Gate count is for the three stages of the ZKMB pipeline (§4). CO/N indicates packets N bytes long. Verification time is less than 5 ms in all cases; verification key size is less than 1 MB.

This shows that removing elliptic curve operations and reducing the cost of h_3 derivation produces substantial benefit. Also noteworthy is the three-second improvement in proving time offered by ChaCha20 vs. AES-GCM.

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 either SCO or ACO for channel opening, then apply the parse-and-extract and policy check sub-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 7. For this case study, the dominant cost, by far, is channel opening: SCO by itself costs nearly 1.5 million gates, where about 380,000 are for AES decryption; the other two steps take just 21,000.

Domain Blocklisting for DoT/DoH (§7.2, §7.3). For this experiment, we use the Python experiment harness described above to create the Merkle tree and compute Merkle paths for the non-membership proof. We use the Poseidon [60] hash function; because of a subtlety with how Poseidon works, our implementation only supports 253-byte domain names, one less than the real length limit. We use a blocklist containing two million entries, obtained from an online blocklist of adult-content domains [96], which we believe to be comparable to the size of proprietary adult-content blocklists [147]. Because the blocklist must be downloaded by the client during Registration, a practical question is the size of the blocklist. Our two million domain blocklist is about fifty megabytes in size and can be compressed with gzip to just over seven 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 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.

Our DoT and DoH experiments use ACO. Because encrypted DNS uses a long-lived TLS connection, it makes sense to run the setup once and use amortized channel opening for each packet. For DoT we use a 255-byte ciphertext with ChaCha20/Poly1305 decryption. Overall, the cost is only 195,000 gates. Parse-and-extract and policy check take only 90,000 gates. Even for our fairly large blocklist, proving non-membership costs only 40,000 gates: Merkle paths are compact and Poseidon's circuit is small, costing only 250 gates.

We next evaluate DoH `GET` with AES and a 500 byte ciphertext. The cost is much higher than the DoT circuit: 495,000 gates. The parse-and-extract policy check costs about 115,000 gates; thus, the bulk of the cost is AES decryption.

Resolver Allowlisting for ODoH (§7.6). We evaluate our ODoH case study described in Section 7.6 with the amortized channel opening circuit for a 500-byte ciphertext. Our current implementation of the policy check circuit only supports a one-entry allowlist and cannot yet use Merkle proofs. This experiment results in similar performance to DoH.

Summary. Overall, our experiments indicate ZKMBs are not yet practical for all use cases: for example, SRS sizes (up to 235 MB) are a barrier to deployment. Also, proving time is too high in our HTTPS firewalling case study, where each new connection would have a ≈20-second setup cost.

On the other hand, in some settings ZKMBs' proving time is nearly practical. For encrypted DNS, where the client has a long-lived TLS connection to a DNS resolver, the setup time must be incurred only once. Then, each query for a new domain costs only 3 seconds, which, while not ideal, is within bounds for latency-sensitive workloads, like a human browsing the web. A browser implementation could even precompute proofs as DNS names are prefetched.

9 Related Work

Implementations of probabilistic proofs. Probabilistic proofs are fundamental in cryptography and complexity the-

ory [2–4, 55–57]. 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 [124, 131, 134] 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 [61] back-end, summarized earlier (§3). Groth16 is built atop the seminal QAP formalism [54], which was initially implemented in Pinocchio [108] (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 [133, §8] and Spartan [116, §9]. Subsequent work has not changed the qualitative costs and trade-offs, though there has been progress in relaxing setup assumptions [1, 8, 11, 21, 25, 26, 52, 91, 92, 116, 117, 133, 145] and extending the machinery to circuits and statements of greater size and scope via recursive proof composition [16, 20, 53, 82, 83].

Turning to front-end work, there are two main approaches: "ASIC" and "CPU" [131, 134]. We use the xJsnark [81] front-end, which is an "ASIC" approach, meaning that each high-level computation transforms to a custom circuit **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 [17, 18, 28, 32, 106, 108, 109, 118, 132, 148]. A variant of this approach is to hand-design circuits by exploiting recent enhancements to arithmetization [14, 51]. In contrast, the "CPU" front-end approach encodes a "universal machine" in a circuit [8–10, 146]. This approach, while in principle facilitating programmability, results in performance that is not competitive (overhead of 50× [132]).

Applications of general-purpose ZK proofs have so far mainly surrounded cryptocurrency [114] (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 to cope with encrypted traffic. Sherry's dissertation provides an overview [120] while Naylor et al. [101] provide a thoughtful framework for comparing different solutions [101].

TEE-based approaches. ETTM [37] 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 [58] 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 to which key material is disclosed [66, 110, 127]; the motivation is enterprise-deployed middleboxes running on untrusted cloud platforms. mbTLS [101] replaces end-to-end TLS with hop-by-hop TLS with TEE enclaves implementing middlebox functionality between each hop in a manner agreed by both endpoints. These works and many others [42, 59, 65, 84, 135] rely on trusted enclaves, which we consider not to meet our compatibility requirement (§1).

TLS modification approaches. Like ZKMBs, Blindbox [119] 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 [88] and many others [47, 77, 86, 87, 90, 94, 104, 105, 111, 140, 143].

Another approach is mcTLS [102], which maintains the standard cryptographic properties of TLS but enables granting one or more middleboxes read and/or write access to certain portions of a TLS connection by revealing context-specific keys. Middleboxes can thereby enforce network policy or provide functionality without removing all security benefits of TLS. For example, DNS filtering can be implemented using mcTLS by granting the middlebox read-only access to the request body, and traffic compression can be implemented by granting read/write access to the response body.

All of these proposals require server-side changes, which we regard as incompatible with widespread adoption.

Comparison to ZKMBs. Assessing our work using Naylor et al.'s taxonomy [101] of design options for middlebox-enabling modifications to TLS highlights that ZKMBs represent a new point in the design space. On the one hand, ZKMBs don't support middleboxes with write access to traffic (only read access); don't (by design) provide transparency to servers about the presence of middleboxes; and don't provide path integrity [101]. On the other hand, in ZKMBs, the middlebox learns only that traffic policy is compliant, as opposed to (in mcTLS) reading all or part of the client's request. ZKMBs thus represent a new category within Naylor et al.'s "granularity of data access" property. A potential avenue for future work is to combine the zero-knowledge data access of ZKMBs with other properties like path integrity or bilateral transparency achieved by mcTLS or mbTLS.

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 [23], which are public-key encryption schemes that enable efficient generation of proofs for predetermined functions of plaintexts (for example, that the message is the dis-

crete 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. [136] 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. [144] 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 via secure multiparty computation (MPC). Clients can optionally prove statements about responses using a ZK proof system instead of revealing them. In an appendix, they suggest that the third party could be an inline proxy, to reduce MPC overhead.

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 sub-circuits 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, though they consider TLS 1.3, their main focus is on TLS 1.2 and optimizing for HMAC-then-CBC encryption, which is unavailable 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) while keeping the circuit as compact as possible (for performance). Another challenge was embedding DNS filtering (§7) in a circuit.

In our experiments (§8), proof length is 128 bytes, and the verifier (middlebox) runs in 2–5ms. These quantities are inherited from Groth16 [61].

Three aspects of performance give us pause for near-term ZKMB deployment. First, even several milliseconds of verification time (roughly, 300–500 proofs per second) could be too costly, when considering that network links often carry many more then several hundred packets per second. A mitigating factor, however, is that policy enforcement need not be synchronous. Although we have described the middlebox as dropping non-compliant traffic, an alternate arrangement would have the middlebox *auditing*: verifying proofs, perhaps in batches, after it has forwarded traffic. The remediation for an invalid proof, which is to say a failed audit, would be blocking the client from the network.

Second, the prover's (client's) running time is around 14 seconds to initiate a TLS connection and several seconds for any packets that call for proofs. On the other hand, there are specialized settings, such as encrypted DNS (§7), where the initiation cost amortizes and the per-proof cost is affordable, or nearly so (§8).

Third, the SRS (§3) is large (hundreds of megabytes). How does the SRS get to clients? If each middlebox generates its own, clients have to download a large package when joining a new network. Alternatively, a trusted entity (such as a browser vendor) could install on clients SRSes for appropriate, fixed circuits. A maliciously generated SRS is a risk only to middleboxes, not clients: a compromised SRS would undermine soundness of the proof system, enabling policy circumvention via forged proofs, but clients' privacy would not be affected [50]. Notice that if the middlebox needs to update policy, pre-installation is not a practical 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 (§7.5), supporting requests that span several TLS records, and implementing functionality that necessitates proofs per packet (such as IDS and malware scanning).

There are several larger extensions to explore. One is *circuit decomposition*, in which each of channel opening, parse and extract, and policy-check (§4) 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, 52, 95] 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 over-

head [1,11,92,116,145] and require no trusted setup, though at the expense of larger verifier running time and larger proofs. Indeed, preliminary investigations indicate that by using a MPC-style proof protocol [44, 138, 139, 142], proving times of roughly one second are achievable, at the cost of verifier time rising to one second as well.

Another interesting direction is extending ZKMBs to other middlebox architectures. We focused here on the simplest case, a single middlebox located in the same local network as the client. We expect ZKMBs to adapt naturally to other settings, for example a middlebox interposed in front of the server, a middlebox running "off-path" in the cloud (§9), a chain of middleboxes enforcing different policies, stateful middleboxes, or traffic-rewriting middleboxes. Finally, a challenging task for future work is an implementation of a ZKMB that runs at the network's line rate.

Acknowledgements

We are indebted to Chris Wood for suggestions and insights about DNS throughout this project. Srikar Varadaraj provided help during the early stages of this work. Malavika Balachandran Tadeusz, Bill Budington, Richard Clayton, Henry Corrigan-Gibbs, Zachary DeStefano, Felix Günther, Anurag Khandelwal, Ian Miers, Eric Rescorla, Justin Thaler, Collin Zhang, Pete Zimmerman, and the anonymous reviewers gave helpful 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*, 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.

[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*, 2014.

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

[12] 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*, 2012.

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

[14] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution. In *IACR Asiacrypt*, 2018.

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

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

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

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

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

[20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Proof-Carrying Data from Accumulation Schemes. In *IACR TCC*, 2020.

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

[22] Jonas Bushart and Christian Rossow. Padding Ain't

Enough: Assessing the Privacy Guarantees of Encrypted DNS. In *USENIX FOCI*, 2020.

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

[24] Configuring networks to disable DNS over HTTPS. https://support.mozilla.org/en-US/kb/configuring-networks-disable-dns-over-https, 2022.

[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*, 2020.

[27] Children's Internet Protection Act. 106th U.S. 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.

[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.

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

[35] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *IEEE Security & Privacy*, 2016.

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

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

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

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

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

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

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

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

[44] EMP-toolkit: Efficient MultiParty computation toolkit, 2021. https://github.com/emp-toolkit/emp-zk.

[45] 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.

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

[47] 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.

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

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

[50] Georg Fuchsbauer. Subversion-zero-knowledge SNARKs. In *IACR PKC*, 2018.

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

[52] 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.

[53] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. Cryptology ePrint Archive, Report 2021/529, 2021.

[54] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *IACR Eurocrypt*, 2013.

[55] Oded Goldreich. Probabilistic proof systems – a primer.

Foundations and Trends in Theoretical Computer Science, 3(1), 2008.

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

[57] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1), 1989.

[58] 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.

[59] 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.

[60] 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.

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

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

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

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

[65] 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.

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

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

[68] Paul Hoffman and Marc Blanchet. Preparation of Internationalized Strings (s̈tringprep)̈. RFC 3454, 2002.

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

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

[71] 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.

[72] Austin Hounsel, Paul Schmitt, Kevin Borgolte, and Nick Feamster. Designing for tussle in encrypted DNS. In *ACM HotNets*, 2021.

[73] 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.

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

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

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

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

[78] 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.

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

[80] Paul C Kocher. On certificate revocation and validation. In *Financial Crypto*, 1998.

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

[82] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. Poppins: A direct construction for asymptotically optimal zkSNARKs. Cryptology ePrint Archive, Report 2020/1318, 2020.

[83] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. Cryptology ePrint Archive, Report 2021/370, 2021.

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

[85] SSL Labs. SSL Pulse. https://www.ssllabs.com/ssl-pulse/.

[86] 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.

[87] 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 TDSC*, 2021.

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

[89] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, June 2016.

[90] 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.

[91] Jonathan Lee. Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments. Cryptology ePrint Archive, Report 2020/1274, 2020.

[92] 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.

[93] Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning Oracle Attacks. In *USENIX Security*, 2021.

[94] 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.

[95] 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.

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

[97] David McGrew and John Viega. The galois/counter mode of operation (GCM). *NIST Modes of Operation Process*, 20, 2004.

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

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

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

[101] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And Then There Were More: Secure Communication for More Than Two Parties. In *ACM CoNEXT*, 2017.

[102] 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.

[103] 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.

[104] 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.

[105] 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.

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

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

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

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

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

[111] 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.

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

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

[114] 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.

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

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

[117] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275, 2020.

[118] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Tak-

ing proof-based verified computation a few steps closer to practicality. In *USENIX Security*, 2012.

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

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

[121] Haya Shulman. Pretty bad privacy: Pitfalls of DNS encryption. In *ACM WPES*, 2014.

[122] 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.

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

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

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

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

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

[128] Mozilla policy requirements for DNS over HTTPs partners. https://wiki.mozilla.org/Security/DOH-resolver-policy, 2022.

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

[130] W3Schools. Chrome Statistics. https://www.w3schools.com/browsers/browsers_chrome.asp.

[131] Riad Wahby. Practical proof systems: implementations, applications, and next steps. https://www.pepper-project.org/simons-vc-survey.pdf, September 2019.

[132] 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*, 2015.

[133] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE Security & Privacy*, 2018.

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

[135] 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.

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

[137] 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.

[138] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *IEEE Security & Privacy*, 2021.

[139] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *USENIX Security*, 2021.

[140] 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.

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

[142] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *ACM CCS*, 2021.

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

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

[145] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *IEEE Security & Privacy*, 2020.

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

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

[148] 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 hkdf.ext, and one which *expands* extracted entropy into key material, which we denote hkdf.exp. In TLS1.3, hkdf.ext takes two arguments (x,y) and simply outputs $\mathsf{HMAC}(x,y)$.

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 hkdflabel as

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

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

$$\begin{aligned}&\mathsf{hkdf.exp}(K, \mathsf{lbl}, \mathsf{cnxt}, \ell) \\ &\quad = \mathsf{HMAC}(K, \mathsf{hkdflabel}(\mathsf{lbl}, \mathsf{cnxt}, \ell) || \ 0\text{x}01)\end{aligned}$$

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

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

All 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 [39] so that distinct labels have distinct numbers. See [113] 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 $\text{GCM-CTR}[E](K,N,M,\ell_1,\ell_2)$ 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 $\text{GCM-CTR}[E](K,N,M)$ to mean $\text{GCM-CTR}[E](K,N,M,1,\mathsf{len}(M))$. 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.

The derived early secret *dES*. One particular value in the key schedule needs additional explanation. The derived early secret *dES* is passed as an input to the derivation of the handshake secret *HS*. It is the result of the "early" stage of the key schedule, which derives secrets for use during resumption and PSK-authenticated handshakes. When there is no PSK (i.e., during standard handshakes), *dES* is $\mathsf{hkdf.exp}(\mathsf{hkdf.ext}(0,0), \mathsf{lbl}_3, h_\varepsilon)$.

B Channel Opening for TLS 1.3

This section describes our channel opening sub-circuits for TLS 1.3 [113]. These sub-circuits are used by all of our case studies (§6–§7).

TLS 1.3 Overview. TLS 1.3 sessions involve a client, who initiates the connection, and a server. TLS 1.3 has two subprotocols: a *handshake* protocol and a *record* protocol. The handshake protocol begins with the client constructing a "Client Hello" message *CH*. (This corresponds to C1 in our simplified SimpleTLS.Handshake in Section B.) 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 messages: first, a "Server Hello" message *SH* of its own, containing a random N_s and key share g^b. (In SimpleTLS, the corresponding message is the DH value B sent during SimpleTLS.S2.

The second message is a ciphertext C_{EE}. It contains the server's authentication data, namely its certificate and both a signature σ and MAC *SF* (the "ServerFinished" value) of the transcript. The encryption key is derived from the shared DH secret and a hash of $CH||SH$. (The corresponding message in SimpleTLS is $C_{\mathsf{cert}}||SF$, sent during SimpleTLS.S2. See Figure 3. Note that in TLS 1.3, *SF* is encrypted, while in

SimpleTLS, it is sent in the clear.)

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. (For simplicity, SimpleTLS does not send a client finished value, but its handshake is completed with SimpleTLS.C3 in Figure 3.)

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.[4] (Readers will find some additional preliminaries on algorithms used in TLS 1.3 in Appendix A.)

Resumption in TLS 1.3. 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".

B.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*. The resulting sub-circuit is depicted in pseudocode in Figure 8. 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. Here, as in Section 5, we depict the sub-circuit as decrypting C to derive message M and delivering it to the next sub-circuit in the pipeline.

This sub-circuit works the same way as our baseline sub-circuit for SimpleTLS in Figure 4, namely by re-running in the sub-circuit all the operations the client uses to derive the session key K when it receives the server's reply.

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. (See Section 8 for a more detailed breakdown of costs.) Some of these costs are inherent and stem from TLS 1.3's use of cryptographic primitives like AES and SHA-256. For example, each hkdf.ext or hkdf.exp operation requires about 100,000 multiplication gates. However, as with the SimpleTLS baseline SimpleBCO, the two most expensive parts of the sub-circuit are (1) the two group operations, which in TLS 1.3 are elliptic-curve scalar multiplications, and (2) computing the transcript hash h_3, which corresponds to *kh* in Figure 3. The two elliptic-curve operations together cost 2.4 million gates, while computing h_3 costs 4 million gates for a 3000-byte transcript. The sub-circuit size must also depend on the transcript length; thus, the sub-circuit must be padded to handle the largest possible transcript.

[4] 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 [71].

BCO$(CH, SH, C_{EE}, C; a)$:
$K \leftarrow \mathsf{BKV}(CH, SH, C_{EE}, a)$
$M \leftarrow \mathsf{Dec}(K, C)$
Return M

BKV(CH, SH, C_{EE}, a):
$A \leftarrow \mathsf{GetDH}(CH)$
$B \leftarrow \mathsf{GetDH}(SH)$
$\mathsf{DHE} \leftarrow B^a$ // 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 ? $CATK$: $\perp$

Figure 8: Baseline channel opening circuit. All notation is defined in Appendix A. Certain details, such as label inputs to hkdf.exp and TLS record headers, are omitted.

B.2 Optimizing the Baseline

In Section 5, we explain how the shortcut SimpleSCO repurposes the *SF* value sent in SimpleTLS as a commitment to two intermediate values of the key schedule. Here we show that a very similar optimization is possible for TLS 1.3.

Figure 9 shows pseudocode for this TLS 1.3 shortcut sub-circuit SCO. For simplicity, the figure presents a simplified variant of the optimization; we discuss the differences at the end of this subsection. It uses 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.

$\underline{\text{SCO}(CH, SH, C^{\text{suff}}_{EE}, C\,; HS, \mathsf{LCCV}, \mathsf{vsuff})}$:
$K \leftarrow \text{SKV}(CH, SH, C^{\text{suff}}_{EE}, HS, \mathsf{LCCV}, \mathsf{vsuff})$
$M \leftarrow \mathsf{Dec}(K, C)$
Return M

$\underline{\text{SKV}(CH, SH, C^{\text{suff}}_{EE}, HS, \mathsf{LCCV}, \mathsf{vsuff})}$:
$SHTS \leftarrow \mathsf{hkdf.exp}(HS, H(CH||SH))$
$SHTK \leftarrow \mathsf{hkdf.exp}(SHTS, \text{“shtkey”})$
$FK_S \leftarrow \mathsf{hkdf.exp}(SHTS, \text{“fkey”})$
$h_7 \leftarrow f(\mathsf{LCCV}, \mathsf{PadLB}(\mathsf{vsuff}))$
$SF' \leftarrow \mathsf{HMAC}(FK_S, h_7)$
$SF \leftarrow \mathsf{Dec}(SHTK, C^{\text{suff}}_{EE})$ // 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 $(SF' = SF)\ ?\ CATK : \perp$

Figure 9: The shortcut channel opening circuit. All notation is defined in Appendix A. Certain details, such as label inputs to hkdf.exp and TLS record headers, are omitted.

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^{suff}_{EE}. These public inputs are roughly the same as for SimpleSCO, except C^{suff}_{EE} corresponds to SF. (Note that since the client and server Hello messages are public, and SCO only needs their hash, the middlebox can simply compute the hash itself instead of incurring overhead in circuit size proportional to the lengths of the Hello messages.)

SCO's private input is the handshake secret HS (corresponding to SS in SimpleTLS). 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. These two values correspond to h_1 in SimpleSCO.

The optimization happens in SKV, which works as follows. It first derives $SHTK$, which we used in BKV to decrypt C_{EE}. Here we use it to decrypt only the last 32 bytes, C^{suff}_{EE}, containing the finished value SF. It then uses the chaining value LCCV and vsuff to complete the derivation of h_7, re-derives SF' (SimpleSCO does the same thing), and then appends SF to vsuff and applies f to recompute h_3 (the line that derives kh in SimpleSCO). This optimization works because the inputs to h_7 and h_3 share a common prefix, to which SF commits. Because of the way we simplified SimpleTLS, there is not a direct correspondence to how SimpleSCO works, but the value h_1 passed as a witness to SimpleSCO is playing the part both of h_7 and LCCV here. Likewise, h_3 is equivalent to kh.

With HS and h_3, SKV re-derives $CATK$. It then ensures that SF obtained from the ciphertext is the same as the re-computed one SF' before returning $CATK$. This corresponds to the last three lines of SimpleSCO.

This description of SCO 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 SCO, 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 SCO somewhat. Appendix C further discusses this point.

$\underline{\text{ECO}(CH, C_{\text{ED}}\,; \mathsf{psk})}$:
$K \leftarrow \text{EKV}(CH, \mathsf{psk})$
$M \leftarrow \mathsf{Dec}(K, C_{\text{ED}})$
Return M

$\underline{\text{EKV}(CH, \mathsf{psk})}$:
$ES \leftarrow \mathsf{hkdf.ext}(\mathsf{psk})$
$BK \leftarrow \mathsf{hkdf.exp}(ES, \text{“resbinder”})$
$fk_B \leftarrow \mathsf{hkdf.exp}(BK, \text{“fkey”})$
$CH_{NE}, \mathsf{PSKExt}, BD \leftarrow CH$
$h_5 \leftarrow H(CH_{NE}||\mathsf{PSKExt})$
$BD' \leftarrow \mathsf{HMAC}(fk_B, h_5)$
$h_1 \leftarrow H(CH_{NE})$
$ETS \leftarrow \mathsf{hkdf.exp}(ES, h_1)$
$EATK \leftarrow \mathsf{hkdf.exp}(ETS, \text{“tkey”})$
Return $(BD' = BD)\ ?\ EATK\ :\ \perp$

Figure 10: Channel opening circuit for early data sent in a 0-RTT resumption handshake. All notation is defined in Appendix A. Certain details, such as label inputs to hkdf.exp and TLS record headers, are omitted.

B.3 Opening Early Data

Next, we describe our channel opening circuit for data sent with the 0-RTT resumption feature of TLS 1.3. The pseudocode is in Figure 10. The circuit takes as public input CH and the client's early data ciphertext C_{ED}, and as private input the PSK psk. The main idea is to use the PSK binder as a commitment to psk, analogous to the Finished value SF in SCO and SimpleSCO.

The key consistency procedure EKV 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 sub-circuit is actually the closest to SimpleSCO: like SimpleTLS, the binder value for early data in TLS 1.3 is sent in the clear, and commits to psk.

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.

B.4 Security and Discussion

Appendix C contains a provable-security analysis of the security of BCO and SCO, showing that both prevent the client from equivocating about M or K during an interaction with an honest server. We do not give a full analysis of ECO; however, in Appendix C we explain why it follows easily from the analysis of SCO.

Amortizing channel opening. In Section 5, we explain that for long-lived SimpleTLS connections, it makes sense to check consistency between the key K and the handshake only once, when the connection is set up, then re-use this work for all subsequent packets. We sketched an approach to doing this for SimpleTLS; here, we present our corresponding "amortized" channel opening sub-circuit for TLS 1.3:

$\underline{\text{AKV}(CH, SH, C_{EE}^{\text{suff}}, h_{\text{key}}; HS, \mathsf{LCCV}, \mathsf{vsuff}):}$
$K \leftarrow \text{SKV}(CH, SH, C_{EE}^{\text{suff}}, HS, \mathsf{LCCV}, \mathsf{vsuff})$
$h_{\text{key}} \leftarrow H(K)$
Return h_{key}

$\underline{\text{ACO}(C, h_{\text{key}}; K):}$
$M \leftarrow \mathsf{Dec}(K, C)$
Return $(h_{\text{key}} = H(K))\ ?\ M :\ \perp$

The SKV in this pseudocode is the same procedure as in Figure 9. Given that SKV outputs the correct session key K derived by the client and server during the TLS 1.3 handshake, if we model the hash function as a random oracle, this sub-circuit prevents equivocation and hides the session key from the middlebox.

After the middlebox has verified the proof, it can be convinced that h_{key} is the hash of the session key. Then, the client can use the above ACO sub-circuit per-packet channel opening. As described in Section 5, we assume the middlebox keeps track of the sequence number for the TLS 1.3 session; this needs to be given as input to decryption, but we have omitted it in pseudocode for simplicity.

C Security Analyses

In this appendix, we analyze the security of our channel opening circuits from Section 5. We begin with the baseline BCO. The security experiment, $\mathsf{TLS\text{-}BIND}_{BL}$ is depicted in Figure 11. 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 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 and one of the handshake transcripts stored in the table T. The adversary wins if BKV returns a key/IV pair *different* than the one stored in the table for this handshake. Now we can state and prove the security theorem for BCO.

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*

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

Proof: The proof of this theorem is extremely simple. The adversary's CH contains g^a, which is a perfectly binding commitment to DH secret a and is checked inside BKV. Since fixing a and the handshake fixes the client keys, the adversary cannot cause BKV to produce any output other than the keys derived in the oracle call that inserted the adversary's handshake (CH, SH, C_{EE}) into the table.

Note that in our security experiment, the client has no control over the randomness the server uses to generate its DH

$\underline{\mathsf{TLS\text{-}BIND}^{\mathcal{A}}_{BL}}$:
$a, CH, SH, C_{EE} \xleftarrow{\$} \mathcal{A}^{O}$
If $T[(CH, SH, C_{EE})] = \perp$ then Return 0
$(CATK, CATN) \leftarrow T[(CH, SH, C_{EE})]$
$(CATK', CATN') \leftarrow \mathrm{BKV}(CH, SH, C_{EE}; a)$
Return $(CATK, CATN) \neq (CATK', CATN')$

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

$\underline{\mathrm{BKV}(CH, SH, C_{EE}; a)}$:
$A \leftarrow \mathsf{GetDH}(CH)$
$B \leftarrow \mathsf{GetDH}(SH)$
$HS \leftarrow \mathsf{hkdf.ext}(dES, B^a)$
$CATK, CATN \leftarrow \mathrm{CATKeyGen}(HS, CH, SH, C_{EE})$
Return $(g^a = A)$? $(CATK, CATN)$: $\perp$

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

Figure 11: Security experiment for binding security of the BCO protocol in Section 5. Notation is explained there and in the prose in this appendix. The string SC is an arbitrary fixed string. DeriveTK is defined in Appendix A.

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 SCO

Next we analyze the security of the optimized SCO circuit discussed in Section 5. Note that this appendix presents a slightly different variant of the protocol: SKV 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 12. 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 12, 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. Then*

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

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 SKV to derive keys. The first game, G_0, is depicted in Figure 12. 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.

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

$HS, \mathsf{LCCV}, CH, SH, C_{EE} \leftarrow\!\!\$\ \mathcal{A}^{O}$
If $T[(CH, SH, C_{EE})] = \perp$ then Return 0
$(CATK, CATN) \leftarrow T[(CH, SH, C_{EE})]$
$(CATK', CATN') \leftarrow \mathsf{SKV}(CH, SH, C_{EE}; HS, \mathsf{LCCV})$
Return $(CATK, CATN) \neq (CATK', CATN')$

$\underline{\mathsf{SKV}(CH, SH, C_{EE}; HS, \mathsf{LCCV})}$:

$h_2 \leftarrow H(CH||SH)$
$SHTS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_5, h_2)$
$(SHTK, SHTN) \leftarrow \mathsf{DeriveTK}(SHTS)$
$FK_S \leftarrow \mathsf{hkdf.exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$
$\ell_{EE} \leftarrow \mathsf{len}(C_{EE})$
$\ell_{h7tr} \leftarrow \mathsf{len}(CH||SH) + (\ell_{EE} - 32)$ // length of h_7 input
$\ell_{lb} \leftarrow \ell_{h7tr} \mod 64$ // length of suffix of EEP in last f call
$SF \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31, \ell_{EE})$
$\text{suff} \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31 - \ell_{lb}, \ell_{EE} - 31)$
// suff is the suffix of EEP in last f call
$h_7 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}))$
$h_3 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}||SF))$
$SF' \leftarrow \mathsf{HMAC}(FK_S, h_7)$
$dHS \leftarrow \mathsf{hkdf.exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$
$MS \leftarrow \mathsf{hkdf.ext}(dHS, 0)$
$CATS \leftarrow \mathsf{hkdf.exp}(MS, \mathsf{lbl}_7, h_3)$
$CATK, CATN \leftarrow \mathsf{DeriveTK}(CATS)$
Return $(SF = SF')\ ?\ (CATK, CATN)\ :\ \perp$

$\underline{O(CH)}$:

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

Figure 12: Security experiment for binding security of SCO in Section 5. The string SC is an arbitrary fixed string. DeriveTK is defined in Appendix A.

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 SKV 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 SKV 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 SKV, 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 SKV and that oracle call; since f's output cannot collide for distinct inputs, the handshake traffic

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

$HS_{\text{out}}, \mathsf{LCCV}_{\text{out}}, CH, SH, C_{EE} \xleftarrow{\$} \mathcal{A}^{O,f,E}$

If $T[(CH,SH,C_{EE})] = \perp$ then Return 0

$(CATK_O, CATN_O) \leftarrow T[(CH,SH,C_{EE})]$

$HS_O, \mathsf{LCCV}_O \leftarrow W[(CH,SH,C_{EE})]$

$(CATK_{\text{out}}, CATN_{\text{out}}) \leftarrow \text{SKV}(CH,SH,C_{EE}; HS_{\text{out}}, \mathsf{LCCV}_{\text{out}})$

Return $(CATK_O, CATN_O) \neq (CATK_{\text{out}}, CATN_{\text{out}})$

$\underline{\text{SKV}(CH,SH,C_{EE}; HS, \mathsf{LCCV})}$:

$h_2 \leftarrow H[f](CH||SH)$

$SHTS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_5, h_2)$

$SHTK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$

$SHTN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{IV}}, h_\varepsilon, 12)$

$FK_S \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$

$\ell_{EE} \leftarrow \mathsf{len}(C_{EE})$

$\ell_{h7tr} \leftarrow \mathsf{len}(CH||SH) + (\ell_{EE} - 32)$

$\ell_{lb} \leftarrow \ell_{h7tr} \mod 64$

$SF \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31, \ell_{EE})$

$\text{suff} \leftarrow \text{GCM-CTR}[E](SHTK, SHTN, C_{EE}, \ell_{EE} - 31 - \ell_{lb}, \ell_{EE} - 31)$

$h_7 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}))$

$h_3 \leftarrow H[f](\mathsf{LCCV}, \mathsf{PadLB}(\text{suff}||SF))$

$SF' \leftarrow \mathsf{HMAC}[f](FK_S, h_7)$

$dHS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_3, h_\varepsilon)$

$MS \leftarrow \mathsf{hkdf}[f].\mathsf{ext}(dHS, 0)$

$CATS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(MS, \mathsf{lbl}_7, h_3)$

$CATK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$

$CATN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(CATS, \mathsf{lbl}_{\text{IV}}, h_\varepsilon, 12)$

Return $(SF = SF')\ ?\ (CATK = CATN)\ :\ \perp$

$\underline{O(CH)}$:

$A \leftarrow \mathsf{GetDH}(CH)$

$N_S \xleftarrow{\$} \{0,1\}^{256}$

$b \xleftarrow{\$} \{1,\ldots,p\}; B \leftarrow g^b$

$SH \leftarrow \text{MakeServerHello}(N_S, B)$

$HS \leftarrow \mathsf{hkdf}[f].\mathsf{ext}(dES, A^b)$

$h_2 \leftarrow H[f](CH||SH)$

$SHTS \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(HS, \mathsf{lbl}_5, h_2)$

$SHTK \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{key}}, h_\varepsilon, 16)$

$SHTN \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_{\text{IV}}, h_\varepsilon, 12)$

$FK_S \leftarrow \mathsf{hkdf}[f].\mathsf{exp}(SHTS, \mathsf{lbl}_6, \varepsilon)$

$\mathsf{tr}_{h7} \leftarrow CH||SH||SC$

$\ell_{lb} \leftarrow \mathsf{len}(\mathsf{tr}_{h7}) \mod 64$

$fbs \leftarrow (\mathsf{len}(\mathsf{tr}_{h7}) - \ell_{lb})/64$ // #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 13: Game G_0 of the proof of Theorem 2.

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 inequality and summing the right-hand sides of each bound completes the proof. ∎

This analysis of SCO 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 prevents 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 modeled as an ideal random function.

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 BCO and SCO channel opening circuits. We omit analyses of our ECO circuit for early data, and our amortized ACO 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 SCO 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.

D Artifact Appendix

D.1 Abstract

The purpose of this artifact is to allow reproduction of the performance results in Section 8, specifically the channel opening microbenchmark table (Figure 6) and the full proof generation benchmark for the case studies (Figure 7). All runtime estimates in this abstract are for a Linux system with an 8-core 2.2 GHz AMD EPYC 7571 CPU and 32 GB of RAM. All of our code is available on GitHub.

After installing dependencies, which should take roughly ten minutes, reproducing these results has two steps: (1) circuit generation and (2) proof generation. Circuit generation takes as input a (roughly) human-readable programmatic description of our circuits written in an extension of Java, and outputs a gate-level description of the corresponding arithmetic circuit. This programmatic circuit description is an intermediate representation obtained by partially compiling the original handwritten xJsnark source code. We do not require the original xJsnark source for the artifact evaluation—reading it requires installing a specific version of a large and unwieldy IDE called MPS—but our GitHub repository includes instructions on viewing the xJsnark source.

Circuit generation involves heavily optimizing the circuit description, and so is computationally quite expensive, and will take up to twenty minutes (to generate the nine example circuits in this artifact). The purpose of re-running the circuit generation as part of the artifact is to allow users to reproduce the claimed gate counts for our circuits. We provide a single script to automatically perform all of circuit generation.

After the circuits' descriptions have been generated, the last step is proof generation. Proof generation takes as input

the circuit descriptions as well as sample circuit inputs (e.g., TLS handshake transcripts and ciphertexts), generates public parameters, produces proofs, and verifies them. The provided proof generation script outputs information about the time taken to generate and verify proofs, as well as the sizes of the public parameters. We estimate this will take in total up to twenty minutes (to complete all nine circuits in the artifact).

D.2 Artifact check-list (meta-information)

- **Algorithm:** zkSNARKs, Groth16
- **Program:** xJsnark, libsnark
- **Compilation:** Java, cmake
- **Data set:** Manually generated test data. Included.
- **Run-time environment:** Ubuntu 20.04, OpenJDK 11.0.13
- **Hardware:** 32 GB RAM, 8 cores
- **Metrics:** Circuit size, proving time, verification time, parameter size
- **Experiments:** Bash scripts
- **How much disk space required (approximately)?:** 3 GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min
- **How much time is needed to complete experiments (approximately)?:** 40 min
- **Publicly available:** GitHub: https://github.com/pag-crypto/zkmbs/
- **Stable URL:** https://github.com/pag-crypto/zkmbs/tree/096ed18772d8e63f4a03e7f4d16e118aa3923135

D.3 Description

D.3.1 How to access

Our artifact's code is publicly available on GitHub here:
https://github.com/pag-crypto/zkmbs

This appendix contains all the instructions specific to installation and reproducing the paper's benchmarks.

D.3.2 Hardware dependencies

We recommend using a machine with 8 cores and at least 32 GB RAM.

D.3.3 Software dependencies

The only major dependency is Java. We recommend using a GNU/Linux system and have provided installation scripts compatible with the Ubuntu 20.04 Linux distribution.

D.4 Installation

1. Clone the git repository and change to the root directory (time required: < 1 minute):
   ```
   $ git clone https://github.com/pag-crypto/zkmbs.git
   $ cd zkmbs/
   ```
2. Install jsnark (a library used by xJsnark) and its dependencies by running this script inside `zkmbs/` (time required: 5–10 minutes): `$ ./install_deps_jsnark`
   - If you can't use the script, follow the "jsnark installation instructions" here: https://github.com/akosba/jsnark#prerequisites
   - On some systems, this step may fail when trying to install the dependencies of libsnark as specified in this file: https://github.com/akosba/libsnark/blob/213547311d16644bde7ef806b77dfae25c7f734c/.gitmodules. Please edit all URLs in your local version of the file at `zkmbs/jsnark/libsnark/.gitmodules` (which should be cloned by this point) to use `https` (and not `git`) and try again.
3. Enter `gen/` and compile xJsnark: `$ cd gen/` and `$ ./compile_circuits`. The exact output will depend on the system but it should finish without any errors. On Ubuntu, our output looks like this:
   ```
   Note: Some input files use ...
   Note: Recompile with -Xlint:unchecked ...
   compilation SUCCESS
   ```

D.5 Experiment workflow

After installation, the structure of the main directories should look like this:

```
zkmbs
    +-- gen
    |    +-- circuits
    |    +-- logs
    |    +-- src
    +-- jsnark
```

The experiment scripts will be run inside `gen/`. The Java source code describing the circuits is located in `gen/src/`.

Experiment 1 will generate full circuits from these descriptions and store them in `gen/circuits/`. Experiment 2 will use these circuit descriptions to generate public parameters and measure proving and verification times using sample input files located in `gen/`.

D.6 Evaluation and expected results

The main performance claims in our paper are stated in Figures 6 and 7. There are nine circuits involved in our experiments (the five entries of Figure 6 and the four entries of Figure 7). The first experiment reproduces the "Total" columns of the two tables. The second experiment reproduces the "Time" and "SRS" columns while ensuring that verification time is under 5 ms. We recommend using a

system with at least 32 GB RAM as generating proofs for the largest of our circuits (ChannelBaseline) requires a lot of heap space, and in fact causes errors on systems with just 16 GB memory.

Note that both Figures 6 and 7 list per-subcircuit gate counts that sum to the "Total" count. Our code only allows verifying the gate counts of the entire circuit, as the per-subcircuit counts were approximated by manually inspecting the functions used to build each circuit.

Experiment 1: Reproduce Gate Counts: The aim is to generate circuits from our descriptions and reproduce the total gate counts of each circuit (the **Total** columns of the two tables). This experiment can be repeated by running the script `./reproduce_total_counts` (time required: 20 minutes) in the `gen/` directory. The script outputs into file `column_total.txt`, which should look like this after the script finishes:

```
ChannelBaseline              747.9   # BCO
ChannelShortcut              111.1   # SCO
Channel0RTT                   60.7   # ECO
ChannelAmortized              19.1   # ACO^AES
ChannelAmortized_ChaCha        8.7   # ACO^Cha
Firewall_HS                  150.1   # Firewall
DNS_Amortized_ChaCha          17.6   # DoT
DNS_Amortized_doh_get         48.1   # DoH GET
ODOH_Amortized                48.1   # ODoH
```

Note that the "# ..." are added here to map to the abbreviations used in Figures 6 and 7. The numbers obtained should be very close to the ones above with perhaps slight variation coming from the performance of xJsnark's optimizer on different systems. Some of the values shown here are different than that of the "Total" columns in Figures 6 and 7 as those were rounded for presentation.

Experiment 2: Reproduce Times and SRS: The aim is to reproduce the structured reference string sizes (**SRS** columns), proving (**Time** columns) and verification time (always under 5 ms) for each circuit. This experiment can be repeated by running the script `./reproduce_times_srs` (time required: 20 minutes) inside the `gen/` directory.

The script outputs into file `columns_ptime_srs_vtime.txt`, the contents of which after a sample execution are as follows:

```
ChannelBaseline           92.7 s  1179 MB   2.6 ms
ChannelShortcut           15.6 s   148 MB   1.6 ms
Channel0RTT                8.4 s    79 MB   1.6 ms
ChannelAmortized           2.9 s    26 MB   1.7 ms
ChannelAmortized_ChaCha    1.4 s    13 MB   1.6 ms
Firewall_HS               21.2 s   206 MB   1.6 ms
DNS_Amortized_ChaCha       3.1 s    29 MB   2.1 ms
DNS_Amortized_doh_get      6.8 s    72 MB   2.6 ms
ODOH_Amortized             7.9 s    76 MB   2.6 ms
```

Proof generation is a randomized algorithm; the results reported in the paper are the median of five runs. We have observed variations of up to 15% for proving time and 2 ms for verifier time, in either direction. The script above performs just one run per circuit.

D.7 Experiment customization

We provide two additional scripts to reproduce the above benchmarks for an individual circuit: `./generate_circuit DNS_Amortized_ChaCha` and `./prove_and_verify DNS_Amortized_ChaCha`, where "`DNS_Amortized_ChaCha`" can be replaced with any of the nine circuits.

D.8 Notes

Custom Inputs. As the circuit metrics we evaluate (gate counts, parameters sizes and running times) are independent of the actual input used to generate the proofs, input customization isn't required to reproduce our results. The experiments generate valid proofs using fixed input files (`test.txt, test_doh.txt, test_wildcard.txt`) provided in the `gen/` directory. These files contain sample data extracted from a real TLS 1.3 connection and a Merkle tree blocklist of two million entries. We provide instructions in our GitHub repository on generating sample data from new DNS requests and custom Merkle trees.

Editing Circuit Descriptions. Our experiments generate circuits using the Java files in the `gen/src/` directory. These are in turn generated from xJsnark's custom language files that are editable only with an IDE called MPS. To inspect and edit our circuits, we recommend installing the MPS IDE by following the instructions here in our GitHub repository: https://github.com/pag-crypto/zkmbs#installation-instructions-mps.