NOPE: Strengthening domain authentication with succinct proofs *

Zachary DeStefano, Jeff J. Ma, Joseph Bonneau, and Michael Walfish

NYU Department of Computer Science, Courant Institute

Abstract *Server authentication* assures users that they are communicating with a server that genuinely represents a claimed domain. Today, server authentication relies on *certification authorities* (CAs), third parties who sign statements binding public keys to domains. CAs remain a weak spot in Internet security, as any faulty CA can issue a certificate for any domain.

This paper describes the design, implementation, and experimental evaluation of NOPE, a new mechanism for server authentication that uses *succinct proofs* (for example, zero-knowledge proofs) to prove that a DNSSEC chain exists that links a public key to a specified domain. The use of DNSSEC dramatically reduces reliance on CAs, and the small size of the proofs enables compatibility with legacy infrastructure, including TLS servers, certificate formats, and certificate transparency. NOPE proofs add minimal performance overhead to clients, increasing the size of a typical certificate chain by about 10% and requiring just over 1 ms to verify. NOPE's core technical contributions (which generalize beyond NOPE) include efficient techniques for representing parsing and cryptographic operations within succinct proofs, which reduce proof generation time and memory requirements by nearly an order of magnitude.

1 Introduction and motivation

This paper revisits a fundamental requirement of Internet security: *server authentication* gives users assurance that when their browser loads example.com, the response is delivered by the true owner of the example.com domain.

Today, server authentication relies on Certification Authorities, or *CAs*. Offline, the server gets a signed *certificate* from a CA that attests to the binding between the server's public key and its domain. The server sends this certificate to clients, such as web browsers, during the handshake phase of a TLS connection; TLS is most familiar as part of HTTPS, indicated by `https://` in the URL. The TLS client then uses the public key to authenticate the TLS handshake, checking that the server knows the private key. We will refer to this public key as the server's *TLS key*.

Major weaknesses in the status quo have led to a long history of failure [10]. A CA that is buggy [8, 27, 86], compromised [52, 111], or rogue [87, 121] can issue a certificate for an attacker-controlled TLS key. Also, any CA can issue a certificate for any domain, making CAs a "weakest link" system [4]. Moreover, a web browser by default trusts over one hundred CAs, multiplying risk.

Finally, even correctly operating CAs can be deceived. Today, certificate issuance is usually based on *domain validation* (DV). For example, the highest-volume CA, Let's Encrypt [2], uses the ACME DV protocol [15]. In ACME, the requester (a domain owner) presents a TLS key and receives a random challenge from the CA. The requester then has to make that challenge available—either over HTTP at a special URL at the given domain, or else in a DNS record—to demonstrate ownership of the domain. The CA checks for the challenge and, if it's present, issues a certificate with the web server's TLS key. Notice that DV relies on legacy unauthenticated protocols (HTTP and DNS), leaving it vulnerable to rogue hosting providers [65] or attackers on the path between the relevant DNS server and the CA [33, 115].

*This work originally appeared in the proceedings of the 2024 ACM SIGOPS Symposium on Operating System Principles.

Is it possible to reduce or even eliminate trust in CAs without introducing additional trusted parties or infrastructure? We are not the first to ask this question (§9). Among many proposals [31] is a family of protocols originally called DANE [14]. These protocols store TLS keys in DNS records and then authenticate these records with DNSSEC [68]. For our purposes, the most relevant proposal in the DANE family is *DNSSEC-chain-extension* [40], which we will refer to as DCE. With DCE, a web server collects signed statements, which form a chain from the root's public key to the web server's TLS key, and delivers that chain to the client.

DANE protocols replace trust in CAs with trust in DNSSEC. On the one hand, DNSSEC involves only a few trusted entities instead of hundreds of CAs. On the other hand, if cryptographic material in DNSSEC is compromised, there is no way to recover. Ideally, any PKI in this context—based on CAs, DNSSEC, or anything else—should have mechanisms for transparency (by which clients and domain owners learn about issued cryptographic statements) and revocation (by which domain owners cancel wrongly issued statements). Indeed, the status quo has an established ecosystem for these functions [94, 124] (§2.1). But DNSSEC, and hence DANE and DCE, have no such infrastructure.

Design goals. What would server authentication that combines the best of both worlds (legacy and DNSSEC-based) look like? Ideally, it would have the following attributes:

- *No single point of failure.* Domains should not be vulnerable to the compromise of any individual server.
- *Compatibility*. Deployed protocols are difficult to change. Therefore, in contrast to intriguing prior work [17, 39, 43, 80, 84, 133], we prefer solutions that are compatible with deployed protocols (TLS, ACME, DNS, and DNSSEC) and require no new infrastructure. Also, upgraded servers should be able to serve legacy clients; likewise, upgraded clients should be able to connect to legacy servers.
- *Transparency*. Domain owners need to monitor all public keys that can be used for their domain. A natural way to achieve this given the compatibility goal is to reuse Certificate Transparency infrastructure [124] (§2.1).
- *Revocation*. If a cryptographic key is compromised or an incorrect statement is signed, the domain owner should be able to recover quickly. As with Transparency, there is established infrastructure for Revocation [53, 71, 94] (§2.1).
- *Low bandwidth and computational overhead*. This means not adding substantially to the server's and client's communication or computation during the TLS handshake. Adding computation off this critical path is acceptable.

A new model: NOPE. Consider a straw man: domain owners embed a DANE-style signature chain in legacy X.509 certificates, CAs sign those certificates, and web clients check both that signature chain and the legacy certificate. This would avoid a single point of trust, because it retains traditional certificate validation. However, this design would not meet the compatibility goal, as we detail later (§2.2).

Instead, we want to somehow embed in legacy certificates not the DNSSEC signature chain but rather something attesting to the *existence* of such a chain, while barely inflating the certificate size or contents. *Succinct proofs* [56] (§2.3) enable exactly that. These proofs, from complexity theory and cryptography, have seen burgeoning implementations and many variants: SNARKs, PCPs, zero-knowledge (ZK) proofs, interactive proofs, and so on [128, 131]. The setup is that a *prover* convinces a *verifier* that a given computational statement holds, by delivering an encoded proof. The verifier doesn't have to step through the statement. Yet, if the statement doesn't hold, then the verifier rejects the proof.

This paper describes the design, implementation, and experimental evaluation of NOPE (Name Ownership Proved Efficiently). In NOPE, the domain owner gathers a chain of DNSSEC signatures from the root to its own TLS key and creates a succinct proof that such a chain of DNSSEC signatures exists (§3). The domain owner embeds that proof in a classical certificate, which is signed by a CA and delivered to clients

as part of the TLS handshake. NOPE-enabled clients extract and verify the embedded proof. This meets the design goals: there is no single point of trust for the same reason as in the strawman, CAs continue to do what they always have, and the existing infrastructure for transparency applies to the enhanced certificates. Legacy revocation works, though the design must take care to ensure that NOPE proofs are invalidated when their enclosing certificate is revoked.

Instantiation. Succinct proofs can be short and efficient to verify (§2.3). The primary issue in all succinct proof work is prover efficiency. Statements must be proved using a formalism, arithmetic constraints, in which conceptually simple operations are often verbose (§4.1). To lower the proving burden, NOPE introduces new techniques for representing protocol parsing (§4.2-§4.3), big-integer modular arithmetic (§5.1), and elliptic curve operations (§5.2), including ECDSA signature verification (§5.3). These techniques are broadly relevant in other uses of succinct proofs. For example, most ZK proofs that represent cryptographic operations would benefit from NOPE's efficient big-integer arithmetic representation.

We implemented NOPE (§7) as a server-side tool for creating proofs and embedding them (§6) in a certificate issued via ACME. We used this tool to obtain a valid Let's Encrypt-signed certificate for a demo server. We also implemented a Firefox browser extension to extract and verify proofs.

Experimental evaluation (§8) shows minimal performance impact. Verification in NOPE costs around 1.5 ms if run natively, though when implemented in a browser extension in Wasm takes about 35 ms. NOPE reduces proving costs from over 8 minutes naively to under 60 s, and memory costs from nearly 18 GB to under 2 GB. Given that servers need only complete such a proof when a new TLS key is created, this overhead is manageable even for commodity servers. Raw proofs are 128 bytes (inherited from NOPE's underlying proving scheme), or 248 bytes encoded in a certificate, which is expected to add less than 10% of the length of the total certificate chain.

While any change to core Internet infrastructure requires a long process to achieve technical consensus, we believe NOPE is a meaningful first step toward reducing today's high trust requirement in CAs, and an interesting new application for succinct proofs. In the long run, the ideas in this paper could facilitate phasing out CAs altogether (§10).

2 Background

2.1 Server authentication today

TLS, certificates, CAs, ACME. TLS (formerly SSL) is the dominant protocol for establishing Internet channels that are secure—meaning confidential, with integrity, and authenticated. TLS is used in many applications: HTTPS, email (StartTLS), DNS-over-TLS (DoT), and so on.

TLS connections begin with a handshake that typically includes a *certificate* in X.509 format [23], signed by a CA. CAs today mainly perform automated *domain validation* (DV) when issuing a certificate (§1). The largest CA, Let's Encrypt [2], is fully automated and provides free certificates to any domain owner. As noted in the introduction, Let's Encrypt uses ACME for DV [15] (and in fact developed ACME).

Certificate transparency (CT). Arguably the most significant upgrade to the web's PKI in the last decade is Certificate Transparency (CT), which records all certificates in public, append-only logs [124]. These logs have become a crucial tool for administrators to monitor extant certificates for their domains so that they can promptly detect any incorrectly issued certificates and revoke them [53, 71] (see below). Modern browsers (for example, Chrome, Firefox, Safari) accept only certificates that have been logged; meanwhile, the standards [92, 93] require that logs accept only valid X.509 certificates signed by well-known CAs. Thus, CAs are effectively gatekeepers that rate-limit what can enter the logs [88, 92].

The relevant mechanics of CT are as follows. Before a CA issues a certificate, it sends a *precertificate* to various CT logs; the precertificate contains almost all of the information that will be in the final certificate, including the TLS key. The logs respond with *Signed Certificate Timestamps* (SCTs): signatures over

the precertificate and a timestamp, verifiable by the logs' public keys. An SCT is a publicly verifiable promise by a log to include the certificate in that log within a certain time frame, called the *maximum merge delay* (MMD). The CA then includes these SCTs in the final certificate. Browsers typically require that certificates contain a minimum number of SCTs from logs that the browser is configured to trust. Although the purpose of SCTs is to free clients from having to query logs themselves, our work will take advantage of the fact that an SCT also functions as a publicly verifiable approximate time of certificate issuance (§3.2).

Revocation. For various reasons (key compromise, key loss, domain transfer), issued certificates may need to be *revoked*. Two prominent approaches exist [94]: the Online Certificate Status Protocol (OCSP) [53] and certificate revocation lists (CRLs) [71]. Roughly speaking, browsers query OCSP *responders* for a signed, timestamped attestation that a certificate has not been revoked. OCSP responses are typically valid for 3–4 days, limiting the speed of revocation [94, 122]. Explicit querying can be avoided via OCSP stapling, in which the server delivers a current OCSP response in-band with the certificate. With CRLs, by contrast, browser vendors track lists of revoked certificates and publish summaries, which clients may take up to 7 days to poll [94, 122]. We note that OCSP and CRLs are fundamentally built around certificates and, as with CT, rely on CAs to prevent spam.

Proactive vs. reactive security. One might wonder whether CT undermines our motivation: if domain owners and browsers can detect misbehavior, do we really need to defend against the compromise of CAs in a *proactive* fashion? Yes, because the *reactive* nature of CT and revocation leaves a window of exposure. Rogue certificates can be used by attackers before detection. In fact, rogue certificates can be used *after* detection but before revocation, which typically takes 3–7 days, as noted above. Furthermore, given the work required to detect and recover from rogue certificates, it is highly worthwhile to limit such cases.

2.2 DNS, DNSSEC, DANE, and DCE

DNS. DNS maps, or *resolves*, domain names to IP addresses. There is a hard-coded set of DNS *roots*, which provide IP addresses for the authoritative servers for top-level domains (TLDs), such as .com, .org, and so on. The TLDs provide IP addresses for the next-level domains, and so on, until the name server for a domain provides the IP address of a requested server at the domain.

By default, DNS responses are not authenticated, and clients may receive bogus DNS records because of malicious network administrators [66] or cache poisoning [123]. Encrypted DNS (DoH [69], DoT [72], ODoH [81]) is growing and is now the default in Firefox, but it *hides* DNS requests from network observers; it does not *authenticate* the contents of responses. There is also increasing use of public DNS servers (for example, Cloudflare's 1.1.1.1 or Google's 8.8.8.8), but that only moves the point of trust to those organizations.

DNSSEC. A mechanism for authenticating DNS records exists, and is partially deployed: DNSSEC [68]. DNSSEC embeds public keys in the name hierarchy. There is a hard-coded public key for the root; this key (indirectly) signs public keys for TLDs, and so on. Note that this Public Key Infrastructure (PKI) is totally separate from that for TLS.

DNSSEC is a complex protocol. We give a crash course here, focusing only on what is necessary for this paper. To begin, every domain has two public keys: a *Key Signing Key* (KSK) and a *Zone Signing Key* (ZSK). (Actually, domains can have multiple KSKs and ZSKs; for simplicity, we assume a domain has one of each.) Figure 1 depicts the hierarchical relationships among KSKs and ZSKs.

Each domain name in DNS is associated with a *zone file*, which contains *resource records* (RRs) with information about the domain. We are principally concerned with four types of RRs: `TXT`, `DNSKEY`, `RRSIG`, and `DS` [113]. A `TXT` record contains unstructured text data (for example, the challenges used by ACME; §1). A `DNSKEY` (Domain Name System Key) record contains a public key; domains that implement DNSSEC have two `DNSKEY` records, one for the KSK and one for the ZSK.

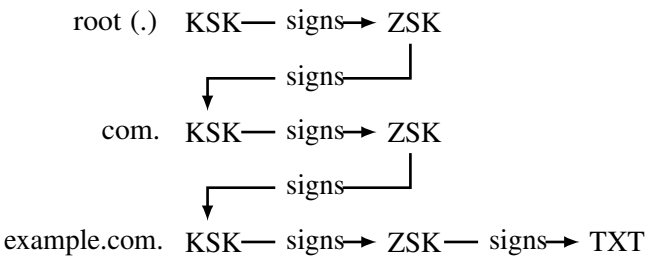

FIGURE 1—Chain of trust, in DNSSEC, from root to example.com, through various public keys (KSKs and ZSKs).

An `RRSIG` (Resource Record Signature) record does not sign a single record (despite its name) but rather a *resource record set* (RRset), which is a collection of records of the same kind. A single RRset can have multiple `RRSIG` records, each signed by a different key; for simplicity, we write as if each RRset is signed by exactly one `RRSIG` record. The `DNSKEY` RRset is signed by the domain's own KSK. All other RRsets for a domain are signed by that domain's ZSK.

A `DS` (Delegation Signer) record, together with the corresponding `RRSIG`, establishes a chain of trust beyond the domain. A `DS` record contains a hash of (specific fields of) a `DNSKEY` record that holds a KSK; this KSK belongs to a child of the domain. This `DS` record, and its accompanying `RRSIG`, mean that a domain's ZSK attests to a child domain's KSK.

As an example, we include below some hypothetical snippets of zone files for example.com and its parent (.com). To be clear, this snippet is made up, and also simplified to remove fields, including TTL, Class, and others:

```
--- example.com. zone file snippet ---
1 example.com. TXT "site-verification=...
2 example.com. TXT "acme-challenge=...
3 example.com. TXT "v=spf1 -all...
4 example.com. RRSIG TXT ...
5 example.com. DNSKEY ZSK AwEAAc...
6 example.com. DNSKEY KSK AwEAAc...
7 example.com. RRSIG DNSKEY ...
--- com. zone file snippet ---
8 example.com. DS 8ACBB0...
9 example.com. RRSIG DS ...
```

Above, there are three `TXT` records (lines 1–3), the set of which is signed (4) by the domain's ZSK (5). There are also two `DNSKEY` records, a ZSK (5) and a KSK (6); this set of two records is signed (7) by the domain's KSK (6). Finally, there is a `DS` record (8), which is signed (9) by the parent domain's ZSK (in .com's zone file).

DANE and DCE. Since DNSSEC authenticates DNS records, why not use it to authenticate the correct TLS key for a domain? This is the idea behind DANE [14] (DNS-Based Authentication of Named Entities).

For completeness, we note that DANE and DNSSEC initially had problems: weak cryptographic parameters (early versions of DNSSEC allowed for MD5 and 512-bit RSA [42], obsolete even at the time); buggy implementations (many pairs of RSA keys across different domains shared common moduli, breaking the security of both [120]); reliability (TLDs regularly experienced outages during key rollover [73]); and bandwidth (DNSSEC records are large; §8).

Those shortcomings have been addressed. For security, the root zone switched from 1024-bit RSA to 2048-bit RSA in mid-2016, 99.6% of RSA KSKs are now at least 2048 bits, ECC is used by a significant fraction of sites, MD5 is no longer used, and SHA-1 is almost entirely unused [37]. Reliability would be addressed by the server's gathering DNSSEC records and delivering them to clients during the TLS

handshake, as is done in DCE [40], discussed earlier (§1). Bandwidth is not ideal but likely tolerable in the web context (although there are applications, like email, where extra kilobytes for each connection would balloon the total bandwidth footprint).

The primary deficiency for our purposes is that none of the DANE family of proposals has an obvious solution for transparency or revocation. Consequently, there is no remediation if a DNSSEC key is compromised or an incorrect record is signed by mistake, or (in analogy with CA compromise) a higher-level DNS server such as the TLD is compromised. A natural question is: why not extend today's CT and revocation (§2.1) to DNSSEC chains?

On transparency, CAs are integral to CT, to prevent log spam (as noted in §2.1). Per RFC 6962, "[current CT] effectively excludes self-signed and DANE-based certificates until some mechanism to control spam ... is found [92]." Thus, to use today's CT with DNSSEC chains, they would have to be embedded in certificates and submitted to CAs to be signed. This would require standardizing an X.509 extension and, more importantly, adoption by CAs. It would also require modifying CT; meanwhile, such changes require careful engineering, as any bug can invalidate a log permanently [9].

Turning to revocation, DNSSEC does not have revocation mechanisms equivalent to the status quo. (The closest thing, RFC5011 [125], is about internal key management, not public dissemination.) While DNSSEC servers could in principle revoke their subdomains' keys, akin to CAs today, this would require a new mechanism to inform browsers of revocations.

A DCE-like approach was trialed in browsers and removed [85, 88]. There now appears to be no momentum toward deployment of DCE or any other use of DANE for TLS, in part because of the transparency and revocation issues.

2.3 Succinct proofs

A *succinct proof* (also known as a *probabilistic proof*) is a cryptographic protocol between a *prover* and a *verifier*, about a *statement*, **S** [56]. **S** can be thought of as a specification of a computation, a program, or a logical assertion. **S** has a public input **X**, local input (or *witness*) **W**, and output **Y**. **X**, **W**, and **Y** are vectors of variables. The prover wants to convince the verifier that a specific $(\mathbf{X}, \mathbf{Y})$ pair, called an *instance*, is *valid* for **S**. Valid means that there exists a **W** such that $\mathbf{S}(\mathbf{X}; \mathbf{W}) = \mathbf{Y}$.

For example, imagine that **X** is the hash digest of a large piece of data, **Y** is 1/0, and the statement is that the data contains the string "HELLO". Then **W** is the data itself, and **S** embeds the logic for: (1) checking if **W**, which is supplied by the prover, contains "HELLO", (2) matching that bit to **Y**, and (3) hashing **W**, and checking if it is equal to **X**. Via a succinct proof protocol, the prover can convince the verifier that the prover knows a witness **W** (this guarantee is known as *knowledge soundness*). If the prover does not know one, or no such witness exists, the verifier will reject the alleged proof (under cryptographic hardness assumptions).

What makes the protocol succinct—and also astonishing—is that the data flowing from prover to verifier, and the work required by the verifier, is much smaller than an unrolled transcript for verifying **S** would be. In particular, the verifier does not ever handle **W**. In fact, some protocols send only a constant amount of data, regardless of the size of **S**. Consequently, the verifier is persuaded of the validity of an $(\mathbf{X}, \mathbf{Y})$ pair while doing far less work than would be needed to re-execute the statement on $(\mathbf{X}, \mathbf{Y})$.

Our focus is on publicly verifiable non-interactive variants, especially ones with a zero-knowledge property. *Publicly verifiable* means that the prover and verifier are not coupled: the prover produces a string π that any verifier can check. *Non-interactive* means that the prover and verifier do not communicate back and forth; instead, the prover produces a proof π that the verifier can check directly, without further communication with the prover. *Zero-knowledge* (ZK) means that **W** is hidden: an adversarial verifier cannot abuse the protocol to learn information about part or all of the witness. A succinct proof that is non-interactive and zero-knowledge is now known as a *zkSNARK* [21, 55] (in the lineage of non-interactive and zero-knowledge proofs, known as *NIZKs* [22, 50]). A succinct proof that is not zero-knowledge is

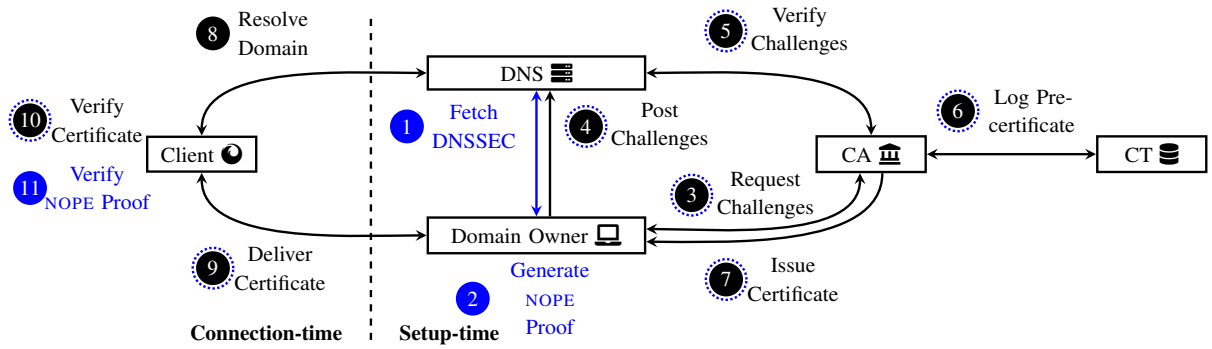

FIGURE 2—NOPE protocol. Domain validation steps are in black, steps introduced by NOPE are in blue, and steps in which the NOPE proof travels around the system are surrounded by a dotted blue circle. ❶ The domain owner fetches the DNSSEC chain for the DS record for their domain. ❷ The domain owner generates a proof connecting their TLS key to the root. ❸ The domain owner requests challenges from the CA, with the proof embedded in a certificate signing request. ❹ The domain owner posts these challenges as DNS records on their domain. ❺ The CA verifies these DNS challenges. ❻ The CA logs the certificate-to-be. ❼ The CA issues the certificate. ❽ A client resolves the domain name to get an IP address. ❾ The client connects to the IP address and receives the certificate. ❿ The client performs standard certificate verification. ⓫ If the client is NOPE-aware, it detects the NOPE proof in the certificate and verifies it.

sometimes called *verifiable computation* [11, 54, 57, 76, 79, 100]. We have presented these definitions informally; precise definitions are available elsewhere [128].

The last 15 years have seen tremendous advances in implementations of succinct proofs, including ZK proofs [128, 131]. Implementations typically have a *front-end* and a *back-end*. The front-end compiles $\mathbf{S}$ into a mathematical representation, specifically a set of *constraints* over a finite field (§4.1). The back-end is the proving and verifying algorithms. The proving algorithm uses the constraints, $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{W}$ to produce the proof π. The verifier uses the constraints, $\mathbf{X}$, $\mathbf{Y}$ to verify π, outputting an accept/reject value.

NOPE's back-end is Groth16, a zkSNARK [55, 60] in which the ZK property adds negligible cost. Among ZK protocols, Groth16 has the shortest proofs and fastest verifier in the literature, which is why we choose it. Specifically, verification costs 1–4 ms on a modern CPU [48], regardless of statement size (actually, there is a cost that scales with the size of $(\mathbf{X}, \mathbf{Y})$, but it is low-order). The size of the proof π is 128 bytes, regardless of statement complexity.

Groth16 has two comparative disadvantages. First, its prover consumes more computation and memory (§4.1, §8) than the best in the literature [35, 36, 58, 116, 118]. However, this is the right trade-off, as servers will compute proofs rarely and not on latency-sensitive paths; verification, by contrast, will be on a client's critical path during page load.

Second, for each statement $\mathbf{S}$, Groth16 requires a one-time setup: statement-specific work that will be reused over all future $(\mathbf{X}, \mathbf{Y})$ pairs. This step must be executed by a trusted party; among other reasons, its output contains a trapdoor that can be used to "prove" even invalid $(\mathbf{X}, \mathbf{Y})$ pairs, and hence must be securely deleted. This requirement is compatible with the present context because DNSSEC already involves a trusted committee (the root key holders) who regularly perform key-signing ceremonies.

3 Design of NOPE

3.1 Threat model, security objectives, core protocol

Threat model. The adversary's goal is to establish a TLS session with a victim client for a domain that the adversary does not legitimately control (a *domain impersonation attack*). This requires either

tampering with traffic that the client sent to the target server (for example, a malicious WiFi network administrator or ISP modifying traffic between the client and server) or redirecting the client's traffic to the attacker's server (for example, using DNS poisoning [123]). Beyond this baseline capability, there are several avenues of attack; NOPE will be designed to tolerate combinations of them but not all of them together.

A *legacy DNS attacker* is able to tamper with DNS resolution between a Certificate Authority and a target domain. The attacker can, therefore, defeat today's DV, obtaining a CA signature on a malicious certificate. This attacker interferes with DNS resolution by DNS poisoning [123] of the target domain's DNS server, or by spoofing or modifying DNS responses on the network. Note that this attack is different than the baseline attacker mentioned above, who modifies network traffic between a *client* and a target domain. A single domain has arbitrarily many network paths to different clients, some of which are surely controlled by attackers at any given time, but only a handful to valid Certificate Authorities. Some CAs also use multi-path probing [3] to make this attack more difficult.

A *CA attacker* is able to directly obtain a valid signature from a CA on arbitrary certificates. For example, the attacker might be a rogue CA, might steal an honest CA's private key, or might exploit a vulnerability to confuse a CA into signing a certificate that it hasn't validated. Note that this attacker is not equivalent to the legacy DNS attacker: while both can obtain a signature on an invalid certificate, the CA attacker can also decline to issue revocation statements (§2.1).

A *DNSSEC attacker* can compromise DNSSEC results for a target domain, obtaining a valid signature on an arbitrary record, for example by stealing DNSSEC keys at the target domain or any DNSSEC server higher in the hierarchy. This attack is not the same as attacking legacy DNS (which can be achieved by a purely network attacker).

A *CT attacker* can obtain SCTs from some CT log (§2.1) on arbitrary certificates (potentially without logging them), by either stealing a CT log's private key or exploiting a vulnerability to confuse it.

Finally, we assume that the attacker cannot break standard cryptographic primitives (for example, forging signatures without knowing a private key), and cannot compromise client machines (a compromised client hurts only itself). In either case, we can make no security guarantees.

Security objectives. Our primary security objective is to prevent domain impersonation by either a DNSSEC attacker (which would succeed against DCE) or a legacy DNS/CA attacker (which would succeed against traditional certificates). Our secondary goal is to enable effective detection and recovery if the attacker does successfully impersonate a domain.

Achieving security incrementally, consistent with compatibility (§1), requires that servers have a secure method of *advertisement*: NOPE clients need to know that a server implements NOPE and that they should not trust non-NOPE certificates. Otherwise, an attacker with a rogue certificate could intercept traffic to a NOPE-enabled server and pretend to clients that the server doesn't support NOPE (and hence launder a rogue, non-NOPE certificate). We discuss mechanisms for advertisement further in Section 6.

Core protocol. Figure 2 depicts NOPE's core protocol in two phases: *setup-time* (not to be confused with the proof system's trusted setup phase; §2.3), where a domain owner interacts with various parties to bind its TLS key to its domain name, and *connection-time*, where the domain owner delivers a proof of the binding to a user agent (a browser).

At setup-time, the owner of a domain D fetches the DNSSEC chain for the DS RRset (§2.2) for D, produces a proof (§2.3), encodes this proof in an ordinary certificate signing request [15, 105], and sends it to an ACME server for signature. The ACME server is oblivious to the proof, and the ACME protocol proceeds as normal, including logging the certificate (§2.1).

At connection-time, once the client resolves a purported IP address for the domain, it initiates a TLS handshake with the server and receives the certificate (and a stapled OCSP response asserting that the certificate has not been revoked recently). The client first validates the certificate in the legacy way,

verifying the CA signature, expiration date, and so on. A NOPE-aware client further extracts the NOPE proof and verifies it, using several fields from the certificate, as public inputs to the proof (§3.2).

Because NOPE proofs are embedded in backwards-compatible X.509 certificates, they can (and must) be logged by existing CT infrastructure to ensure transparency. Existing revocation methods for X.509 certificates also continue to work (but care is required; see details below).

Roadmap. The rest of this section delves into the details of NOPE's proof statement (§3.2) and analyzes NOPE's security (§3.3). Sections 4 and 5 describe techniques for concisely representing NOPE's proof statement in constraints. This conciseness makes costs at NOPE setup-time tolerable (§8).

3.2 Constructing and verifying NOPE proofs

The core statement, $\mathbf{S}_{\text{NOPE}}$, takes as public input:

- a domain name D
- the root ZSK for DNSSEC
- a TLS key T
- the organization name N of the CA
- a truncated timestamp TS

There is no public output. The statement establishes the existence of a valid DNSSEC chain of signatures from the root ZSK to some KSK (§2.2) K for D, and also establishes that the private key corresponding to K, call it K_S, has signed T, N, and TS. The purpose of N and TS is to bind the NOPE proof to a specific enclosing certificate, as explained later in this section. The witness includes K, K_S, and the DNSSEC chain from the root to K.

$\mathbf{S}_{\text{NOPE}}$ comprises two main parts. The first, *DS-knowledge-of-secret* ($\mathbf{S}_{DS.K}$), verifies that (in a sense made more precise below) K_S has signed T, N, and TS, and connects knowledge of K_S to the ZSK of the parent of D. $\mathbf{S}_{DS.K}$ checks four things:

(1) *KSK-knowledge-of-private-key* ($\mathbf{S}_{KSK.K}$): The prover knows K_S, the private key for some public key K.
(2) *KSK-hash* ($\mathbf{S}_{KSK.H}$): K hashes to some value H.
(3) *DS-parse* ($\mathbf{S}_{DS.P}$): H is contained in a `DS` record R for domain D.
(4) *DS-signature* ($\mathbf{S}_{DS.S}$): R has a signature (`RRSIG`) that is validated by a key K' (presumed to be D's parent's ZSK).

These checks reduce the claim that a domain D owns some KSK K to the claim that the parent of D (call it D') owns a given ZSK K'. To establish that D' indeed owns K', $\mathbf{S}_{\text{NOPE}}$ uses its second main part, *ZSK-verify*. *ZSK-verify* reduces a claim that a domain C owns a key ZSK_C to the claim that the parent of C owns a different key, ZSK_P. $\mathbf{S}_{\text{NOPE}}$ applies *ZSK-verify* iteratively, with C initially equal to D'. The iteration ends with a claim about the ZSK of the DNS root; $\mathbf{S}_{\text{NOPE}}$ "directly checks" that last claim, by enforcing equality between the public root ZSK input and the variables in the statement that represent the ZSK of the DNS root.

ZSK-verify ($\mathbf{S}_{ZSK}$) checks five things:

(1) *DNSKEY-parse* ($\mathbf{S}_{DNSKEY.P}$): ZSK_C is contained in a validly formatted `DNSKEY` record on C.
(2) *DNSKEY-signature* ($\mathbf{S}_{DNSKEY.S}$): This `DNSKEY` record has a signature (`RRSIG`) that is validated by some key: KSK_C, presumed to be the KSK of C.
(3) *KSK-hash* ($\mathbf{S}_{KSK.H}$): KSK_C hashes to some value H.
(4) *DS-parse* ($\mathbf{S}_{DS.P}$): H is contained in a `DS` record R for domain C.
(5) *DS-signature* ($\mathbf{S}_{DS.S}$): R has a signature (`RRSIG`) that is validated by a key ZSK_P.

Attacker Subset (§3.1)				Domain Impersonated				Time to Detect				Can be Revoked			
Legacy DNS	CA	CT	DNSSEC	DV	DV+	DCE	NOPE	DV	DV+	DCE	NOPE	DV	DV+	DCE	NOPE
–	–	–	–	No	No	No	No	–	–	–	–	Yes	Yes	No	Yes
☠	–	–	–	Yes	No	No	No	$\leq$ 24h	–	–	–	Yes	Yes	No	Yes
–	☠	–	–	Yes	Yes	No	No	$\leq$ 24h	$\leq$ 24h	–	–	No	No	No	No
☠	☠	–	–	Yes	Yes	No	No	$\leq$ 24h	$\leq$ 24h	–	–	No	No	No	No
–	–	☠	–	No	No	No	No	–	–	–	–	Yes	Yes	No	Yes
☠	–	☠	–	Yes	No	No	No	$>$ 24h	–	–	–	Yes	Yes	No	Yes
–	☠	☠	–	Yes	Yes	No	No	$>$ 24h	$>$ 24h	–	–	No	No	No	No
☠	☠	☠	–	Yes	Yes	No	No	$>$ 24h	$>$ 24h	–	–	No	No	No	No
–	–	–	☠	No	No	Yes	No	–	–	∞	–	Yes	Yes	No	Yes
☠	–	–	☠	Yes	Yes	Yes	Yes	$\leq$ 24h	$\leq$ 24h	∞	$\leq$ 24h	Yes	Yes	No	Yes
–	☠	–	☠	Yes	Yes	Yes	Yes	$\leq$ 24h	$\leq$ 24h	∞	$\leq$ 24h	No	No	No	No
☠	☠	–	☠	Yes	Yes	Yes	Yes	$\leq$ 24h	$\leq$ 24h	∞	$\leq$ 24h	No	No	No	No
–	–	☠	☠	No	No	Yes	No	–	–	∞	–	Yes	Yes	No	Yes
☠	–	☠	☠	Yes	Yes	Yes	Yes	$>$ 24h	$>$ 24h	∞	$>$ 24h	Yes	Yes	No	Yes
–	☠	☠	☠	Yes	Yes	Yes	Yes	$>$ 24h	$>$ 24h	∞	$>$ 24h	No	No	No	No
☠	☠	☠	☠	Yes	Yes	Yes	Yes	$>$ 24h	$>$ 24h	∞	$>$ 24h	No	No	No	No

FIGURE 3—Analysis of subsets of attackers (§3.1). Domain Impersonated indicates whether the attackers can fool a client into believing that they are the domain owner. Time to Detect indicates the time that it takes for evidence of the attack to be visible in transparency logs (∞ indicates that no such evidence is produced). Can be Revoked indicates whether proactive TLS key revocation is possible; without it, the domain owner must, in the event of an attack, wait for the attacker's certificate or signed DNSSEC records to expire.

Digital signatures vs. signatures of knowledge. Notice that in $\mathbf{S}_{DS.K}$ there is no explicit signature of T, N, or TS. In fact, none of these even appear in the logic of the statement! The advantage of this construction is avoiding the need to represent signature verification (in this case, using public key K) in the statement. The validity of the construction is that the proof *itself* [61, 129] is a *signature of knowledge* [30], which can be thought of as a stylized signature, by an entity possessing the entire witness including K_S, of the inputs to the proof, including T, N, and TS.

For this to work, the proof protocol requires zero knowledge (§2.3); otherwise, an adversarial verifier could gain information about the witness, including K_S. A property called *weak simulation extractability* [12] is also required. This ensures that an attacker cannot take a valid proof, constructed from T, N, and TS, and change it to a new (and "valid") proof for a different T', N', and TS'. The back-end of NOPE, Groth16 [60], is zero-knowledge by design, and NOPE applies a straightforward modification to gain weak simulation extractability [12, 104]. Weak simulation extractability still allows an attacker to take a valid proof π for T, N, and TS and generate a new proof π' for the same T, N, and TS. We deal with such *proof malleability* below.

Binding NOPE proofs to certificates. Existing revocation mechanisms work with certificates. Thus, we want each NOPE proof to be bound to a specific certificate—revoking that certificate then revokes the NOPE proof. Ideally, the NOPE proof would directly reference the certificate (for example, by hash or serial number), but this isn't possible: to be embedded in the certificate, the proof must be in the certificate request, so the proof must be materialized first.

This is why N and TS are included in the NOPE proof statement: they commit the NOPE proof to an intended certificate. NOPE verification rejects proofs in which N and TS do not match those of the enclosing certificate. A compromised CA could later issue a certificate with an earlier timestamp (to match a prior NOPE proof), but it would significantly differ from the CT-controlled SCTs in the certificate (§2.1). Thus, NOPE clients must also check that the SCT timestamp (approximately) matches the certificate's timestamp. Note that a detected inconsistency could be broadcast as irrefutable evidence of misbehavior.

Similarly, a compromised CA could, on receiving a valid certificate signing request with a new NOPE proof, attempt to immediately issue multiple certificates with the same proof (or superficial modifications

via proof malleability). Recovery would require multiple revocations and would be observable in CT logs as evidence of misbehavior.

A challenge is that the prover cannot predict the precise issuance time of the certificate, owing to variable latency in domain validation by the CA. To address this, the domain owner truncates *TS* to within a few minutes.

Verifying a NOPE proof. The client runs the back-end verifier against the proof π (extracted from the certificate), the proof statement $\mathbf{S}_{\text{NOPE}}$, and the input to the proof statement. This input is the root ZSK, the domain name D, the TLS key T, canonical name N of the CA, and time TS. The client extracts the latter three from the certificate itself; the client performs the same truncation of TS as described above.

NOPE-managed. So far, NOPE has assumed that the domain owner knows the private key for their KSK. But some domain owners outsource the administration of their DNSSEC keys to a managed DNS provider. A variant called NOPE-managed handles this case; it is described in Appendix A.

3.3 Analysis

We compare NOPE to DV and DCE, under various attackers (§3.1). We also compare to domain validation bolstered by requiring DNSSEC proofs in addition to legacy DNS queries, which we call DV+. Figure 3 summarizes.

No single point of failure. NOPE can be thought of as a "belt-and-suspenders" approach, with clients performing two independent validity checks on a server's TLS key. Achieving domain impersonation under NOPE requires *both* a fraudulent certificate (which a legacy DNS attacker or CA attacker is capable of) and obtaining fraudulent DNSSEC records for the NOPE proof (which a DNSSEC attacker is capable of). This is a strictly better security guarantee than DV (which is vulnerable to a legacy DNS/CA attacker), DV+ (vulnerable to a CA attacker), or DCE (vulnerable to a DNSSEC attacker).

One might ask if the same outcome could be achieved by having two different CAs sign a certificate. The main issue (beyond consolidations in the CA industry that sometimes obscure common operations of ostensibly distinct CAs [7, 41]) is that nearly all CAs employ similar DNS-based DV, so any attacker that could fool one could very likely fool two.

Detection and recovery. Even in the event of a successful domain impersonation attack, NOPE provides transparency and revocation equivalent to the status quo (§1). This is strictly better than DCE, which sacrifices both.

For transparency, NOPE proofs must be included in a certificate included in a CT log. In the absence of a CT attacker, this ensures any domain impersonation will be detectable within the MMD (§2.1) of 24 hours (equivalent for DV, DV+, and NOPE). If domain impersonation is combined with a CT attacker—if, that is, all or nearly all attacker types are in effect, with the CT attacker issuing an SCT but neglecting to log the rogue certificate—then there is no guarantee of detection for any of the systems. (A fallback in this case is a victim client performing SCT auditing [99], but web browsers do not do so by default today.)

DV, DV+, and NOPE enable revocation unless the certificate-issuing CA is compromised; that CA can refuse to issue revocation statements. Blocking revocation thus requires the attacker to compromise a specific CA, not an arbitrary one.

4 Representing DNS parsing efficiently

This section describes techniques for efficiently representing parsing operations in proof statements. This parsing occurs in $\mathbf{S}_{DNSKEY.P}$ (for `DNSKEY` records) and $\mathbf{S}_{DS.P}$ (for `DS` records), but we expect the techniques to apply to other uses of succinct proofs. Next, we describe the compilation target.

4.1 R1CS

Various proof back-ends (§2.3), including the one that NOPE uses (Groth16 [60]), require statements to be compiled, by the front-end, to a *rank-one constraint system* (R1CS), which we sometimes refer to as *constraints*. Our description of R1CS here borrows heavily from Zombie [134]. An R1CS instance is a system of equations (over a finite field, $\mathbb{F}$). An instance has m equations (or constraints), n variables, and matrices A, B, C, each of dimension $m \times n$. We say that a specific $(\mathbf{X}, \mathbf{Y})$ pair (§2.3) *satisfies* the instance if there exists some $\mathbf{W}$ such that, for z defined as $(\mathbf{X}, \mathbf{Y}, 1, \mathbf{W})$, we have: $Az \circ Bz = Cz$, with $\circ$ denoting entry-wise multiplication. Unpacking the algebra, each constraint $i \in \{1, \ldots, m\}$ restricts any satisfying $z = (z_1, \ldots, z_n)$ as follows: $(A_{i,1}z_1 + \cdots + A_{i,n}z_n) \cdot (B_{i,1}z_1 + \cdots + B_{i,n}z_n) = (C_{i,1}z_1 + \cdots + C_{i,n}z_n)$. The proof π then convinces the verifier that $(\mathbf{X}, \mathbf{Y})$ satisfies the R1CS instance and that the prover knows $\mathbf{W}$.

The primary metric for the efficiency of an R1CS representation is *number of constraints*, m; that is because the prover's costs scale with $m \cdot \log m$ (from manipulating degree-m polynomials) and the constant is large (§8.2). Proof verification and proof size were quantified earlier (§2.3).

Unfortunately, R1CS is not a natural way to express computation. Loops must be unrolled; each iteration compiles to separate variables in the constraints. Representing conditional flow requires separate constraints for each branch. Comparisons and bitwise operations are expensive. More generally, every statement in a program has to be arithmetized, sometimes requiring further variables [25, 26, 110, 117, 119, 130, 136]. Emulating RAM also brings cost [19, 26, 82, 109, 130]; this applies to any situation in which the index into an array isn't known at compile time.

4.2 Framework and motivation

For our purposes, parsing means taking a long message, allegedly in a specific format, and extracting a given field from a location. Neither the length of the field nor the location is known at compile-time.

Consider, for example, what has to happen in NOPE's proof statements. Recall that $\mathbf{S}_{DNSKEY.P}$ involves a claimed `DNSKEY` RRset for a domain C and a key ZSK_C (§3.2). The corresponding constraints have to extract the `type covered` field to check that the claimed RRset is really a `DNSKEY` RRset. They also have to extract the `rrname` and `key` fields from an RR that is itself extracted from the RRset, to ensure that the RR is for the claimed domain C and that ZSK_C is contained in the RRset. Similarly, $\mathbf{S}_{DS.P}$ must validate the contents of various fields, and check that the claimed `DS` RRset for a given domain indeed is a `DS` RRset, indeed is for the claimed domain, and indeed contains the claimed hash.

But it is not a priori clear how to encode these tasks in R1CS efficiently. DNS parsing seemingly calls for control flow and RAM, and naively translates to an unpalatably large number of constraints (§4.1). Instead, NOPE creates a general procedure for expressing parsing operations in R1CS:

(1) *scan* performs a linear pass over a string *msg* to identify the start of fields of a given type; it returns the start of an arbitrarily chosen field of that type. No such prior primitive that targets constraints exists in the literature. *scan* costs only 4 constraints per-byte of the string, given length-prefixed formats.

(2) *slice* takes a string *msg* and a starting point i, and returns a substring (of statically-known length) starting at i. NOPE's *slice* improves on existing slice primitives [13, 130] (for example, quadratic to quasi-linear or linear, with a smaller constant).

(3) *mask* takes a string and, starting at a dynamically-given location, zeroes out all of the bytes. NOPE's *mask* improves on existing mask primitives [13, 34] by a logarithmic factor in the length of the input.

These primitives are intended to be composed: by applying *scan*, then *slice*, then *mask*, the constraints themselves can parse out a field of a desired type, as well as perform the other tasks listed in the motivating example. To communicate a flavor of the primitives, we present the details for *mask*, the simplest of the three. Appendix B details *slice* and *scan*.

4.3 Mask primitive

The signature of *mask* is *mask*<*L*>(arr, ℓ). It takes an array of length L and returns an array of the same length with the bytes beyond index ℓ zeroed. The angle brackets (<>) indicate that L is known at compile-time, which enables loops to be unrolled. A possible implementation is the following:

```
1: procedure maskNaive<L>(arr, ℓ)
2:     for i = 1 to L do
3:         res[i] ← (i ≤ ℓ ? arr[i] : 0)
4:     return res
```

This procedure, compiled to R1CS, costs $L \cdot (2 + \lceil \log L \rceil)$ constraints; the log term comes from the $\leq$ comparison (§4.1).

NOPE instead introduces three lightweight "sub-primitives" and composes them to achieve the same effect. The first is *mapNonZeroToZero(x)*: if x is non-zero, return 0, and if x is zero, return any value (including possibly zero). This sub-primitive compiles to a single R1CS constraint: $x \cdot z = 0$, where z is the return value. Notice that if x is non-zero, then z must be 0, but if x is 0, then z can be anything.

The second sub-primitive is *indicator*<*L*>(i). Given an index i, *indicator* returns an array of length L with all 0s except for a 1 at index i:

```
1: procedure indicator<L>(i)
2:     sum ← 0
3:     for j = 1 to L do
4:         res[j] ← mapNonZeroToZero(j − i)
5:         sum += res[j]
6:     constrain sum == 1
7:     return res
```

This works because only at $j = i$ can *mapNonZeroToZero* return a non-zero value, and that value is forced to 1 by line 6, which translates to a single constraint that enforces the equality. The total cost of *indicator* is $L + 1$ constraints: one for each *mapNonZeroToZero* and one for line 6.

The third sub-primitive is *suffixSum*<*L*>(arr). Given an array *arr* of length L, return an array *res* of length L with $res[i]$ equal to the sum of $arr[j]$ for all j greater than or equal to i. It is implemented as follows:

```
1: procedure suffixSum<L>(arr)
2:     sum ← 0
3:     for i = L down to 1 do
4:         sum += arr[i]
5:         res[i] ← sum
6:     return sum
```

Perhaps surprisingly, this sub-primitive costs no constraints! To see why, imagine a constraint variable z^* that is supposed to be equal to $\sum_{k=1}^{K} d_k z_k$, where the d_k are constants known at compile time, and the z_k are variables within the constraints. Naively this could be expressed with a constraint in R1CS format: $z^* = (1) \cdot (d_1 z_1 + \cdots)$. But the compiler can replace any occurrence of z^* anywhere else in the R1CS instance (in the "A", "B", or "C" part of a constraint) with the linear combination; doing so only changes the coefficients of the z_k in the constraint(s) where the substitution takes place, while preserving R1CS form. The general point is that linear combinations do not cost constraints. Meanwhile, above, for each i, $res[i]$ is a linear combination of elements in *arr*.

mask<*L*>(arr, ℓ) is now implemented as follows:

```
1: procedure mask<L>(arr, ℓ)
2:     h ← suffixSum<L>(indicator<L>(ℓ))
3:     for i = 1 to L do
4:         res[i] ← arr[i] · h[i]
5:     return res
```

Notice that h has the form $(1,1,\ldots,1,0,0,\ldots,0)$; the component-wise multiplication of *arr* and h then ensures that $res[i]$ contains what it would in *maskNaive*. The total cost of *mask* is $2 \cdot L + 1$ constraints, an asymptotic and concrete improvement over *maskNaive*.

5 Representing cryptography efficiently

With NOPE's parsing techniques (§4) in place, *signature verification* contributes the majority of the constraints in the compiled R1CS representation (§4.1). Signature verification occurs in $\mathbf{S}_{DS.S}$ and $\mathbf{S}_{DNSKEY.S}$ (§3.2). This section describes three sets of techniques for reducing those costs.

As context, DNSSEC uses the RSA [77] and ECDSA [70] cryptosystems; the most common parameter choices (RSA with SHA-256 and ECDSA with curve P-256 [102]) account for 96% of all TLDs (of the 1513 TLDs in the root zone, 1331 use RSA and 129 use ECDSA) [32], and over 99.9% of all domains globally [38]. ECDSA uses ECC (elliptic curve cryptography). ECC has traditionally been costly in constraints because of the complexity of elliptic curve operations and the difficulty of performing big-integer arithmetic in constraints.

The first set of techniques (§5.1) applies to any cryptosystem with modular arithmetic of large numbers, including RSA and ECDSA. The second set of techniques (§5.2) applies to elliptic curve (EC) cryptosystems generally; the third (§5.3) applies to a subset of EC cryptosystems, including ECDSA. The latter techniques lower the cost of ECDSA to be within a small constant multiple of RSA (§8.3).

5.1 Big-integer arithmetic

Cryptography often deals with big numbers. Typically, software represents such numbers as vectors $x = (x[0],\ldots,x[N])$. In this section we write such vectors as row vectors, 0-indexed. Each component of x is a digit in a base-b representation. A common value for b is $b = 2^{32}$, which allows each component to fit comfortably in a 64-bit machine word. (It may be helpful to visualize b as something much smaller, like 10; then the vector is the digits of the base-10 representation, in reverse order.) The numerical value of x is $val(x) = \sum_{i=0}^{N} x[i] \cdot b^i$. Each component of this vector representation is called a *limb*.

Cryptographic operations are often over a field $\mathbb{F}_q$, so arithmetic is performed modulo a prime q In this case, x must be interpreted as a member of $\mathbb{F}_q$: $mval_q(x) = \sum_{i=0}^{N} x[i] \cdot b^i \bmod q$. Naively, arithmetic in $\mathbb{F}_q$, known as modular operations, require division by q to compute remainders. There are improvements [16, 89, 101], but they are still far more expensive than non-modular operations.

In constraints, a large number is a vector of limbs, with each component being an element in $\mathbb{F}$ (the field that the constraints are defined over; §4.1). Usually $\mathbb{F}$ is different from $\mathbb{F}_q$, the field for the actual computation. In the state of the art [82], the cost of a mod operation scales with the number of *bits* in q, unlike multiplication, which scales with the number of *limbs* (but can result in intermediate values larger than q). The distinction is an order of magnitude: whereas q typically has 256 bits with intermediate computations commonly on 512- or 768-bit numbers, $N = 16$ and $N = 24$ respectively assuming the base $b = 2^{32}$.

NOPE's approach is to explicitly allow intermediate values in $\mathbb{F}_q$ to have an unorthodox representation. For $i = 0,\ldots,N$, choose m_i as a vector with S limbs so that $val(m_i) \equiv b^i \bmod q$. Notice that although b^i can be far larger than q, m_i can be chosen so that $val(m_i) < q$. Next, stack these row vectors: $m_0, m_1, \ldots, m_N$. Call the resulting $(N+1) \times S$ matrix M. It follows from the definitions that $mval_q(x \cdot M) = mval_q(x)$. So,

instead of using a traditional mod operation to reduce the size of x from $N+1$ to S limbs, NOPE uses vector-matrix multiplication with M. As with the traditional mod operation, multiplying by M results in a vector with S limbs and preserves the equivalence class of the original number (when interpreted mod q). Unlike the traditional mod operation, multiplying by M does not add any constraints. That is because M is pre-computed, vector-matrix multiplication (with a constant matrix) becomes a collection of linear combinations, and linear combinations contribute no constraints (§4.3).

Here is a worked example, using ordinary arithmetic. Take $b = 10$ and $q = 89$. One can precompute the rows of M by taking the limb representations of $10^i \bmod 89$ for $i = 0, \ldots, 4$ and stacking them to get the following.

$$\begin{aligned} 01 &\equiv 10^0 \bmod 89 \\ 10 &\equiv 10^1 \bmod 89 \\ 11 &\equiv 10^2 \bmod 89 \\ 21 &\equiv 10^3 \bmod 89 \\ 32 &\equiv 10^4 \bmod 89 \end{aligned} \qquad M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \\ 2 & 3 \end{bmatrix}$$

Now, consider the number 51277. In limb form, it is $x = [7,7,2,1,5]$. But $x \cdot M = [20, 26]$. The required equivalence holds: $mval_{89}(x \cdot M) = mval_{89}(x)$. Meanwhile, $x \cdot M$, with 2 limbs, is considerably smaller than x, which has 5 limbs. Notice that $x \cdot M$ and x are not equivalent in "val" terms: $val(x \cdot M)$ is 280, not 51277. Also notice that padding $x \cdot M$ with zeros and multiplying by M again is idempotent.

This approach yields massive savings versus the traditional mod in constraints: whereas mod, recall, has costs proportional to the number of bits in the modulus q (typically 256–768 bits for ECDSA, 2048–4096 bits for RSA), this approach costs no constraints. The gain dramatically outweighs the cost, which is that the number of bits in each limb do not reduce, and thus the constraints require a periodic "clean" operation to prevent limbs from overflowing in $\mathbb{F}$.

5.2 Faster elliptic curve operations

Background on elliptic curves. See elsewhere [127] for a proper treatment. For our purposes, an elliptic curve (EC) is a special "point at infinity," denoted O, together with all points (x,y) over a finite field $\mathbb{F}_q$ (the integers modulo a prime q) that satisfy $y^2 = x^3 + ax + b \bmod q$, for some fixed $a, b \in \mathbb{F}_q$. We sometimes write an EC point $\mathcal{P}$ as $(x_\mathcal{P}, y_\mathcal{P})$.

An EC is equipped with a group operation, written as "+". The additive identity is O; the additive inverse of (x,y) is $(x,-y)$. We describe this operation for the case that neither addend is the identity and the addends are not inverses. If $\mathcal{P} \neq \mathcal{Q}$, the sum $\mathcal{P} + \mathcal{Q} \to \mathcal{R}$ has a geometric interpretation: draw a line between $\mathcal{P}$ and $\mathcal{Q}$, find the point (call it $-\mathcal{R}$) where that line intersects the curve, and then reflect it over the x-axis to get $\mathcal{R}$. This operation is called *point addition*. When $\mathcal{P} = \mathcal{Q}$, the geometric interpretation of $\mathcal{P} + \mathcal{P} \to \mathcal{R}$ is: draw a line tangent to the curve at $\mathcal{P}$, find the point $-\mathcal{R}$ where that line intersects the curve and reflect it over the x-axis to get $\mathcal{R}$. This operation is called *point doubling*. Given a scalar k and a point $\mathcal{P}$, we write $\mathcal{P}$ added to itself k times as $k \cdot \mathcal{P}$; this operation is called *scalar multiplication*.

Given w (scalar, point) pairs $(k_i, \mathcal{P}_i)$, computing $\sum_{i=1}^{w} k_i \mathcal{P}_i$ is called a *multi-scalar multiplication (MSM)*. This operation frequently arises in EC cryptosystems. Even with existing techniques for computing MSM efficiently, MSM induces many point additions and doublings. For example, ECDSA signature verification (§5.3) requires an MSM with 2 points and 256-bit scalars and can induce over 500 point operations, each of which involves multiple operations over $\mathbb{F}_q$.

Point addition and point doubling can be computed algebraically. For example, for point addition:

$$\begin{aligned} s &\equiv (y_Q - y_P) \cdot (x_Q - x_P)^{-1} \bmod q \\ x_R &\equiv s^2 - x_P - x_Q \bmod q \\ y_R &\equiv s \cdot (x_P - x_R) - y_P \bmod q \end{aligned}$$

NOPE's techniques. In constraints, the state of the art representation requires, for point addition, 23 modular multiplications and 2 modular equality checks, and for point doubling, 12 modular multiplications and 2 modular equality checks [1]. This representation rearranges the algebraic formulas for computing addition and doubling, to avoid inversion and cut a single modular equality check.

NOPE's approach is to return to the geometric interpretation. In NOPE, $P + Q$ ($P \neq Q$) requires the prover to supply R to the constraints. The constraints then check that P, Q, and $-R$ are collinear, and that R is actually on the curve. This amounts to two checks:

$$\begin{aligned} (y_Q - y_P)(x_R - x_Q) + (y_R + y_Q)(x_Q - x_P) &\equiv 0 \bmod q \\ y_R^2 - x_R^3 - a \cdot x_R - b &\equiv 0 \bmod q \end{aligned}$$

The cost is 5 multiplications and 2 modular equality checks, down from 23 and 2 above. For point doubling the constraints check that the slope between P and $-R$ is the same as the slope of the curve at P and that R is on the curve, which amounts to replacing the first check above with:

$$(3x_P^2 + a)(x_R - x_P) - 2y_P(y_R + y_P) \equiv 0 \bmod q$$

Point doubling now costs 6 multiplications and 2 modular equality checks, down from 12 and 2.

(We have simplified; the constraints also have to prevent adding inverses together or adding by O.)

5.3 Fewer EC operations in ECDSA

An ECDSA private key is a 256-bit scalar d (providing 128 bits of security). The public key is defined as $Q := d \cdot G$, where G is a generator of the curve. Checking a signature involves checking an operation $R = h_0 \cdot G + h_1 \cdot Q$, where R, h_0 and h_1 are derived from the message and the signature. This operation requires a 256-bit MSM (§5.2). To represent it efficiently, NOPE exploits a transformation to a 128-bit MSM [5, 51]. In non-constraint contexts, this transformation is usually not worthwhile because it involves an expensive computation to identify side information. NOPE observes that it is worthwhile to do that computation outside constraints, with the constraints validating the side information. This saves $2\times$ in constraints for each signature verification. More detail, including several other refinements, is in Appendix C.

6 Protocol considerations

Proof delivery. Satisfying the compatibility goal (§1) requires embedding a NOPE proof within a standard X.509 certificate. CAs do not allow arbitrary data in certificates. X.509 does support extension fields, but IANA would have to approve an extension, and CAs would have to adopt it. Instead, NOPE encodes the proof as a subdomain in the Subject Alternative Name (SAN), which appears to be the only field of a certificate that carries minimally-constrained requester-specified data. This field is typically used to create one certificate that contains related domain names (say, `example.com` and `example.org`). Appendix D details the encoding.

Advertisement. NOPE requires an advertisement mechanism (§3.1). A simple approach is *pinning*, in which clients store a policy that certain domains must use NOPE. Mobile applications often already pin a limited set of trusted CAs [106], and browsers preload CA pins for high-value sites [83]. HSTS [67] is used to dynamically pin an HTTPS-only policy for sites on a trust-on-first-use basis. Any of these could be extended to require NOPE support for a limited period. In the longer term, browsers could sunset support for non-NOPE certificates in several steps, starting with warning about non-NOPE certificates issued after a certain date, and eventually dropping support for non-NOPE certificates.

7 Implementation

Proofs. NOPE's back-end (§2.3) uses two implementations of Groth16 [60], described below. NOPE's front-end (§2.3) uses the Circom [74] language and compiler. The statements in NOPE comprise 3419 lines of Circom code and 176 lines of C++ code that synthesizes additional Circom code.

Server-side. We implemented a tool for domain owners to automatically generate NOPE certificates. This tool is a wrapper around the existing ACME Certbot [45] scripts; it also passes the NOPE proof statements (see above) to zkUtil's [112] interface into the bellman [98] implementation of the Groth16 prover. Given the domain name, a TLS key, and the DNSSEC private key for the domain's KSK, the tool produces a NOPE proof, encodes it in a certificate signing request (§6), and performs the ACME challenge-response protocol with Let's Encrypt (§2.1) to obtain a legacy certificate that embeds a NOPE proof. The tool is 370 lines of Python, 72 lines of JavaScript, and 85 lines of Bash.

Client-side. We implemented an extension for Firefox. We chose Firefox because it is the only current browser that allows extensions to read raw certificate data and override the default certificate verification process; however, this is not fundamental. The extension is triggered on startup (to fetch the root ZSK) and whenever the browser begins a TLS connection. The extension contains a parser to extract NOPE proofs; it also contains a Groth16 verifier in Wasm, originally from SnarkJS [75] and modified by us, to enhance performance. The extension comprises 780 lines of JavaScript.

8 Experimental evaluation

We aim to answer these questions experimentally:

(1) What is the connection-time impact of NOPE on upgraded and non-upgraded clients and servers?
(2) What are the costs of NOPE certificate issuance? And how do they compare to the setup-time costs associated with unmodified ACME?
(3) How effective are NOPE's techniques (§4–§5) at reducing the cost of NOPE certificate issuance?
(4) How does the encoded NOPE proof compare in size to other components of a TLS certificate?

Experimental setup. We use a proof statement for a second-level domain; pessimistically all DNSSEC keys are ECDSA (since it is costlier; §8.3) except the root's ZSK, which is always RSA. As the domain owner of `nope-tools.org`, we create a proof that binds this domain to its TLS key.

Client-side, we measure the NOPE extension (§7) and a native implementation of Groth16 verification [24]. Baselines are JavaScript (JS) versions of legacy certificate validation and DCE [40] (§1, §2), and native implementations of the required cryptography. The JS baselines use the Node.js crypto module with a library that we wrote. The crypto module is equivalent to the Web Crypto API in the browser, and accelerates certain operations using native code. Our library performs certificate validation, and we wrote it because existing Node.js packages with equivalent function perform cryptography only in JS; by contrast, ours invokes native functions, which is more generous to the status quo. For native, we estimate the non-NOPE systems by microbenchmarking the requisite cryptographic operations using OpenSSL [107] and

Server	Client	Bandwidth	time (JS)	time (native)
Legacy	Legacy	2554 B	0.3 ($\pm$ 0.1) ms	*0.3* ms
Legacy	NOPE	2554 B	0.3 ($\pm$ 0.1) ms	*0.3* ms
NOPE	Legacy	2783 B	0.3 ($\pm$ 0.1) ms	*0.3* ms
NOPE	NOPE	2783 B	34.9 ($\pm$ 2.2) ms	*1.5* ms
DCE	DCE	5-6 KB	1.1 ($\pm$ 0.2) ms	*0.7* ms

FIGURE 4—Client-side costs to verify a server's authenticity. The NOPE client (native) adds tolerable overhead. The NOPE client (JS) suffers from a lack of native support for the cryptographic operations that NOPE depends on. Italicized values are estimates derived from isolated native costs for various cryptographic operations. The NOPE-legacy configurations show that NOPE does not add material overhead when the counterparty is not NOPE-aware.

counting the number of such operations. All benchmarks are run on a Google Cloud [59] e2-highmem-2 machine on a single thread with 16 GB of RAM.

8.1 Impact on client latency in TLS handshake

We consider five (server, client) configurations: {legacy server, NOPE server} $\times$ {legacy client, NOPE client} plus DCE server with DCE client. We consider JS and native client versions.

We start the clock when the client has the certificate or signature chain in hand, and stop the clock when the client is convinced that the server is authentic. For each configuration, we measure this verification time 10,000 times, reporting the average and standard deviation (displayed in $\pm$) after removing 1% of outliers. Figure 4 depicts the results.

The JS implementation of the NOPE client performs the worst, because there is no native support for the cryptographic operations in the Groth16 verifier (§7). Given a native implementation, NOPE would contribute roughly 1.2 ms (1.5 − 0.3) above the existing costs for certificate validation.

The JS and native implementations of legacy certificate checking are very close, because the JS extensions leverage browsers' built-in capabilities for handling classical certificates. Similarly, the DCE JS and native versions are on the same order, because, even from JS, the cryptographic operations that are required by DCE have native support.

8.2 Costs of NOPE certificate issuance

Are the costs of certificate issuance reasonable for a low-resource domain owner? We compare NOPE to ACME, measuring the time that a server needs to get a valid certificate from Let's Encrypt (LE) for `nope-tools.org`. Rate-limited by LE, we run five trials of the ACME protocol (DNS01 challenge), and report the average time. We configure NOPE to use a single thread, and measure the proof generation time, taking the average of 100 iterations. For all of the benchmarks, we assume that updating DNS records has no cost, but that these records take 30 seconds to propagate (the default propagation delay for Certbot [45]).

Figure 5 depicts the results. NOPE proof generation is the most computationally intensive step, requiring 35–55 seconds on a single thread and just under 2 GB of RAM. This is obviously dramatically larger than the required computation under ACME. The additional latency from NOPE is about 3$\times$ as long as ACME, and could be lowered by configuring the NOPE prover for parallelism. However, a typical domain will need to pay this cost only 4 times a year.

8.3 Effect of NOPE's constraint representations

We evaluate the impact of the design choices in Section 3 and the techniques in Section 4 and 5. We focus on m, the number of constraints (§4.1) to represent the proof statement. We begin by approximating the number of constraints needed to encode a statement that associates a TLS key with a DNSSEC chain, using the best-known techniques prior to NOPE. Then we introduce NOPE's techniques. For each number

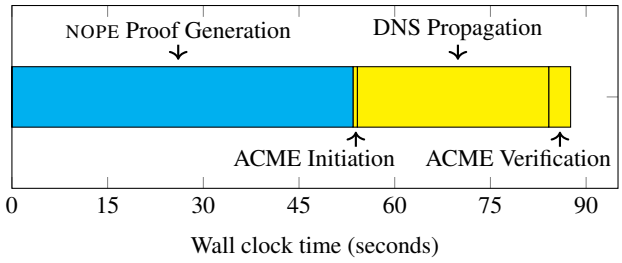

FIGURE 5—Timeline of the NOPE protocol (ACME included). Bars are labeled for each major step in the protocol. NOPE *Proof Generation* is 1–2 from Figure 2, *ACME Initiation* is 3, *DNS Propagation* is the time between 4 and 5, and *ACME Verification* is 6.

Techniques	m ($\times 10^6$)	time	memory
Baseline (§8.3)	*10.15*	*486* s	*17.80* GB
+ design (§3)	*5.33*	*255* s	*9.35* GB
+ parsing (§4)	*3.60*	*173* s	*6.32* GB
+ crypto (§5)	*1.19*	*57* s	*2.09* GB
+ misc.	1.13	54 s	1.99 GB

FIGURE 6—Effects of NOPE's techniques on the constraint representation (m is the number of constraints; §4.1) and costs to generate a proof. Italicized numbers indicate estimates using a mix of calculated and measured values.

of constraints, we use an experimentally derived model relating m to real performance, to estimate the time and memory for the server to generate a proof. Figure 6 depicts the results.

In total, NOPE's techniques lower the cost of proof generation by nearly 9×. It is not directly depicted, but our techniques (§5.1–§5.3) reduce the cost of ECDSA by about 4.5×, taking it from nearly 17× more expensive than RSA to only 3–4×. Additionally, NOPE's parsing costs become negligible compared to the cryptographic operations.

8.4 NOPE in the certificate chain

Using `asn1parse` [108], we measure the communication overhead of NOPE's proof (both raw and encoded in the certificate), comparing to legacy certificates and a full DNSSEC chain, as transmitted according to the DCE specification [40]. Figure 7 depicts the results. The NOPE proof is comparable in size to other certificate components.

8.5 Summary

NOPE has a performance cost, but it is tolerable. Using the extension, clients need ∼35 ms of processing time to verify the certificate and contained NOPE proof (when present) in JS. A native implementation would reduce this number by roughly 23× (§8.1). The domain owner requires orders of magnitude more computation to create a NOPE proof than to participate in ACME, but the absolute cost is well within the capabilities of lower-end machines (§8.2), after applying NOPE's techniques (§8.3). Furthermore, this cost is required only when the server requests a new certificate, which is typically not latency-sensitive. Compared to DCE, NOPE imposes less bandwidth but more computation; the latter is the price of compatibility with existing infrastructure. A final point of comparison is additional trusted modules. NOPE trusts that the proof compiler, proof statements, and verifier implementation are correct while DCE adds DNSSEC parsing and validation code to the browser.

Component	Actual Bytes	Percentage
Certificate Chain	2554	100.0%
Intermediate Certificate	1290	50.5%
Subscriber Certificate	1264	49.5%
Certificate metadata	174	6.8%
Subject name	14	0.5%
Subject public key	294	11.5%
Extensions	534	20.9%
OCSP (§2.1)	89	3.5%
SCT (§2.1)	264	10.3%
Other	181	7.1%
Signature	248	9.7%
Raw NOPE proof (§2.3)	**128**	**5.0%**
Encoded NOPE proof (§6)	**248**	**9.7%**
DCE (§1, §2.2) [40]	5870	229.8%

FIGURE 7—Decomposition of a certificate chain for NOPE issued by Let's Encrypt for domain `nope-tools.org`. OCSP refers to a revocation-related field (specifically for the `authority information access extension` [23]) and does not include the size of the OCSP response if present.

9 Other related work

CA replacement proposals. Security researchers have attempted to bolster or replace the CA model almost continuously since the development of HTTPS [31, 47].

One line of work is *reactive*, aiming to detect rogue certificates after the fact. Early efforts included client-side multi-path probing [18, 96, 132] and crowd-sourced certificate scanning [44]. This led to a line of work on transparency, including Certificate Transparency (CT) [92, 93] (§2.1), Sovereign Keys [43], AKI [80], ARPKI [17], and Revocation Transparency [91]. Of these, only CT has real-world traction.

The other line of work is *proactive*, aiming to prevent rogue certificates by retaining CAs but restricting the set of CAs that can issue certificates for a given domain, including CAge [78], TACK [97] and key-pinning (HPKP [49]). These approaches (unlike NOPE, DANE, and DCE) risk "domain bricking", in which a mistaken overly-restrictive policy prevents a site from being accessible. CAge and TACK were never deployed. HPKP was deployed in Chrome in 2013, but limited uptake [83] and frequent domain bricking [64] caused support to be dropped by 2018. (Chrome and Mozilla still include a small number of preloaded key pins.)

Succinct proofs in networked systems. NOPE works in an emerging tradition of applying succinct proofs to legacy network protocols to improve security or enable new functionality. Like NOPE, these works must parse network payloads and encode "legacy" cryptographic operations that have not been designed to be constraint-friendly.

Cinderella [34] is perhaps most related to NOPE. Cinderella uses zero-knowledge proofs to free clients from having to receive a full X.509 certificate chain. In Cinderella, the server proves that a valid chain exists, saving bandwidth and computation, and opening up new use cases like anonymous credentials. Cinderella and NOPE could be combined, by enhancing the proof statement to show, for example, the existence of an X.509 certificate that contains a NOPE proof. In representing certificate parsing in R1CS, Cinderella faces an analogous problem to one of NOPE's (§4). Cinderella *merges* instead of *parses* (these are inverses), but its approach is currently quadratic; NOPE's techniques would make this linear.

Works in the ZKMB paradigm [62, 95, 134] prove that encrypted traffic complies with network usage policy. The performance requirements are different: they must compute proofs in near real-time whereas NOPE's proofs are computed infrequently (§3).

DECO [135], DiStefano [28], DIDO [29] and Janus [90] use proofs to provide transferable attestations

from interactions with a TLS server. Several other lines of work aim to build non-interactive or anonymous credentials out of existing protocols, including SMTP (email) [63], Open ID Connect [6, 13, 103], JSON [137], and e-Passports [114]. Many of these works face analogous string manipulation challenges to NOPE's. As examples, string manipulation primitives in zkLogin [13] are quadratic, and IDEA-DAC [137] achieves linear complexity but requires instantiating a random oracle in R1CS. NOPE's techniques would improve the concrete performance of these works by several orders of magnitude.

10 Concluding discussion

Putting aside the challenges of compatibility, transparency, and revocation (§1), one could potentially use the techniques developed for NOPE to wean the web off of CAs completely. While it would mean reverting to a new single point of trust, DNSSEC instead of CAs, there are advantages to doing so: DNS is already essential for the Internet, and DNSSEC trust is hierarchical (no single server beyond the carefully guarded root has authority over all domains). Indeed, we hope our work might revitalize interest in DNSSEC.

In the end, building consensus to deploy any security protocol that affects the entire Internet is never an easy process, but we offer the NOPE proposal to the community for several reasons: to improve on it, to disseminate techniques that may be of general interest, and to revive discussion about CA replacement in the era of succinct proofs, which open new design points that are worth exploring.

The code for NOPE is available at `https://github.com/PepperSieve/nope`.

Acknowledgements

Kevin Choi contributed to the conception and development of the project. This paper was improved by discussions with, and suggestions by, Benedikt Bünz, Jeremy Clark, Quang Dao, Paul Grubbs, Brad Karp, Aurojit Panda, Justin Thaler, our shepherd Nickolai Zeldovich, and the anonymous reviewers. This work was supported by DARPA under Cooperative Agreement HR00112020022, NSF CNS-2239975, and a gift from Google. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any supporting organization.

References

[1] 0xPARC. circom-ecdsa. `https://github.com/0xPARC/circom-ecdsa`, 2022.

[2] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, Seth Schoen, and Brad Warren. Let's Encrypt: an automated certificate authority to encrypt the entire web. In *ACM CCS*, 2019.

[3] Josh Aas, Daniel McCarney, and Roland Shoemaker. Multi-Perspective Validation Improves Domain Validation Security. Let's Encrypt Blog, Feb 2020. `https://letsencrypt.org/2020/02/19/multi-perspective-validation.html`.

[4] Ross Anderson and Tyler Moore. The Economics of Information Security. *Science*, 314(5799), 2006.

[5] Adrian Antipa, Daniel Brown, Robert Gallant, Rob Lambert, René Struik, and Scott Vanstone. Accelerated verification of ECDSA signatures. In *Selected Areas in Cryptography*, pages 307–318, 02 2005.

[6] Aptos Keyless. Aptos Labs, 2024. `https://aptos.dev/guides/keyless-accounts/`.

[7] Hadi Asghari, Michel Van Eeten, Axel Arnbak, and Nico ANM van Eijk. Security economics in the HTTPS value chain. In *WEIS*, 2013.

[8] Andrew Ayer. Duplicate Signature Key Selection Attack in Let's Encrypt. https://www.agwa.name/blog/post/duplicate_signature_key_selection_attack_in_lets_encrypt, December 2015.

[9] Andrew Ayer. How Certificate Transparency Logs Fail and Why It's OK. https://www.agwa.name/blog/post/how_ct_logs_fail, July 2021.

[10] Andrew Ayer. Timeline of Certificate Authority Failures. https://sslmate.com/resources/certificate_authority_failures, 2024.

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

[12] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of Groth's zk-SNARK. In *Financial Crypto (FC)*, 2021.

[13] Foteini Baldimtsi, Konstantinos Kryptos Chalkias, Yan Ji, Jonas Lindstrøm, Deepak Maram, Ben Riva, Arnab Roy, Mahdi Sedaghat, and Joy Wang. zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials. *arXiv preprint arXiv:2401.11735*, 2024.

[14] Richard Barnes. Use Cases and Requirements for DNS-Based Authentication of Named Entities (DANE). RFC 6394, October 2011.

[15] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, March 2019.

[16] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In *Annual International Cryptology Conference*, 1986.

[17] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *ACM CCS*, 2014.

[18] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, and Kevin RB Butler. Forced perspectives: Evaluating an SSL trust enhancement at scale. In *IMC*, 2014.

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

[20] Daniel J. Bernstein. Pippenger's exponentiation algorithm. 2002.

[21] 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.

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

[23] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[24] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. Consensys/gnark: v0.11.0, September 2024.

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

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

[27] Matthew Bryant. Keeping Positive – Obtaining Arbitrary Wildcard SSL Certificates from Comodo via Dangling Markup Injection. The Hacker Blog, July 2016. https://thehackerblog.com/keeping-positive-obtaining-arbitrary-wildcard-ssl-certificates-from-comodo-via-dangling-markup-injection/.

[28] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. Cryptology ePrint Archive, Paper 2023/1063, 2023. https://eprint.iacr.org/2023/1063.

[29] Kwan Yin Chan, Handong Cui, and Tsz Hon Yuen. DIDO: Data Provenance from Restricted TLS 1.3 Websites. In *IPSEC*, 2023.

[30] Melissa Chase and Anna Lysyanskaya. On Signatures of Knowledge. In *CRYPTO*, 2006.

[31] Jeremy Clark and Paul C Van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Security and Privacy*, 2013.

[32] Cloudflare. ECDSA: The missing piece of DNSSEC. https://www.cloudflare.com/dns/dnssec/ecdsa-and-dnssec/.

[33] Tianxiang Dai, Haya Shulman, and Michael Waidner. Let's Downgrade Let's Encrypt. In *ACM CCS*, 2021.

[34] 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 and Privacy*, 2016.

[35] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023.

[36] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, Paper 2024/504, 2024.

[37] DNSSEC and DANE Deployment Statistics. https://stats.dnssec-tools.org/, 2024.

[38] Domain name registrations in Generic TLDs. https://domainnamestat.com/statistics/tldtype/generic, 2024.

[39] Huayi Duan, Rubén Fischer, Jie Lou, Si Liu, David Basin, and Adrian Perrig. RHINE: Robust and High-performance Internet Naming with E2E Authenticity. In *NSDI*, 2023.

[40] Viktor Dukhovni, Shumon Huque, Willem Toorop, Paul Wouters, and Melinda Shore. TLS DNSSEC Chain Extension. RFC 9102, August 2021.

[41] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *IMC*, 2013.

[42] Donald E. Eastlake. RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS). RFC 3110, May 2001.

[43] Peter Eckersley. Sovereign Keys: A proposal to make HTTPS and email more secure. `www.eff.org/sovereign-keys`, 2011.

[44] Peter Eckersley and Jesse Burns. An Observatory for the SSLiverse. DEFCON, 2010.

[45] Electronic Frontier Foundation. Certbot. `https://github.com/certbot/certbot`, 2024.

[46] ElGamal, Taher. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO*, 1985.

[47] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1), 2000.

[48] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs. Cryptology ePrint Archive, Paper 2023/1503, 2023. `https://eprint.iacr.org/2023/1503`.

[49] Chris Evans, Chris Palmer, and Ryan Sleevi. Public Key Pinning Extension for HTTP. RFC 7469, April 2015.

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

[51] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *CRYPTO*, 2001.

[52] Eva Galperin, Seth Schoen, and Peter Eckersley. A Post Mortem on the Iranian DigiNotar Attack. *EFF DeepLinks Blog*, 2011. `https://www.eff.org/deeplinks/2011/09/post-mortem-iranian-diginotar-attack`.

[53] Slava Galperin, Dr. Carlisle Adams, Michael Myers, Rich Ankney, and Ambarish N. Malpani. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560, June 1999.

[54] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, 2010.

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

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

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

[58] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic SNARKs for R1CS. In *CRYPTO*, 2023.

[59] Google. Google cloud. https://cloud.google.com/compute/docs/general-purpose-machines, 2024.

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

[61] Jens Groth and Mary Maller. Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. In *CRYPTO*, pages 581–612. Springer International Publishing, 2017.

[62] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-Knowledge Middleboxes. In *USENIX Security*, 2022.

[63] Aayush Gupta. ZK Email, 2024. https://blog.aayushg.com/zkemail/.

[64] Scott Helme. I'm giving up on HPKP. https://scotthelme.co.uk/im-giving-up-on-hpkp/, 2017.

[65] Encrypted traffic interception on Hetzner and Linode targeting the largest Russian XMPP (Jabber) messaging service. https://notes.valdikss.org.ru/jabber.ru-mitm/, November 2023.

[66] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. How Great is the Great Firewall? Measuring China's DNS Censorship. In *USENIX Security*, 2021.

[67] Jeff Hodges, Collin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS). RFC 6797, November 2012.

[68] Paul E. Hoffman. DNS Security Extensions (DNSSEC). RFC 9364, February 2023.

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

[70] Paul E. Hoffman and Wouter Wijngaards. Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC. RFC 6605, April 2012.

[71] Russ Housley, Tim Polk, Dr. Warwick S. Ford, and Dave Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459, January 1999.

[72] 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.

[73] Major DNSSEC Outages and Validation Failures. IANIX, March 2024. https://ianix.com/pub/dnssec-outages.html.

[74] iden3. Circom, Circuit Compiler. https://github.com/iden3/circom, 2024.

[75] iden3. snarkjs. https://github.com/iden3/snarkjs, 2024.

[76] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEE Conference on Computational Complexity (CCC)*, 2007.

[77] Jelte Jansen. Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC. RFC 5702, October 2009.

[78] James Kasten, Eric Wustrow, and J Alex Halderman. CAge: Taming certificate authorities by inferring restricted scopes. In *Financial Crypto (FC)*, 2013.

[79] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). May 1992.

[80] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (AKI) a proposal for a public-key validation infrastructure. In *WWW*, 2013.

[81] 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.

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

[83] Michael Kranch and Joseph Bonneau. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[84] Murat Yasin Kubilay, Mehmet Sabir Kiraz, and Hacı Ali Mantar. CertLedger: A new PKI model with Certificate Transparency based on blockchain. *Computers & Security*, 85, 2019.

[85] Adam Langley. DNSSEC authenticated HTTPS in Chrome. Imperial Violet, June 2011. https://www.imperialviolet.org/2011/06/16/dnssecchrome.html.

[86] Adam Langley. Enhancing digital certificate security. Google Security Blog, April 2013. https://security.googleblog.com/2013/01/enhancing-digital-certificate-security.html.

[87] Adam Langley. Maintaining digital certificate security. Google Security Blog, July 2014. https://security.googleblog.com/2015/03/maintaining-digital-certificate-security.html.

[88] Adam Langley. Why not DANE in browsers. Imperial Violet, January 2015. https://www.imperialviolet.org/2015/01/17/notdane.html.

[89] Robin Larrieu. *Fast finite field arithmetic*. PhD thesis, Université Paris-Saclay, 2019.

[90] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for tls 1.3. Cryptology ePrint Archive, Paper 2023/1377, 2023.

[91] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 33, 2012.

[92] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, June 2013.

[93] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. RFC 9162, December 2021.

[94] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *IMC*, 2015.

[95] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. Cryptology ePrint Archive, Paper 2023/643, 2023.

[96] Moxie Marlinspike. Convergence. convergence.io, 2011.

[97] Moxie Marlinspike. Trust Assertions for Certificate Keys. Internet-Draft draft-perrin-tls-tack-02, Internet Engineering Task Force, January 2013. Work in Progress.

[98] Matter Labs. bellman_ce. https://github.com/matter-labs/bellman, 2023.

[99] Sarah Meiklejohn, Joe DeBlasio, Devon O'Brien, Chris Thompson, Kevin Yeo, and Emily Stark. SoK: SCT auditing in Certificate Transparency. *PETS*, 2022.

[100] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[101] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[102] National Institute of Standards and Technology. Digital Signature Standard (DSS), feb 2023.

[103] Zachary Newman. Reducing Trust in Automated Certificate Authorities via Proofs-of-Authentication. *arXiv preprint arXiv:2307.08201*, 2023.

[104] Andrija Novakovic and Kobi Gurkan. Groth16 malleability. https://geometry.xyz/notebook/groth16-malleability, 2022.

[105] Magnus Nyström and Burt Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986, November 2000.

[106] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections. In *USENIX Security*, 2015.

[107] OpenSSL. OpenSSL. https://github.com/openssl/openssl, 2024.

[108] OpenSSL. OpenSSL asn1parse. https://docs.openssl.org/1.1.1/man1/asn1parse/, 2024.

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

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

[111] Riva Richmond. An Attack Sheds Light on Internet Security Holes. *The New York Times*, April 2011.

[112] Roman Semenov. zkUtil. https://github.com/poma/zkutil, 2021.

[113] Scott Rose, Matt Larson, Dan Massey, Rob Austein, and Roy Arends. Resource Records for the DNS Security Extensions. RFC 4034, March 2005.

[114] Michael Rosenberg, Jacob White, Christina Garman, and Ian Miers. zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure. In *IEEE Security and Privacy*, 2023.

[115] Lorenz Schwittmann, Matthäus Wander, and Torben Weis. Domain impersonation is feasible: a study of CA domain validation vulnerabilities. In *IEEE EuroS&P*, 2019.

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

[117] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Eurosys*, 2013.

[118] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with Lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023.

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

[120] Haya Shulman and Michael Waidner. One key to sign them all considered vulnerable: Evaluation of DNSSEC in the internet. In *NSDI*, 2017.

[121] Ryan Sleevi. Sustaining Digital Certificate Security. Google Security Blog, October 2015. `https://security.googleblog.com/2015/10/sustaining-digital-certificate-security.html`.

[122] Trevor Smith, Luke Dickinson, and Kent Seamons. Let's revoke: Scalable global certificate revocation. In *NDSS*, 2020.

[123] Sooel Son and Vitaly Shmatikov. The hitchhiker's guide to DNS cache poisoning. In *ICST*, 2010.

[124] Emily Stark, Joe DeBlasio, and Devon O'Brien. Certificate transparency in Google Chrome: Past, present, and future. *IEEE Security & Privacy*, 19(6), 2021.

[125] Michael StJohns. Automated Updates of DNS Security (DNSSEC) Trust Anchors. RFC 5011, September 2007.

[126] Ernst G. Straus. Addition chains of vectors (problem 5125). *American Mathematical Monthly*, 71:806–808, 1964.

[127] Andrew Sutherland. Elliptic curves. `https://ocw.mit.edu/courses/18-783-elliptic-curves-spring-2021`, 2021.

[128] Justin Thaler. Proofs, Arguments, and Zero-Knowledge. `http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html`, 2020.

[129] Mikhail Volkhov. *Malleable Zero-Knowledge Proofs and Applications*. PhD thesis, University of Edinburgh, 2023.

[130] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

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

[132] Dan Wendlandt and Adrian Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX Annual Technical Conference*, 2008.

[133] Pengcheng Xia, Haoyu Wang, Zhou Yu, Xinyu Liu, Xiapu Luo, and Guoai Xu. Ethereum Name Service: the Good, the Bad, and the Ugly. *arXiv preprint arXiv:2104.05185*, 2021.

[134] Collin Zhang, Zachary DeStefano, Arasu Arun, Joseph Bonneau, Paul Grubbs, and Michael Walfish. Zombie: Middleboxes that don't snoop. In *NSDI*, 2024.

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

[136] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, 2017.

[137] Shuhao Zheng, Zonglun Li, Junliang Luo, Ziyue Xin, and Xue Liu. IDEA-DAC: Integrity-Driven Editing for Accountable Decentralized Anonymous Credentials via ZK-JSON. Cryptology ePrint Archive, Paper 2024/292, 2024.

A NOPE-managed

NOPE-managed (§3) makes two changes versus NOPE. First, the proof statement is somewhat larger, which affects only costs at setup-time (roughly, twice as expensive for the prover). Second, NOPE-managed doesn't require zero knowledge (§2.3), only succinctness, as no secret information enters the proof.

Instead of proving knowledge of the private key, the domain owner writes a hash of their TLS public key, *TS*, and *N* to a `TXT` record and then creates a proof of the existence of a valid DNSSEC chain of signatures to this record.

The proof statement. NOPE-managed's proof statement ($\mathbf{S}_{\text{NOPE-managed}}$) is similar to $\mathbf{S}_{\text{NOPE}}$, but with $\mathbf{S}_{DS.K}$ replaced by $\mathbf{S}_{TXT}$. $\mathbf{S}_{TXT}$ checks two things:

(1) *TXT-parse* ($\mathbf{S}_{TXT.P}$): The hash of the TLS public key T, *TS*, and *N* is contained in a validly formatted `TXT` record *R* on domain *D*.

(2) *TXT-signature* ($\mathbf{S}_{TXT.S}$): *R* has a signature (`RRSIG`) that is validated by a key K', presumed to be the ZSK of *D* itself.

$\mathbf{S}_{TXT}$ ties the contents of a `TXT` record to a specific key K'. The next step is verifying that K' is a ZSK for *D*. This is done by repeatedly applying $\mathbf{S}_{ZSK}$ from $\mathbf{S}_{\text{NOPE}}$, which ultimately reduces the claim about *D*'s ZSK to claim about the root ZSK.

For example, $\mathbf{S}_{TXT}$ lifts a claim about the contents of a `TXT` record in `example.com` to a more general claim about `example.com`. Next, $\mathbf{S}_{ZSK}$ lifts this claim about `example.com` to a claim about `.com`. Applying $\mathbf{S}_{ZSK}$ again lifts a claim about `.com` to a claim about the root.

B Other string primitives in constraints

B.1 NOPE's string slicing primitive

mask isolated a variable-length prefix from a fixed-length buffer. Here, we present *slice*, which extracts this fixed-length buffer from a much larger buffer containing the original message. The signature of *slice* is *slice*<*M*, *L*>(msg, i). This is a function that takes a buffer *msg* of length *M* and returns a buffer *arr* of length *L* that starts at index *i* in *msg*. This function is templated by the lengths of the input and output buffers *M* and *L* respectively, as both are known at compile time.

The (simplified) naive implementation of *slice* is as follows:

```
1: procedure sliceNaive<M,L>(msg,i)
2:     for j ∈ [1,M−L] do
3:         res[j] ← msg[i+j]
4:     return res
```

Because i is not known at compile time, this requires emulating RAM in constraints. There are two ways to do this.

(1) Techniques for emulating RAM that require $O((M+L)\cdot\log(M+L))$ constraints [130].

(2) A scan technique that requires $M\cdot L$ constraints and is asymptotically worse but has smaller constants [13].

The first approach is typically preferred when L is small relative to M.

As an alternative, NOPE introduces a lightweight primitive with linear complexity and small constants, and then chains a logarithmic number of these primitives together to construct a more efficient version of *slice*. This primitive, *condshift*, has the signature *condshift*<M, D>$(msg, flag)$. Given a buffer *msg* of length M, and a Boolean flag *flag*, this function returns a buffer *res* of length M with two possibilities:

(1) If *flag* is 0, *res* is a copy of *msg*.

(2) If *flag* is 1, *res* is *msg* shifted left by D.

This is implemented as follows:

```
1: procedure condShift<M,D>(msg,flag)
2:     res ← ∅
3:     for i ∈ [1,M−D] do
4:         res[i] ← (flag ? msg[i+D] : msg[i])
5:     for i ∈ [M−D,M] do
6:         res[i] ← (flag ? 0 : msg[i])
7:     return res
```

The total cost of this procedure is M constraints: each ternary can be compiled to a single constraint.

NOPE implements *slice*<M, L>(msg,i) by repeatedly applying *condshift*. The implementation looks like the following:

```
1: procedure slice<M,L>(msg,i)
2:     arr ← msg
3:     bits ← toBinary(1+logM)(i)
4:     # iterate down from 1 + log M to 1
5:     for j ∈ [1+logM,1] do
6:         # reachable prefix length
7:         M' ← min(L+2^j−1,M)
8:         # conditional shift by 2**(j−1)
9:         S ← 2^(j−1)
10:        # if the jth bit of i is set
11:        arr ← condshift<M',S>(arr,bits[j])
12:    return arr
```

toBinary is a standard function which converts an integer to an array of bits (and costs $2+\log M$ constraints). The complexity of this primitive requires careful cost analysis, but the worst case cost is $M\log(M)+\log(M)+2$ constraints, an asymptotic and concrete improvement over the naive implementation. For small L, this effectively becomes $O(M)$ constraints with a small constant.

It is possible to fully eliminate the $\log M$ term in some cases by packing adjacent array elements into a single field element. This looks like the following:

```
1: procedure sliceAndPack<M,L>(msg,i)
2:     arr ← msg
3:     bits ← toBinary(1+logM)(i)
4:     # iterate down from 1 + log M to 1
5:     for j ∈ [1+logM,1] do
6:         # reachable prefix length
7:         M' ← min(L+2^j−1,M)
8:         # conditional shift by 2**(j−1)
9:         S ← 2^(j−1)
10:        # if the jth bit of i is set
11:        arr ← condshift<M',S>(arr,bits[j])
12:        # merge adjacent elements
13:        for k ∈ [1,M'] do
14:            arr[k] ← arr[2·k−1]+arr[2·k]
15:    return arr
```

This reduces the cost to just under $2 \cdot M + \log(M) + 2$ constraints, but the final output is in a different (packed) format. Depending on the application, this additional optimization may or may not be useful.

B.2 NOPE's string scan primitive

slice isolated a fixed length buffer from a much larger buffer. but this requires knowing where to start the extraction. To find this location, we describe a recipe for implementing *scan*. This recipe generalizes to any length-prefixed format.

The signature of *scan* is *scan*<*M*>(*msg*, *start*, ...). This is a function that takes a buffer `msg` of length M and checks that *start* is actually the start of a field of a certain type. *start* is a private input that is computed outside of the constraints and supplied as part of the witness (**W**) to be checked. The "..." at the end of the signature leaves room for additional inputs depending on context. This function returns the length of the field that starts at *start*.

Below, we present a toy example of what DNS RRsets look like for the purposes of exposition. These toy DNS RRsets consist of a header followed by a sequence of records. The header is a variable-length domain name field, and records are triples of the following format:

(1) a 1-byte length field

(2) a 1-byte type field

(3) a variable-length data field

We will consider the case where an application needs to scan for the initial index (and length) of an arbitrary record in the RRset to be fed into *slice* and *mask*.

Given the public length of the domain name (and thus the header), called *headerlen*, a private index *start*, and a private buffer *msg* of length M, NOPE introduces a function *scanToyRRset* that scans for the start of a record as follows.

```
1: procedure scanToyRRset<L>(msg,start,headerlen)
2:     counter ← headerlen
3:     len ← 0
4:     loc ← indicator<M>(start)
5:     for i ∈ [1,M] do
6:         # counter is only 0 when
7:         # i is at the start of a record.
8:         # loc is only non-zero when i == start.
9:         # start must be the start of a record
10:        constrain counter * loc[i] == 0
11:        # read length (but only if i == start)
```

```
        len ← len + msg[i] * loc[i]
        # if we are at the start of a record
        # read the length and reset the counter
        counter ← (counter == 0 ? msg[i] : counter) − 1
    return len
```

This procedure requires $4 \cdot M + 1$ constraints. It verifies that *start* is the start of a record, thus validating the input to *slice*. This procedure also provides the true length of the record, which is necessary for *mask*.

Unlike this toy RRset, real DNS RRsets have length fields longer than 1 byte and more fields; however, the same mechanics presented here apply.

C Fewer EC point operations in ECDSA

Recall (§5.3) that an ECDSA public key is a point, $\mathcal{Q}$, on the elliptic curve; the private key is a 256-bit scalar d (providing 128 bits of security). The public key is defined: $\mathcal{Q} := d \cdot \mathcal{G}$, where $\mathcal{G}$ is a generator of the curve. Let n be the order of the generator.

As a simplification, let the ECDSA signature for a message msg be $\sigma = (\mathcal{R}, s)$. Additionally, let $H(msg)$ be the hash of msg. This signature is verified in three steps:

$$\begin{aligned} \text{compute: } & h_0 \leftarrow s \cdot H(msg) \bmod n \\ \text{compute: } & h_1 \leftarrow s \cdot x_{\mathcal{R}} \bmod n \\ \text{check: } & \mathcal{R} = h_0 \cdot \mathcal{G} + h_1 \cdot \mathcal{Q} \end{aligned}$$

The costliest operation here is the final step, the 256-bit multi-scalar multiplication (MSM; §5.2). If the prover can find a 128-bit scalar v such that $h_1 \cdot v \bmod n$ is small, then the 256-bit MSM can be transformed into a 128-bit MSM [5, 51]. Such a v always exists; however, normally finding this v is not worth the effort. The main idea that makes it worthwhile for NOPE is that the prover can find v outside of constraints. Then, the constraints check that v has the desired property, and check a 128-bit MSM instead of a 256-bit MSM. This saves nearly $2\times$ in constraints. The overheads outside of constraints are swamped by the savings in constraints.

In more detail, given $\mathcal{H} := 2^{128}\mathcal{G}$ precomputed offline, the prover uses the extended Euclidean algorithm to find a non-zero v such that $h_1 \cdot v \bmod n$ and v can each be represented in 128 bits. Then, the third step in signature verification (above) becomes:

$$\begin{aligned} \text{check: } & 0 < v < 2^{128} \\ \text{compute: } & t \leftarrow h_0 \cdot v \bmod n \\ \text{compute: } & v_0 \leftarrow t \bmod 2^{128} \\ \text{compute: } & v_1 \leftarrow \lfloor t/2^{128} \rfloor \\ \text{compute: } & v_2 \leftarrow h_1 \cdot v \bmod n \\ \text{check: } & v_2 < 2^{128} \\ \text{check: } & O = v_0 \cdot \mathcal{G} + v_1 \cdot \mathcal{H} + v_2 \cdot \mathcal{Q} - v \cdot \mathcal{R} \end{aligned} \tag{3}$$

Checking the conditions on v and v_2, and computing t, v_0, v_1, and v_2 are relatively inexpensive. The final step is a 128-bit MSM, which is far less expensive than a 256-bit MSM.

NOPE applies two other refinements. First, the MSM transformation [5, 51] uses the Straus method [126] (see also [20, 46]), in which a kind of basis is precomputed to accelerate the computation of $v_i \cdot \mathcal{P}$, where $\mathcal{P}$ stands in for each of the points in the MSM in equation (3). Although this method seemingly involves point additions and point doublings (§5.2), NOPE replaces all point doublings with point additions. Second,

point addition naively includes some cost, to handle the case that a sum produces the point at infinity. NOPE shows how to avoid that point, thereby shedding the constraint logic that would have to check and adjust for this case.

D SAN encoding example

To encode the 128-byte proof into the Subject Alternative Name (SAN) field of an X.509 certificate, NOPE uses a base-37 encoding, which fits the proof into 197 hostname characters. NOPE adds one character for versioning, another for metadata, and a final character for a checksum. The resulting 200 characters are split into four 50-character labels, (<a>, <b>, <c>, <d>). For particularly long domains, NOPE spreads these labels across multiple SANs, but for most domains this is not necessary. Each NOPE SAN is prefixed with a NOPE identifier `n0pe.` that is incremented in the multi-SAN case to indicate the order of the SANs (`n0pe.`, `n1pe.`, etc.). For a short domain, it might look like

```
n0pe.<a>.<b>.<c>.<d>.<short domain>
```

For a longer domain, it might look like

```
n0pe.<a>.<b>.<long domain>
n1pe.<c>.<d>.<long domain>
```

For example, the full NOPE SAN for `example.com` might look like:

```
n0pe.06o6kheobgzqj52omigylyk-ec3vzrbmtg42k5eajl8ykitd8j.g8t96eylete226olso-tnd
e03iu461rzgurowlimiqh2t8r47l.o8mjcv20g9vdxrcdzzlbo63nc7ldyqgndocum5ct3i4vzdelg
v.9ury0kwu2o-d6ns06ow9zihqakgqpusuyqcmj8zjauc45f171t.example.com
```