2016 IEEE Symposium on Security and Privacy

Cinderella: Turning Shabby X.509 Certificates into Elegant Anonymous Credentials with the Magic of Verifiable Computation

Antoine Delignat-Lavaud Cédric Fournet Markulf Kohlweiss Bryan Parno
{antdl,fournet,markulf,parno}@microsoft.com
Microsoft Research

***Abstract*—Despite advances in security engineering, authentication in applications such as email and the Web still primarily relies on the X.509 public key infrastructure introduced in 1988. This PKI has many issues but is nearly impossible to replace.**

Leveraging recent progress in verifiable computation, we propose a novel use of existing X.509 certificates and infrastructure. Instead of receiving & validating chains of certificates, our applications receive & verify *proofs* of their knowledge, their validity, and their compliance with application policies. This yields smaller messages (by omitting certificates), stronger privacy (by hiding certificate contents), and stronger integrity (by embedding additional checks, e.g. for revocation).

X.509 certificate validation is famously complex and error-prone, as it involves parsing ASN.1 data structures and interpreting them against diverse application policies. To manage this diversity, we propose a new format for writing application policies by composing X.509 templates, and we provide a template compiler that generates C code for validating certificates within a given policy. We then use the Geppetto cryptographic compiler to produce a zero-knowledge verifiable computation scheme for that policy. To optimize the resulting scheme, we develop new C libraries for RSA-PKCS#1 signatures and ASN.1 parsing, carefully tailored for cryptographic verifiability.

We evaluate our approach by providing two real-world applications of verifiable computation: a drop-in replacement for certificates within TLS; and access control for the Helios voting protocol. For TLS, we support fine-grained validation policies, with revocation checking and selective disclosure of certificate contents, effectively turning X.509 certificates into anonymous credentials. For Helios, we obtain additional privacy and verifiability guarantees for voters equipped with X.509 certificates, such as those readily available from some national ID cards.

I. Introduction

Since its standardization in 1988, the X.509 public key infrastructure (PKI) has become ubiquitous. It serves as the cornerstone of security for Web browsing (TLS), software updates (Authenticode), email (S/MIME), document authentication (PDF), virtual private networks (IPSec), and much more. In part because of this ubiquity, the X.509 PKI is averse to change. Even when concrete attacks are known, as when Stevens et al. [64] used a collision on MD5 to forge a bogus certificate, it still takes years to replace insecure algorithms.

Given this structural inertia, it is perhaps unsurprising that, despite almost thirty years of innovation in security and cryptography, the X.509 PKI primarily employs techniques and algorithms known at its inception. Since that time, the cryptographic community has developed schemes for anonymous authentication [61], pseudonymous authentication [24] and attribute-based authentication [15], for group signatures [25], and for cryptographic access control [22]. However, none of these features are available in the world's default PKI.

Beyond inertia, one barrier to innovation is a disconnect between the existing X.509 infrastructure and the 'primitive' operations required in these cryptographic schemes. The vast majority of X.509 keys and certificates employ RSA, whereas many modern schemes rely on a particular algebraic structure, often in an elliptic curve group [46], e.g. a discrete logarithm representation or a bilinear relation between elements [1, 20]. Furthermore, in many deployments, the private RSA keys reside in secure hardware (e.g., in a smartcard for client authentication, or in a hardware security module (HSM) for server authentication), and hence can only be accessed via a predefined API. Finally, X.509 certificate parsing and processing is notoriously complex even in native code [17], let alone in the context of a cryptographic scheme.

With Cinderella, we leverage recent progress in verifiable computation to bridge the gap between existing X.509 infrastructure and modern cryptography. Abstractly, a verifiable computation protocol [41] enables a verifier to outsource an arbitrary computation to an untrusted prover before efficiently checking the correctness of that computation. Both the verifier and the prover can supply inputs to the computation, and the prover may opt to keep some or all of his inputs private.

Cinderella employs such a verifiable computation protocol so that a prover can demonstrate that he holds (1) a valid X.509 certificate chain and (2) a signature computed with the associated private key, without actually sending them to the verifier. In other words, Cinderella outsources to the prover all of the checks that the verifier would have done on those certificates, and the verifier's job is simplified to checking only that the outsourced validation was performed correctly.

Cinderella's approach allows us to re-use existing certificate chains (including well-established certificate authorities and issuance policies) and their signing mechanisms in more advanced applications. It integrates well with existing infrastructure, since it does not require direct access to X.509 signing keys, making it compatible with existing hardware-based solutions, such as smartcards and HSMs. Furthermore, as discussed shortly, Cinderella can drop seamlessly into existing protocols such as TLS. As an important example, several countries such as Belgium, Estonia, and Spain issue "national identity" X.509 certificates on smartcards, and even

DOI 10.1109/SP.2016.22

provide APIs for commercial applications. We can directly re-use these carefully-managed, highly-trusted identity providers, without support or even approval from their authorities.

Cinderella adds "cryptographic power" by improving the flexibility, expressivity, and privacy of X.509 authentication and authorization decisions. For instance, although the PKI currently supports certificate revocation via signed revocation lists (CRLs) and online checks (OCSP), checking for revocation remains brittle, as one may need to collect evidence from third-party servers and deal with potential failures in the process. With Cinderella, a given validation policy can move the responsibility of collecting (and validating) recent evidence of non-revocation to the certificate holder, therefore making revocation checking simpler and more efficient for the verifier. Other issuers may prefer a policy that accepts only recent, short-lived certificates. Both approaches can co-exist, and the resulting joint policy can be enforced by the verifier without any additional local processing. Similarly, Cinderella proofs can embed other advanced validation policies, such as controlled delegation and Certificate Transparency (CT [45]).

Our evaluation (§VIII) shows that, depending on the chain length and certificate type, Cinderella proofs are 1.2–5.4× smaller than the evidence that would otherwise be communicated. Policies that embed checks that would involve collecting additional evidence from a third party (such as OCSP) also allow significant latency gains by eliminating this request entirely. Even though some protocols (e.g., TLS) feature mechanisms to attach ("staple") third party evidence, Cinderella still saves the cost of sending such evidence (which can take several additional kilobytes). Cinderella proofs can be verified at a cost comparable to native certificate chain validation (8 ms). However, proof generation is many orders of magnitude slower than native execution (10 minutes for the most complex policies in §VIII) and involves large keys (up to 1 GB). In the applications we consider, we deal with the prover overhead by computing proofs offline, and by re-using the computed proof for short periods of time.

From a privacy standpoint, because modern verifiable computation protocols support zero knowledge properties for the prover's inputs, Cinderella enables the selective disclosure of information embedded in standard X.509 certificates. Instead of revealing these certificates in the clear, the prover can convey only the attributes needed for a particular application. For example, the outsourced computation can validate the prover's certificate chain, and then check that the issuer is on an approved list, that the prover's age is above some threshold, or that he is not on a blacklist. The verifier learns that these checks were performed correctly, and nothing more. As a concrete example, Estonian ID certificates embed information about the subject's name, address and email, as well as a unique national identity number ("isikukood"), which encodes the gender and birth date of the owner. Estonian law mandates ownership of an ID card for citizens over 15, and by necessity considers the information it contains public (as it may be sent in clear), even though many users, if given the choice, may prefer to keep some of this information private when signing into government or commercial websites. Certificate privacy is also compelling for scenarios such as e-voting, where the strong identification provided by X.509-based ID cards needs to be balanced with voter privacy.

Cinderella uses Pinocchio [30, 59], a state-of-the-art system for verifiable computation. Pinocchio compiles C code first into arithmetic equations in a large prime field, and then into cryptographic keys. While Pinocchio accepts C code as input, programmed naïvely, it will produce enormous keys that require tremendous work from the prover. Thus, an important challenge for Cinderella is developing C code for standards-compliant X.509 certificate-chain validation that Pinocchio will compile into an efficient cryptographic scheme.

The first part of the chain-validation challenge is to encode the verification of RSA PKCS#1 signatures. Cinderella achieves this via a carefully implemented multi-precision arithmetic C library tailored to Pinocchio; the library takes non-deterministic hints—the quotients, residues, and carries—as input to circumvent costly modular reductions. The second challenge is to verify X.509 parsing, hashing, and filtering of certificate contents. These tasks are already complicated in native code. To handle X.509 formats efficiently, Cinderella supports policies based on certificate templates, written in a declarative language, and first compiled to tailored C code before calling Pinocchio on whole certificate-chain-validating verifiable computations.

To demonstrate Cinderella's practicality, we first show how to seamlessly integrate Cinderella-based certificate-chain validation into SSL/TLS. Rather than modifying the TLS standard and implementations, we replace the certificate chains communicated during the handshake with a single, well-formed, 564-byte X.509 pseudo-certificate that carries a short-lived ECDSA public key (usable in TLS handshakes) and a proof that this key is correctly-signed with a valid RSA certificate whose subject matches the peer's identify. We experiment with both client and server authentication. For clients, we use templates and test certificates for several national ID schemes. For servers, we use typical HTTPS policies on certificate chains featuring intermediate CAs and OCSP stapling. Although the resulting Cinderella pseudo-certificates can take up to 9 minutes to generate for complex policies, they can be generated offline and refreshed, e.g., on a daily basis. Online verification of the pseudo-certificates and their embedded proof takes less than 10 ms.

We also employ Cinderella as a front end to Helios [2], a popular e-voting platform. Assuming that every potential voter is identified by some X.509 certificate, we enhance voter privacy while enabling universal verifiability of voter eligibility. Similarly, we do not modify the Helios scheme or implementation. Rather, to each anonymous ballot, we attach a Cinderella proof that the ballot is signed with some valid certificate whose identifier appears on the voter list, and that the ballot is linked to an election-specific pseudo-random alias that is in turn uniquely linked to the voter's certificate. This allows multiple ballots signed with the same certificate to be detected and discarded. The proof reveals no information about the voter's identity. Proof generation takes 90 s, and proof verification is fast enough to check 400,000 ballots in an hour.

Contributions In a nutshell, Cinderella contributes:

- a new practical approach to retrofit some flexibility and privacy into an ossified X.509 infrastructure (§II-B);
- a real-world application of verifiable computation: outsourcing certificate-chain validation (§III);
- a template-based compiler and a collection of libraries for RSA-PKCS#1 (§IV) and ASN.1 (§V) carefully tailored for verifiable computations over X.509 certificates;
- deployment case studies for TLS (§VI) and Helios [2] (§VII) and their detailed evaluation (§VIII).

Limitations While Cinderella does not require changes to certificate authorities or other existing X.509 infrastructure (e.g., smartcards or HSMs), it does require changes to both clients and servers, which may hinder its deployment for general purpose applications like HTTPS. Conversely, under the assumption that both clients and servers can be changed, some of Cinderella's features may be achieved without advanced cryptography. For instance, one may re-use Cinderella's compiler from high-level policies to certificate-validation functions and simply run the resulting code natively at the verifier.

Hiding certificate contents may prevent local checks, such as public key pinning [38] or CAge [48]. This is a matter of policy: Cinderella may enable these checks if desired, by disclosing sufficient information.

An essential limitation of systems based on succinct proofs of outsourced computation is their reliance on a trusted party to generate the cryptographic keys. An honest key generator uses a randomly selected value to create the scheme's keys and then deletes the random value. A rogue Cinderella key provider, however, could save the random value and use it as a backdoor to forge proofs for the associated policy. Dangerously, and in contrast to the use of rogue certificates, such proofs would be indistinguishable from honest proofs and thus undetectable. Besides careful key management and local checks to reduce the scope of policies, a partial remedy is to generate keys using a multi-party protocol [10], which only requires that one of the parties involved is honest.

II. Background

We review key facts about the verifiable computation techniques (§II-A) used by Cinderella and the X.509 PKI (§II-B).

A. Verifiable Computation

Cinderella requires a succinct, zero knowledge, public, verifiable computation protocol to validate X.509 certificates. In our implementation, we use the Geppetto compiler [30] for the Pinocchio [59] protocol, itself a refinement of a protocol developed by Gennarro et al. [42]. We refer to this combination as Pinocchio, and we review it below, but our approach is compatible with similar schemes [3, 6, 8, 9, 16, 68].

Pinocchio enables a *verifier* to efficiently check computations performed by untrusted *provers*, even when the untrusted provers supply some of the computation's inputs. Concretely, the untrusted prover generates a proof that he computed $F(u,w)$, where F is a verifier-specified function, u is a verifier-specified public input, and w is a private input provided by the prover. Verifiable computation protocols supporting zero knowledge (also known as succinct, non-interactive zero-knowledge arguments [14, 43]) allow the prover to optionally keep w secret from the verifier.

More formally, Pinocchio consists of three algorithms:

1) $(EK_F, VK_F) \leftarrow \mathbf{KeyGen}(F, 1^\lambda)$: takes the function F to be computed and the security parameter λ, and generates a public evaluation key EK_F and a public verification key VK_F.
2) $(y, \pi_y) \leftarrow \mathbf{Compute}(EK_F, u, w)$: run by the prover, takes the public evaluation key, an input u supplied by the verifier, and an input w supplied by the prover. It produces the output y of the computation and a proof of y's correctness (as well as of prover knowledge of w).
3) $\{0,1\} \leftarrow \mathbf{Verify}(VK_F, u, y, \pi_y)$: using the public verification key, takes a purported input, output, and proof and outputs 1 only when $F(u,w) = y$ for some w.

In brief, Pinocchio is *Correct*, meaning that a legitimate prover can always produce a proof that satisfies **Verify**; *Zero Knowledge*, meaning that the verifier learns nothing about the prover's input w; and *Sound*, meaning that a cheating prover will be caught with overwhelming probability. Prior work provides formal definitions and proofs [30, 59].

Pinocchio offers strong asymptotic and concrete performance: cryptographic work for key and proof generation scales linearly in the size of the computation (measured roughly as the number of multiplication gates in the arithmetic circuit representation of the computation), and verification scales linearly with the verifier's IO (e.g., $|u| + |y|$ above), regardless of the computation, with typical examples requiring approximately 10 ms [59]. The proofs are constant size (288 bytes).

To achieve this performance, Pinocchio's compiler takes C code as input and transforms the program to be verified into a Quadratic Arithmetic Program (QAP) [42]. In a QAP, all computation steps are represented as an arithmetic circuit (or a set of quadratic equations) with basic operations like addition and multiplication taking place in a large (254-bit) prime field. In other words, unlike a standard CPU where operations take place modulo 2^{32} or 2^{64}, in a QAP, the two basic operations are $x + y \bmod p$ and $x * y \bmod p$, where p is a 254-bit prime. As a result, care must be taken when compiling programs. For example, multiplying two 64-bit numbers will produce the expected 128-bit result, but multiplying four 64-bit numbers will "overflow", meaning that the result will be modulo p, which is unlikely to be the intended result. To prevent such overflow, the Pinocchio compiler tracks the range of values each variable holds, and inserts a reduction step if overflow is possible. Reduction involves a bit-split operation, which splits an arithmetic value into its constituent bits. Bit splits are also necessary for bitwise operations, such as XOR, as well as for inequality comparisons.

In Pinocchio's cost model, additions are free, multiplications cost 1, and bit splits cost 1 per active bit in the value split. Hence splitting the result of multiplying two 64-bit numbers costs $128\times$ more than the initial multiplication did. Finally, dynamic array accesses (i.e., those where the index into the array is a run-time value) must be encoded and

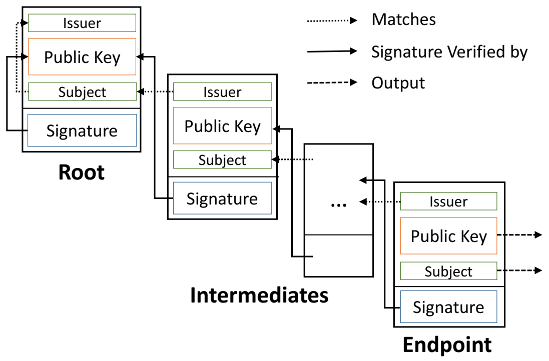

Fig. 1. High-level overview of the X.509 PKI

can be expensive. While various techniques have been devised [5, 9, 13, 16, 56, 68, 70], the costs remain approximately $O(\log N)$ per access to an array of N elements.

From the programmer's perspective, Pinocchio compiles typical C functions. Each function takes as arguments the inputs and outputs known to the verifier (the values u and y above). The function can also read, from local files, additional inputs available only to the prover (the value w above).

B. The X.509 Public Key Infrastructure

We recall X.509's salient characteristics and summarize the main classes of issues with the PKI. Clark et al. provide more details and references [28].

X.509 defines the syntax and semantics of public key certificates and their issuance hierarchy. The purpose of a certificate is to bind a public key to the identity of the owner of the matching private key (the *subject*), and to identify the entity that vouches for this binding (the *issuer*). Certificates also contain lifetime information, extensions for revocation checking, and extensions to restrict the certificate's use.

The PKI's main high-level API is certificate-chain validation (illustrated in Figure 1), which works as follows: given a list of certificates (representing a *chain*) and a validation context (which includes the current time and information on the intended use), it checks that:

1) the certificates are all syntactically well-formed;
2) none of them is expired or revoked;
3) the issuer of each certificate matches the subject of the next in the chain;
4) the signature on the contents of each certificate can be verified using the public key of the next in the chain;
5) the last, *root* certificate is trusted by the caller; and
6) the chain is valid with respect to some context-dependent application policy (e.g. "valid for signing emails").

If all these checks succeed, chain validation returns a parsed representation of the identity and the associated public key in the first certificate (the *endpoint*).

Syntactic & semantic issues X.509 certificates are encoded using the Distinguished Encoding Rules (DER) of ASN.1, whose primary goal is to ensure serialization is injective: no two distinct certificates have the same encoding. In ASN.1, there is no absolute notion of syntactic correctness; instead, a payload is well-formed with respect to some ASN.1 grammar that specifies the overall structure of the encoded data, with semantic annotations such as default values and length boundaries (Figure 2 depicts a fragment of the X.509 grammar).

```
Certificate  ::=  SEQUENCE  {
 tbsCertificate       ToBeSigned,
 signatureAlgorithm   AlgorithmIdentifier,
 signature            BIT STRING }

ToBeSigned  ::=  SEQUENCE  {
 version         [0]  Version DEFAULT v1,
 serialNumber         SerialNumber,
 signature            AlgorithmIdentifier,
 issuer               Name,
 validity             Validity,
 subject              Name,
 subjectPublicKeyInfo SubjectPublicKeyInfo,
 issuerUniqueID  [1]  IMPLICIT UID OPTIONAL,
 subjectUniqueID [2]  IMPLICIT UID OPTIONAL,
 extensions      [3]  Extensions OPTIONAL }
```

Fig. 2. ASN.1 Grammar of X.509 Certificates

Mistakes and inconsistencies in implementations of X.509 have led to dozens of attacks [26]. Famously, the first version of X.509 did not include a clear distinction between certificate-authority (CA) and endpoint certificates. A later version introduced an extension to clarify this distinction, but Marlinspike [55] showed that several browsers could be confused to accept endpoint certificates as intermediates in a chain, and similar errors have reoccurred periodically. Similarly, Bleichenbacher showed that many implementations of DER are incorrect, leading to universal forgery attacks against PKCS#1 signatures; again, variations of this attack have reappeared every so often. In contrast, Cinderella does not trust X.509 parsers; instead, it verifies the correctness of untrusted parsing by re-serializing and hashing (§V).

Cryptographic failures The X.509 PKI is slow to change; it is not uncommon for certification authorities (CAs) to use the same root and intermediate certificates for years, without upgrading the cryptographic primitives used for signing. For instance, the MD5 hashing algorithm remained widely used for several years after Wang et al. [69] demonstrated that it was vulnerable to collision attacks. The ongoing migration from SHA1 to SHA2 has also been delayed by over a year, due to pressure to maintain legacy support. Similarly, a number of certification authorities have allowed the use of short RSA public keys [33], or keys generated with low entropy [47]. As a moot point, Cinderella may partially mitigate these issues by allowing certificate owners to hide their public keys and certificate hashes, and hence to prevent some offline attacks; arguably, it also makes such issues harder to measure.

Issuance & validation policy issues As explained above, certificate-chain validation includes a notion of compliance with respect to some application policy. For instance, for HTTPS, a certificate needs to satisfy a long list of conditions to be considered valid. Even the most basic condition is quite complicated: the HTTP domain of the request must match either the common name field of the subject of the certificate, or one of the entries of type DNS from the Subject Alternative Names (SAN) extension. This matching is not simply string equality, as domains in certificates may contain wildcards.

Fahl et al. [39] show that a large number of Android applications use a non-default validation policy for their application. More generally, Georgiev et al. [44] report that a large fraction of non-browser uses of the PKI use inadequate validation policies. Instead of writing a custom policy suited to their application (e.g., pinning, custom root certificates, trust on first use), most developers simply use an empty validation policy that can be trivially bypassed by an attacker.

Similarly, even though all certification authorities are subject to a common set of issuance guidelines [18, 23, 37], the variability of their issuance policies remains high [33]. Thus current validation policies are only as strict as the PKI's most permissive policy (in terms of key sizes, maximum lifetime, or availability of revocation information). With Cinderella, validation policies are mostly specified through a declarative template system (§III) and transformed into Pinocchio keys, allowing greater flexibility from one issuer to the other.

Revocation X.509 revocation checking can take one of two forms: revocation lists (CRL) must be downloaded out of band, while the *online certificate status protocol* (OCSP) can either be queried by the validator to obtain a signed proof of non-revocation; or this proof may be *stapled* to the certificate to prevent a high-latency query. Unfortunately, these mechanisms are not widely used in practice; a 2013 study indicated that only 20% of servers supported OCSP stapling [58]. A similar study conducted in 2015 showed an even lower number: only 3% of certificates were served by hosts supporting OCSP stapling [54]. Worse, once a certificate is compromised and subsequently revoked [51], they prevent attacks only inasmuch as failures to verify non-revocation are treated as fatal errors, an issue recognized and quantified in recent PKI papers [4, 63, 65]. As shown in §VI, Cinderella naturally supports OCSP stapling with no additional effort from the verifier.

Delegation Many practical applications rely on some form of authentication delegation. In particular, many servers delegate the delivery of their web content to *content delivery networks* (CDNs). Websites that use HTTPS with a CDN need to give their X.509 credentials to the CDN provider, which can cause serious attacks when CDNs improperly manage customer credentials [32]. In a survey about this problem, Liang et al. [52] propose to reflect the authentication delegation of HTTPS content delivery networks as X.509 delegation. Unfortunately, this is impractical, because it requires an extension of X.509 which CAs are unlikely to implement, as it is detrimental to their business. Cinderella allows a content-owner to implement secure X.509 delegation to CDNs using short-lived pseudo-certificates, without the CA's cooperation (§VI).

III. Cinderella's Certificate-Chain Validation

A. Architecture Overview

Cinderella targets applications in which a *certificate holder* presents a certificate chain to a *validator* who checks both that the chain is well formed (§II-B) and that it adheres to the application's validation policy. With Cinderella, the validator

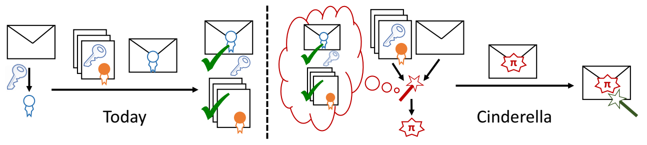

Fig. 3. Cinderella S/MIME example. Today, an email sender includes her signature and a certificate-chain for her public key, and the email's recipient checks both. With Cinderella, the sender performs those checks using verifiable computation and produces a succinct proof π that the recipient checks.

no longer performs these checks; instead, we outsource them to the certificate holder using verifiable computation (§II-A). Specifically, we write a procedure (as C code) that checks a certificate chain and checks that the chain adheres to the validation policy. We then compile this procedure into public evaluation and verification keys.

As a concrete running example, consider a client who wishes to sign her email using the S/MIME protocol (Figure 3). She holds a certificate issued by a well-known CA for her public key, and she uses her corresponding private key to sign a hash of her message.

With the current PKI, she attaches her certificate and signature to the message. The recipient of the message extracts the sender's email address (*from*), parses and checks the sender's certificate, and verifies, in particular, that the sender's certificate forms a valid chain together with a local, trusted copy of the CA certificate; that its subject matches the sender's address (*from*); and that it has not expired. Finally, he verifies the signature on a hash of the message using the public key from the sender's certificate. These checks may be performed by a C function, declared as

```
void validate(SHA2* hash, char* from, time* now,
              CHAIN* certs, SIG* sig);
```

For simplicity, assume that all S/MIME senders and receivers agree on this code for email signatures, with a fixed root CA.

With Cinderella, we compile `validate` into cryptographic keys for S/MIME, i.e., an evaluation key and a verification key (§II-A). Email-signature validation then proceeds as follows.

- The sender signs the hash of her message as usual, using the private X.509 key associated with her certificate. Instead of attaching her certificate and signature, however, she attaches a Pinocchio proof. To generate this proof, she calls Cinderella with the S/MIME evaluation key, her message hash, email address, time, certificate, and signature. Cinderella runs `validate` on these arguments and returns a proof that it ran correctly.
- Instead of calling `validate`, the recipient calls Cinderella with the S/MIME verification key and the received message's *hash*, its *from* field, and its proof. Although the certificate and signature never leave the sender's machine, the recipient is guaranteed that `validate` succeeded, and hence that the message was not tampered with.

While this protocol transformation requires the sender to generate the Pinocchio proof, it still fully enforces the recipient's validation policy (by Pinocchio's soundness); it offers greater privacy to the sender, since her certificate need not be revealed (by Pinocchio's zero-knowledge properties); and it simplifies the recipient's task, since he now runs a fixed verification algorithm on a fixed proof format.

Furthermore, it is possible to generate Cinderella keys for extended policies not supported by S/MIME. For instance, if the recipient is a mailing list, `validate` may also check that the email address listed in the certificate is a member of the mailing list, or even that the sender holds a valid list-membership certificate. Thus, Cinderella naturally enables group or attribute-based signatures using existing credentials.

Next, we address the challenges in specifying policies (§III-B), checking certificate chains (§III-C), and managing Cinderella's evaluation and verification keys (§III-D).

B. Template-Based Certificate Validation Policies

We need to capture application policies in a high-level, programmatic manner. Indeed, while Pinocchio guarantees the correct execution of the validation code, it will not check that the code itself correctly implements the intended policy.

To this end, Cinderella supports validation policies written by composing certificate templates [33] (one for each kind of certificate that may appear in a chain) and by adding custom application checks, for instance matching an email address with the common name of a certificate. Thus, application writers can author mostly-declarative policies by customizing templates and adding a few lines in C, while Cinderella automatically translates their templates into custom, lower-level, optimized C code that deals with parsing and cryptography.

None of this relies on new cryptography for verifiable computation. Arguably, our templates and declarative policies for certificate-chain validation are of independent interest. Their translation may be adapted to generate code for local enforcement at the verifier, instead of outsourcing.

1) Writing X.509 Templates: Certificate templates define classes of certificates that differ only in the values of a fixed set of variable fields. Fortunately, issuers tend to conform to a relatively small set of templates: a recent study used machine learning to derive 1500 endpoint templates that sufficed to cover over a million certificates issued during a one year period [33]. As we also use templates for CA certificates and partial chains, we claim our approach can scale to the whole PKI, at the cost of managing more Cinderella keys than ($\sim$ 200) root certificates.

Cinderella defines a syntax similar to ASN.1 grammars for writing certificate templates. This syntax supports all the ASN.1 types for data structures in X.509: sequences and sets, encapsulated bit and octet strings, and custom tagging. Our template syntax also supports the primitive types used in certificates: integers, object identifiers, timestamps, and various flavors of strings. All primitive fields must be defined as *constant* or *variable*.

Variable fields (`var<type,x,n,m>`) use four parameters: *type* is the ASN.1 type of the variable, x is the name of the variable field, and $n..m$ is the range of the length (in bytes) of the field. As we discuss in detail in §V, bounding the length of variable fields is critical for performance.

```
seq { # Validity Period
  var<date, notbefore, 13, 13>;
  var<date, notafter, 13, 13>; };
seq { # 2 to 3 subjects fields
  varlist<subjectnum, 2, 3>:
  set {
    seq { # each with an OID and a value
      var<oid, subject_oid, 3, 6>;
      var<x500, subject_val, 5, 25>; };};};
seq { # Public Key
  seq { # Key algorithm (RSA)
    const<O1.2.840.113549.1.1.11>;
    const<null>; };
  bitstring: # Encapsulated key
    seq {
      var<int, pubkey, 257, 257>; # 2048 bits
      const<65537L>; # fixed public exponent
};};
```

Fig. 4. Fragment of a template for a class of email certificates

As a concrete example, Figure 4 shows a fragment from a certificate template for S/MIME (the full template requires less than 200 lines). The fragment specifies the validity period, the subject, and the public key of the certificate. Following current practice, this template uses a constant signature algorithm (1.2.840.113549.1.1.1 is the object identifier for RSA keys), and public exponent $e = 65537$. The ASN.1 type of constants is inferred from the syntax; for instance, object identifiers start with an O, while integer types end with an L.

In addition to variable fields, we support constructors for structural variability: `option<x>` allows an entire substructure to be omitted (e.g. an optional extension), while `varlist<x,n,m>` allows a substructure to be repeated a bounded number of times. For instance, the subject of a certificate is encoded as a list of key-value pairs (where the key is an object identifier that represents, say, the country, organization or address of the subject). The template in Figure 4 allows certificates with either 2 or 3 subject fields (allowing for instance subjects with an optional contact email).

A certificate *matches* a given template when there exists a (well-typed) assignment to the template variables such that the certificate and the template yield exactly the same sequence of bytes according to the X.509 grammar.

Besides algorithm identifiers, templates mandate many checks on certificates. For example, one portion of our S/MIME template (not shown in Figure 4) mandates that the sender-certificate's issuer matches the (fixed) CA certificate's subject, and that its 'extended key usage' have its 'email authentication' flag set.

2) Compiling X.509 Templates to C Verifiers: Cinderella includes a compiler from templates to C certificate-verifiers, that is, C functions that take as parameters the RSA modulus of the certificate parent, an assignment to the template variables and (as auxiliary input) an RSA signature σ. Each function (1) computes a hash of the ASN.1-formatted certificate that results from instantiating the template with the concrete variable assignments; and (2) verifies that σ is a valid signature on that hash using the parent's modulus. Thus, given an assignment, the certificate-verifier code guarantees the existence of a well-signed certificate that matches its template.

```
typedef struct { unsigned char v[13]; } notbefore;
typedef struct { unsigned char v[13]; } notafter;
typedef struct { int len; unsigned char v[6]; } subject_oid;
...
void hash_MailCert(SHA2* hash, subject_oid* soid, ... ){
    hashBuffer b; // hash buffer, explained in §V
    append(&b, 0x30);
    ...
    for(i=0; i<6; i++) // appends a string of at most 6 chars
      if(i < soid[0].len))
        append(&b, soid[0].v[i]);
    ...
    SHA2(hash, b); // computes the certificate hash
}
void verify_MailCert(modulus* mod /* parent key */,
  subject_oid* soid, notbefore* nbefore, ...){
    SHA2 hash; // certificate hash
    hash_MailCert(hash, soid, nbefore, nafter, ...);
    load_Signature("MAILCERT_SIGNATURE", signature, ...);
    verify_Signature(mod, signature, hash); }
```

Fig. 5. Fragment of the C code compiled from the template of Figure 4

```
#include "RSA.c"      // explained in §IV
#include "SHA.c"      // explained in §V
#include "MailCert.c" // compiled from MailCert template

// The function outsourced by the recipient to the sender
void validate(SHA2* hash, char* from, time* now) {
  ...
  // validate sender certificate
  load_Modulus("S/MIME_CA_MODULUS", root, COMPILE_TIME);
  verify_MailCert(root, soid, nbefore, nafter, sval, pkey ...);

  // check contents against 'from' and 'now' arguments
  match_email_address(from, soid, sval);
  assert(time_compare(nbefore, now) == 1);
  assert(time_compare(now, nafter) == 1);

  // verify signature on the email hash argument
  load_Signature("MSG_SIGNATURE", signature, RUN_TIME);
  verify_Signature(pkey, signature, hash); }
```

Fig. 6. Fragment of a certificate validator for S/MIME

On the prover side, Cinderella includes a template-based parser that reads a certificate and returns the variable assignments and auxiliary inputs necessary to produce a proof. This parser is not trusted by the verifier.

Figure 5 shows a fragment of the 900 lines of C code compiled from the template in Figure 4. It includes a C structure definition for each of the variables of the template. It defines an auxiliary function `hash_MailCert`, to compute the hash of the email certificate based on concrete values for all of the template variables. This automatically generated code handles the many complications of ASN.1 encoding. As one small example, it handles the fact that the length of structured tags (such as sequences) depends on variable length fields within the structure, and even the length of the length encoding may vary. Figure 5 shows a small example with conditional calls to `append` to add the variable-length field `soid[0]` to the certificate's hashed contents, one byte at a time.

The generated code also defines a function `verify_MailCert` that takes as an additional input the RSA modulus of the parent certificate, loads a signature from a local file, and verifies that it is a correct signature on that hash. This code may fail on bad inputs; it returns only if all checks succeed.

3) Writing Template-Based Application Policies: Although templates offer a convenient, declarative way of enforcing certificate policies, we still have to write the 'top-level' `validate` function that properly chains together template-verifier calls (following the chaining of the actual certificates) and includes application-specific checks on the values of template variables. In our prototype, these are written in C.

For example, our S/MIME policy checks that the sender's email address is listed in the subject of the certificate, that the current time is within the certificate's `notbefore` `..notafter` interval, and that the signature on the message hash verifies using the public key of the certificate. These checks are facilitated by Cinderella's library functions.

Figure 6 outlines the resulting 'top-level' validator in C, whereas §VI and §VII describe more complex examples. The code illustrates the three checks explained above. By convention, the arguments of `validate` are those provided by the verifier, whereas prover inputs are read from files. In our sample validator, the (fixed) modulus of the root certificate is read from a file, whereas the (variable) signature value is read from a file provided by the prover. More complex validators involving intermediates would include further certificate-verifiers compiled from their templates.

C. Compiling Validation Policies from C to Cinderella Keys

At this point, we have constructed a C validator that implements our application policy but, in principle, given the additional prover inputs (the certificate, the signature, etc.) we could still run this code at the verifier.

The next step is to call Cinderella in key-generation mode, passing the validator code, the template-derived certificate-verifier code, auxiliary input files for constants, and Cinderella's cryptographic libraries for handling RSA-PKCS#1 (§IV) and ASN.1 (§V). Cinderella 'bakes' all these inputs into public evaluation and verification keys for the application policy. The key-generation step is similar to certificate issuance; it must be trusted by the application users, which may in turn involve existing PKI mechanisms, or it may rely on decentralized key generation protocols [10]. In contrast with plain X.509 certificates, however, Cinderella policies are more expressive, so fewer keys may need to be deployed. In our S/MIME example, for instance, a single pair of keys covers all certificate-based signature validations for a given CA; verifier keys (which use about a kilobyte of storage) are then distributed together with mail clients and kept in their local configuration instead of the CA certificate (which use a similar amount of space). Certificate holders who wish to use Cinderella for S/MIME must download the prover key associated with their certificate's policy; this incurs a much larger overhead, as these keys may be over 160 MB.

In effect, we propose to partition all X.509 certificates of interest into classes via templates and then generate one pair of evaluation and verification keys for each combination of class and validation policy. Naïvely, we could try to compile a few generic, lax policies that accommodate, for example, all certificate chains currently accepted for webserver authentication. This approach would impose an unrealistic computational cost on the prover (see §V) and, besides, it would not help enforce custom application policies. Conversely, restricting a key pair to a particular certificate class and application policy simplifies the task of parsing and validating certificates in that class, and hence results in less effort for the Cinderella prover.

D. Discussion: Managing Cinderella keys

1) Controlling Policies (the Application's Viewpoint): Even with Cinderella, determining the 'right' X.509 certificate validation policy for a given application remains a hard problem. Nonetheless, Cinderella simplifies policy deployment: since each policy is 'baked' into a pair of public keys, a new policy can be deployed to all users by installing the short (1 to 2KB) verifier key. Since there are no changes to the verifier's software, there is no risk of inserting remotely-exploitable software errors. In comparison, today, deploying a new policy may involve a combination of software, configuration, and certificate updates.

Our approach enables applications to design and distribute their own custom policies, rather than rely on those currently supported by popular client software. For example, a service or a regulator (say, for a school, or the banking industry) may decide which checks to include, which roots to trust, which algorithms to use, and which latency to tolerate for OCSP certificates. The policy may incorporate some application logic or even algorithms not implemented by the certificate verifier. The policy may be deployed, e.g., as a key in the configuration of the client banking application, or by re-using existing public key management mechanisms, such as key-pinning. We expect such ad hoc, application-specific deployment of policies to help break the circular dependency of servers only wanting to use a policy once supported by almost all clients.

Cinderella policies also provide a greater degree of control to the application, inasmuch as their enforcement is not left up to the interpretation of the verifier's software—indeed, the proof verification steps are largely independent of the policy.

Cinderella policies also enable some emancipation from traditional certificate issuers, notably root CAs, who can currently impersonate any of their customers. As an example, an S/MIME policy may require that a class of official mail be signed by *two* certificates, issued by two independent CAs, or that the sender certificate be endorsed by some independent organization. Similarly, such policies can be deployed just by pushing a new key to the browser or the client software.

2) Enforcing Certificate Validation (the Verifier's Viewpoint): At the other end, enforcing general-purpose certificate validation is also known to be difficult and error-prone; it involves managing a certificate store, vetting root CAs, storing pinned certificates, checking for key revocation, etc.

From the verifier's viewpoint, Cinderella verification keys are just as easy (and as hard) to manage and to use as any others; in that sense, we do not 'solve' the PKI problem, we just introduce a new set of keys.

However, a single Cinderella key can enforce more flexible and expressive authorization and authentication policies than those expressible within the X.509 certificate text. Thus, a single, long-lived Cinderella key can encapsulate a complex policy that might otherwise require many short-lived traditional certificates. Experimental data suggests that, for a given application, a few policies and templates may suffice to cover the uses of X.509 certificates for most client platforms [33]. For example, instead of installing a root certificate key to access some exotic service, installing a Cinderella key for that service is more specific and more versatile, inasmuch as the client, or some trusted third party, can review the precise policy associated with the key.

Empirically, many past vulnerabilities have been due to bugs in X.509 certificate parsing and validation code, for example in their handling of ill-formed certificates, or their lax interpretation of certificate-signing flags, and each of those bugs required a full software patch. Our technical answer to this class of problem is to mostly generate the parsing and validation routines from high-level specifications; however, we note that this approach can be applied to the native validation code, and an interesting research direction is to certify the compilation process. In addition, we argue any (potential) bug in a Cinderella policy or its implementation would be easier to patch by updating the policy key, regardless of the variety of application implementations. Furthermore, after Cinderella's key generation phase, if the generation is done honestly, there is no longer any secret associated with the Cinderella keys, so they cannot be dynamically compromised. The fixed code used by the Cinderella verifier itself constitutes a smaller trusted computing base, used in a uniform manner, independent of the actual certificate contents. However, it also means that implementation-specific checks that depend on non-public values of the certificate are no longer possible.

E. Cinderella's Security

In Appendix A, we show that Cinderella is (almost) as secure as a system in which the certificate-chain validation code is performed by the verifier. Cryptographically, the argument relies on Pinocchio's proof-of-knowledge property. In other words, if Cinderella successfully verifies a proof, even one generated by a malicious prover, then given sufficient control of the prover and its randomness, a simulator can extract valid inputs to successfully run the `validate` function. Continuing with our example, we can thus reduce the security of Cinderella's S/MIME to the security of plain S/MIME signing and its PKI. The proof of privacy is straightforward and relies on Pinocchio's zero-knowledge property. A simulator that controls a trapdoor installed during Pinocchio parameter generation can create a proof without knowledge of the prover's private inputs to `validate`.

In practice, any deployment must preclude the existence of such a trapdoor, e.g., using the techniques of Ben-Sasson et al. [10]. In addition, one should carefully balance what is hidden by the proof and what is verified in the clear, to ensure compatibility with, e.g., Certificate Transparency [45] and the EFF's SSL Observatory [36].

IV. RSA Signature Verification

Cinderella supports the RSA PKCS#1v1.1 signature verification algorithm on keys of up to 2048 bits, coupled with the SHA1 and SHA256 hash functions. This combination of algorithms is sufficient to validate over 95% of recently issued certificate chains on the Web, according to recent PKI measurement studies [33, 35]. We assume all RSA certificates use the public exponent $e = 65537$, the only choice in practice.

To prove knowledge of a valid RSA signature, given as input a SHA digest h, an RSA modulus N, and the signature value s, we must show that

$$s^e \bmod N = \text{Padding}(h) \tag{1}$$

Depending on the application, either N or h may be a fixed input, i.e., a value known when we generate Cinderella keys. For a given modulus size, $\text{Padding}(h)$ is simply $h + P$ for some constant P. However, the arithmetic operations above operate on much larger numbers than the 254-bit prime used by Pinocchio's computations (§II-A); hence, the main challenge of this section is multi-precision QAP arithmetic.

We encode a big integer S as an array of n words ($S[j]$) of w bits each, such that $w < 254$ and $nw > 2048$. Thus, $S = \sum_{j=0}^{n-1} S[j]2^{jw}$. Inlining the standard square-and-multiply algorithm for computing the exponentiation in Equation (1) on the (sparse) binary decomposition of $e = 65537$, we can calculate the result recursively as follows

$$S_i = \begin{cases} s & \text{if } i = 0 \\ S_{i-1}^2 \bmod N & \text{if } 0 < i < 17 \\ sS_{i-1} \bmod N & \text{if } i = 17 \end{cases}$$

This gives us the result $s^e \bmod N = s^{65537} \bmod N = S_{17}$.

Instead of verifiably computing the S_i (which requires expensive modular reductions), we have the prover pre-compute them externally and provide them as private prover inputs during the verifiable computation. The goal of the validation program is then to verify that the values S_i were computed honestly. To this end, we perform the multi-precision squaring of steps 1 to 16 without propagating carries (that is, as a multiplication between formal polynomials $\sum S_i[j]x^j$):

$$\begin{aligned} C_i &= \textstyle\sum_{j=0}^{2n-1} \left(\sum_{k=0}^{j} S_{i-1}[k]S_{i-1}[j-k]\right) 2^{jw} \\ &= \textstyle\sum_{j=0}^{2n-1} C_i[j]2^{jw} \end{aligned}$$

For the final multiplication in step 17, the same formula is used but with $S_0[j-k]$ instead of $S_{i-1}[j-k]$. Observe that if the maximum width of $C_i[j]$, denoted $w' = 2w + [\log_2(w)] + 1$, is under 254 bits, and if we decompose inputs over $n' = 2n$ words (i.e., the n most significant words are 0), it becomes possible to compute $C_i[j]$ by using native Pinocchio additions and multiplications.

Still, the computed values $C_i[j]$ must be related to the untrusted input values $S_i[j]$. To verify that $C_i \bmod N = S_i$, we ask the prover to provide a value $Q_i = \sum_{j=0}^{n'} Q_i[j]2^{jw}$

```
big_copy(Si, s);
for(i=0; i < 17; i++) {
  if(i < 16) big_square(Ci, Si);
       else big_mul(Ci, Si, s);
  big_mul(Mi, Ni, Q[i]);
  big_sub(Di, Ci, Mi, compl); // compl = 2^{w'-w}
  check_eqmod(Di, S[i], R[i]);
  big_copy(Si, S[i]); }
```

Fig. 7. Cinderella code for modular exponentiation

```
void check_eqmod(bignum D, bignum S, bignum R){
  int i, j; Elem U, V;
  elem_init(U) elem_init(V);
  elem_copy(U, compl); // compl = 2^{w'-w}
    for(i=0; i < INPUT_WORDS; i++) {
      elem_add(U, U, D[i]);
      elem_mul(V, R[i], wf); // wf = 2^w
      elem_add(V, V, compl);
      elem_add(V, V, D[i]);
      elem_sub(U, U, V);
      zeroAssert(1-elem_eq_zero(U));
      elem_copy(U, R[i]);  }}
```

Fig. 8. Cinderella code for checking Equation (2)

such that $C_i - S_i = NQ_i$. The computations of NQ_i can also be carried out as words $M_i[j]$ of w' bits as before:

$$\begin{aligned} NQ_i &= \textstyle\sum_{j=0}^{2n-1} M_i[j]2^{jw} \\ &= \textstyle\sum_{j=0}^{2n-1} \left(\sum_{k=0}^{j} N_i[k]Q_i[j-k]\right) 2^{jw} \end{aligned}$$

Let $D_i[j] = C_i[j] - M_i[j]$. Since S_i and C_i are equal modulo N, $D_i[0]$ and $S_i[0]$ are equal on their w least significant bits. Furthermore, the most significant $w' - w$ bits of $D_i[0] - S_i[0]$, denoted $R_i[0]$, are such that the w least significant bits of $R_i[0] + D_i[1] - S_i[1]$ are all 0.

This propagation of carries leads to the following invariant:

$$R_i[j] + D_i[j+1] - S_i[j+1] = 2^w R_i[j+1] \tag{2}$$

While at first glance it appears that computing $R_i[j+1]$ from $D_i[j+1] - S_i[j+1]$ requires a division by 2^w, we instead assume the $R_i[j]$ are given as private prover inputs, and we verify their correctness with a (cheap) multiplication by 2^w.

The main fragment of the code that verifies the correctness of the S_i is shown in Figure 7. In particular, the function that verifies equation (2) is shown in Figure 8. A final concern in implementing Equations (1) and (2) is the handling of signed values. Our choice of w' allows one spare bit for encoding values $x < 0$ as $x + 2^{w'-w}$.

In our implementation, inputs are encoded as 36 words of 120 bits each (with the exception of the $R_i[j]$ words, which use 128 bits because of additive overflows). Note that it is necessary to verify that all prover inputs are within these bounds; otherwise the prover may be able to cheat using the overflows produced in the multiplications between inputs. The (omitted) code that performs this check using binary decompositions and that compares the final value of S_{17} to $h + P$ is straightforward.

V. ASN.1 FORMATTING & HASHING

To verify a signature on a certificate, we must first format the certificate's contents (roughly the concatenations of the binary encodings of all its fields) and compute a SHA digest.

1) SHA review: We rely on a new, custom C library for SHA1 and SHA256 tailored to Pinocchio. We omit the algorithm's details and only recall its structure. SHA1 and SHA256 take as input a variable number of bytes $x_0 \dots x_{n-1}$ and compute their fixed-sized cryptographic digest as follows:

- append to the input some minimal padding followed by the input length (in bits) to obtain a sequence of 64-byte blocks $B_0, \dots B_{N-1}$ (the padding and length prevent some input-extension attacks);
- iterate a cryptographic compression function f to hash each of these blocks together with an accumulator (starting from a constant block C) and finally return $h = f(\dots f(f(C, B_0), B_1) \dots\dots, B_{N-1})$.

2) Concatenating ASN.1 fields: Many X.509 fields have variable lengths, making their (verifiable) concatenation expensive. Recall that random access within arrays (using, in our case, indexes computed from the actual run-time lengths of fields in certificates) would require a potentially expensive encoding of memory. To improve performance (§VIII-A), we instead write custom, 'arithmetic' code for concatenations.

A direct, naïve implementation of concatenation As an example, consider concatenating two fields whose lengths range over $0..n-1$ and $0..m-1$, respectively. Assume those fields are stored in two byte arrays b and c of fixed lengths m and n, padded with 0s, with the actual length of the first field stored in variable ℓ. Using just comparisons, additions and multiplications, each byte of the resulting $m+n$ byte array may be computed as:

$$x_i = (i < n) * b_i + \textstyle\sum_{j=0}^{n} (j = \ell) * c_{i-\ell}$$

Although we may optimize this code, for instance by sharing sub-expressions, the concatenation still involves at least $(n+1)m$ quadratic equations. Worse, as we concatenate sequences of variable-length fields, the range of the result is the sum of the ranges of the inputs, making their concatenations increasingly expensive; this is problematic for ASN.1-formatted certificates, which typically include thousands of bytes and dozens of variable-length fields. Fortunately, we do not actually need to concatenate the entire certificate's contents into a single byte array to compute its digest.

Concatenating and Hashing We instead compute hashes incrementally, using a buffer of bytes to be hashed and carefully controlling the actual length of that buffer.

To leverage length annotations in the template as we generate the corresponding C program, we track precise bounds on the number of bytes available for hashing; this allows us to reduce the complexity of concatenating the certificate's bytes by emitting calls to SHA's compression function.

The main insight leading to an efficient implementation is that, by *conditionally* applying the compression function on partial concatenations, we can *reduce* the range of the remaining bytes to be hashed in the buffer, and hence the cost of the next concatenations. For instance, if we know (at compile-time) that the buffer currently holds between 5 and 123 bytes to be hashed, then by emitting code that hashes one block if there is at least 64 bytes, we know that the resulting buffer will hold between 0 and 63 bytes.

Another insight is that, by using Pinocchio's large words instead of bytes, we can minimize the number of variables that represent the hash buffer and the number of branches to consider for inserting a byte at a variable position.

Next, we explain our buffer implementation. Let x be an array of B 16-byte words, holding n bytes $c_0, \dots, c_{n-1}$ to be hashed. We encode x as:

$$x[i] = \begin{cases} \prod_{j=16i}^{j=16i+15} 256^{16i+15-j} * c_j & \text{for } i < n/16 \\ \prod_{j=16i}^{j=n} 256^{n-j} * c_j & \text{for } i = n/16 \\ 0 & \text{for } i > n/16 \end{cases}$$

and consider two functions that operate on this buffer:

- `append` requires that the buffer be not full ($n < 16B$); it adds one byte to it and increments n;
- `reduce` requires that the buffer contain at least 64 bytes; it calls the SHA compression function on the first 4 words of the buffer (`x[0],x[1],x[2],x[3]`) and the accumulator; it decrements n by 64 and shifts the buffer contents by 4 words (`x[0] = x[4]; ...`).

As we compile a template to C code, as illustrated in Figure 5, we emit a sequence of `append` and `reduce` calls that meet the requirements and preserves the invariant above, based on a (static) approximation of the range of values n may take at each step of the program at run time. More precisely, the template generator uses the variants `appendif` and `reduceif` that accept an additional boolean condition—the function does nothing if the condition is set to false. We emit calls to reduce whenever n is at least 64. This shifts the range, without changing its size. Otherwise, we emit a conditional `reduceif` call when the maximal value of n reaches the capacity of our buffer: this reduces the range by 64, but incurs the cost of a call to the compression function.

To finalize the hash, we 'flush' the buffer using similar conditional calls to ensure that $n < 55$; we then add minimal padding and the total length; and we return the digest obtained by calling the compression function one last time.

As a final example of 'arithmetic' programming, we include in Figure 9 the optimized code for *conditionally* appending one byte to the buffer as we concatenate variable-sized fields: if b is 1, then append c to x; otherwise do nothing. Note that our code uses multiplications by Boolean flags instead of conditionals (which are usually less efficient when compiling to arithmetic circuits). It also uses native operations on field elements (`Elem`) to operate on the buffer's 128-bit words.

VI. APPLICATION: TLS AUTHENTICATION

Transport Layer Security (TLS) is the most widespread cryptographic protocol on the Internet. It secures communications between billions of users and millions of HTTPS

```
typedef struct { int n, total; Elem x[B];} buffer;

void appendif(buffer* x, int c, int b) {
  Elem f, ce, z, t; (...)  // local field elements
  elem_set_ui(f,  b * 255);// conditional shift
  elem_set_ui(ce, b * c);  // conditional new byte
  int high = ((b * x->n) >> 4) & 31;
  for (int i = 0; i < B; i++) {
    // possibly add the byte to any word i
    elem_set_ui(t, (i == high));
    elem_mul(z, x->x[i], f);
    elem_add(z, z, ce);
    elem_mul(z, z, t); // no change when t = 0
    elem_add(x->x[i], x->x[i], z); }
  x->n += b;  x->total += b; }
```

Fig. 9. Cinderella code for conditionally hashing a byte

websites on a daily basis. It primarily relies on X.509 certificates to identify and authenticate both clients and servers. Server certificates are pervasive and have been the focus of many attacks and controversies (§II-B). Client certificates are optional, but widely deployed by large organizations and embedded in several national identity card schemes.

In the context of TLS, the need for a stronger PKI has been advocated [4, 49, 65], and improvements have been proposed in a patchwork, 'opt-in' fashion. Annoyingly, any proposed improvement must remain backward compatible with X.509 certificates issued many years ago.

Communicating Cinderella proofs instead of traditional X.509 certificate chains is a radical departure from existing proposals; we show how it improves the prover's privacy and the verifier's performance (by exchanging less data and checking small constant-size proofs), without any change to current CAs or the TLS standard.

Arguably, Cinderella also helps improves security once deployed (by embedding additional checks such as OCSP or Certificate Transparency [45] and mandating uniform application of certificate policies), without requiring additional bandwidth or changes at the client.

A. Approach: Pseudo-certificates

During TLS session establishment, certificate chains are treated as opaque byte arrays, encapsulated in specific handshake messages, and passed to a certificate manager to be validated and to extract the public key associated with the peer. Endpoint authentication is typically achieved by checking (using the key extracted from the endpoint certificate) a signature over some session-specific parameters (nonces and Diffie-Hellman parameters for server authentication; the transcript of protocol messages for client authentication).

With Cinderella, one could replace this signature by a proof of its knowledge, as illustrated in §III. However, such a design is impractical for two reasons. First, the proof would have to be computed online by the certificate holder during the handshake (as it depends on the session parameters), and thus, the connection would be significantly delayed due to the computational cost of building the proof. Second, the handshake message in which the signature is sent would have to be extended by introducing new cipher suites. To minimize

```
seq {
 tag<0>: const<2L>; # Version
 const<0L>; # Serial number
 seq { # RSA with SHA256
  const<01.2.840.113549.1.1.11>;
  const<null>; };
 seq { set { seq { # Issuer
  const<02.5.4.3>;
  const<printable:"Cinderella Pseudonym">;
 }; }; };
 # Validity period
 seq { var<date, pseudostart, 13, 13>;
   var<date, pseudoend, 13, 13>; };
 seq { set { seq { # Subject
  const<02.5.4.3>;
  const<printable:"Cinderella Pseudonym">;
 }; }; };
 seq { seq { # Elliptic curve key on NIST P256 curve
  const<01.2.840.10045.2.1>;
  const<01.2.840.10045.3.1.7>; };
 var<bitstring, pseudokey, 66, 66>; };
 tag<3>: seq { ... } # Basic mandatory extensions
}
```

Fig. 10. Template for the signed part of a TLS pseudonym

the disruption to TLS, we opt not to change the protocol, but rather to extend the associated certificate libraries.

Instead of proving knowledge of a signature on the protocol session, we leverage the modularity of X.509 by replacing existing certificate chains (owned by clients and servers) with short-lived *pseudo-certificates*. A pseudo-certificate combines an ephemeral public key pair with a Cinderella proof that the original chain has been verified to correctly connect to the pseudo-certificate. This proof can be computed offline; then, during the online TLS session establishment, the prover computes a standard signature using the private portion of the ephemeral key pair. The validator then checks both the signature and the Cinderella proof.

In more detail, a pseudo-certificate carries an ephemeral public key, a subset of the public attributes from the original certificate chain, and a Cinderella proof that the original chain has been verified to correctly connect to the pseudo-certificate. Within the pseudo-certificate, the Cinderella proof takes the place of the RSA or ECDSA signature typically found in a standard certificate. Figure 10 shows the concrete template for a bare-bones pseudo-certificate in which no attribute from the original chain is kept. Except for the unusual signature, pseudo-certificates are still well-formed X.509. They can be passed to TLS unchanged and cached in existing certificate stores. Their processing is relatively cheap (see §VIII).

Before running TLS, the owner of an endpoint certificate can prepare any number of pseudo-certificates (each associated with a freshly generated key pair) and compute Cinderella proofs that each pseudo-certificate indeed stands for the proper validation of the chain they replace. For instance, a web server may generate a fresh, short-lived pseudo-certificate every day, or a content provider may generate one pseudo-certificate for every server hosting its content for the day.

B. Security Enhancement: Revocation Checking

Certificate revocation has consistently failed to prevent misuse of compromised certificates (§II-B). Combining Cinderella

```
seq {
  tag<2>: const<octet string:X...>;  # KeyHash of responder
  var<gen date, producedAt, 15, 15>; # Timestamp
  seq { seq { seq {
    seq { const<01.3.14.3.2.26>; const<null>; }; # SHA1 OID
    const<octet string: X....>;# Hash of issuer's subject
    const<octet string: X...>; # Hash of issuer's public key
    var<int, ocspserial, 16, 20>; # Queried serial number
   }; const<0:"">; # Response status (0 = good)
   var<gen date, thisupdate, 15, 15>;
   tag<0>: var<gen date, nextupdate, 15, 15>; }; };
  tag<1>: # OCSP extensions
  seq { seq {
    const<01.3.6.1.5.5.7.48.1.2>; # OCSP nonce
    var<octet string, nonce, 18, 18>; }; };
}
```

Fig. 11. Template for the signed part of an OCSP proof

with OCSP can help optimize revocation checks and hence make them more widely used. A Cinderella policy can easily embed the validation of an OCSP proof, thus saving the cost for the verifier to fetch one from the CA, using an additional ASN.1 template to hash and verify. While TLS already features an extension for the server to attach an OCSP proof (OCSP stapling), with Cinderella, the server does not need to send it to the client, thus saving an additional 1 to 3KB of data in the handshake. Figure 11 illustrates a concrete OCSP template where we assume that both the OCSP responder certificate and the issuer of the certificate to verify are fixed in advance. The only variables in the OCSP proof are the timestamps, and the OCSP query nonce. In practice, CAs may use additional intermediates for their OCSP responder certificates; each such intermediary requires its own template. Besides OCSP, it is possible to verify other X.509 extensions as part of an application's validation policy. For instance, Certificate Transparency [45] offers signed proofs that a certificate has been included in a public, closely audited certificate log. Such proofs can be easily verified as part of an application's validation policy. More advanced schemes that assume mutually distrusting auditors of the certificate logs [4, 65] can similarly be embedded.

C. *Using Cinderella to Validate TLS Server Certificates*

To demonstrate Cinderella's support for large, complex application validation policies, we describe the steps we took to apply Cinderella to the validation policy that existing TLS clients apply to server certificate chains.

Building a complete certificate policy validator involves several templates, each of which gets compiled into a certificate-verifier function that loads (as private, prover inputs) the variable fields of the template, computes the hash of the certificate, and checks its signature. While these template verifier functions are automatically generated by Cinderella's template compiler, the application policy developer still must still manually 'chain' them together and write any application-specific checks on their variable fields (see §III-B).

Below, we summarize the top-level `validate` function that a TLS client typically applies to certificate chains it receives from the server (Figure 12 contains more detail).

```
#include "maincert.c" // Server certificate template
#include "ocsp.c"     // OCSP template
#include "pseudo.c"   // Pseudo-certificate template

// Checks whether the CN field of the subject matches hostname,
// or a SAN entry of type DNS matches hostname
void check_subject(hostname* host, subjectoid* soid,
                   subjectval* sval, sanentry* san);
int cmp_date(date d1, date d2);       // Compare dates
int cmp_serial(serial s1, serial s2); // Compare serial numbers

void validate(unsigned char* d, date t, bignum pseudokey,
              date pstart, date pend)
{
  bignum ca_key; bignum end_key; // Public keys (CA & endpoint)
  date start; date end; serial sn; // Validity interval; SN
  subjectoid soid[MAX_SUBJECT_FIELDS]; // Subject fields (keys)
  subjectval sval[MAX_SUBJECT_FIELDS]; // (values)
  sanentry san[MAX_SAN_ENTRIES]; // Subject alternative names
  // ... other variable fields

  // Load top-of-the-chain public key (e.g. root)
  load_Modulus("maincert", &ca_key, 0, COMPILE_TIME);

  // ... intermediate CA checks go here

  // Load private inputs; hash; verify signature with ca_key
  verify_maincert(ca_key, &sn, &start, &end, &soid[0],
                  &sval[0], &end_key, &san[0], /*...*/);

  date producedAt; serial ocsp_sn; // OCSP variables ...
  load_Modulus("ocsp", &ca_key, 0, COMPILE_TIME); // OCSP CA
  verify_ocsp(ca_key,  &ocsp_sn, &producedAt, /* ... */);
  assert(!cmp_serial(sn, ocsp_sn)); // Check SN in OCSP

  // Hash & check signature of pseudo-certificate
  // The variables in this template are **verifier** inputs
  verify_pseudo(end_key, pstart, pend, pseudokey);

  check_subject(d, soid, sval, san);     // Match domain name
  assert(!cmp_date(producedAt, pstart)); // Check dates
  assert(cmp_date(pstart, end));
  assert(!(cmp_date(pend, end)+1)); }
```

Fig. 12. Top-level validator for TLS clients, without intermediate certificates.

Cinderella outsources the execution of `validate` to the server, so the client only checks a succinct proof.

The `validate` function involves the following templates:

- one template for the endpoint certificate we replace, with additional templates for any intermediate CAs;
- one template for the OCSP proof (Figure 11), with additional templates for any OCSP intermediate certificates;
- one template for the pseudo-certificate (Figure 10).

Given the domain name d that the client expects to connect to and the current time t, the validator proceeds as follows:

1) Load the (static) public key of the "root" of the chain.
2) Hash and verify all potential intermediates, based on their templates, and the public key of their parent (either the root public key for the first intermediate, or the verified public key returned by the previous intermediate template verifier function).
3) Hash and verify the endpoint certificate (returning the assignment from the variable template fields).
4) Load the (static) public key of the "root" of the OCSP chain, unless it is one of the intermediate keys previously verified on the main chain.

5) Hash and verify all intermediates from the OCSP chain.
6) Hash and verify the OCSP proof, returning the timestamps and serial number it contains.
7) Check that the serial number in the OCSP proof is equal to the serial number of the endpoint certificate.
8) Hash and verify the pseudo certificate, taking as input the ephemeral key and validity time interval from the verifier; the signature is verified using the endpoint certificate's public key.
9) Check that the verifier's input domain d either matches the `Common Name` field of the subject or one of the `Subject Alternative Names` entries, taking into account wildcards, such as `*.a.com`.
10) Check that the verifier time t is within the validity intervals of every template.

The above steps are very close to what current browsers implement, except for the steps already enforced by our certificate templates. For instance, for a chain to be valid for TLS server authentication, a specific object identifier needs to appear in the *extended key usage* extension of all certificates in the chain. Extensions like the *basic constraints* specify the certificates that can be used to sign other certificates and the maximum length of a chain rooted at a given intermediate. The improper validation of these extensions have led to critical attacks (§II-B); in contrast, we encode all these checks in our certificate templates, whose conformance with browser validation policies can be easily tested—indeed, the original motivation for certificate templates was to evaluate their conformance with CA/Browser Forum's baseline requirements [33].

Delegation As short-lived Cinderella pseudo-certificates are used in place of the X.509 chain they represent in TLS, they can be handed to third parties (such as content delivery network operators) without exposing the corresponding certificate and its private key.

D. Security

After applying the extraction techniques from Cinderella's general proof of security, the only difference compared with existing certificate-chain validation is the additional signature verification introduced by the pseudo-certificate (Appendix A). The TLS proof then relies on the security of the signature scheme used in the pseudo-certificates.

Cinderella's zero-knowledge property also implicitly protects user privacy. The contents of the pseudo-certificate are constant, except for the freshly generated public key and the proof, and they do not contain private information.

VII. Application: Voter anonymity and eligibility in Helios

A. Helios (Review)

Classically, the privacy of users in elections, petitions, and surveys can be protected in two ways: (1) unlink users' input from their identities through a process of anonymous submission; or (2) compute the result from encrypted user inputs by exploiting homomorphic properties of the encryption scheme. These approaches are complementary: users may submit encrypted inputs anonymously.

The popular online voting system Helios [2] follows the second approach: its public election trail includes a list of identities and encrypted ballots for all participants. The Helios specification, however, notes that "in some elections, it may be preferable to never reveal the identity of the voters" and supports voter aliases for that purpose.[1] Such aliases are used, e.g., in IACR elections. Helios does not support any mechanism for authorizing anonymous voters to the voting server. Consequently, even if voter aliases are used over an anonymous communication system, the voting server is still able to link submitted ballots to user login credentials.

Helios expects an external mechanism to authenticate voters, and thus does not provide what Kremer et al. call *eligibility verifiability* [50]. From the verification trail, one cannot publicly check whether a ballot was cast by a legitimate voter. This enables ballot stuffing by the voting server, which may for instance wait until the end of the election and then inject ballots for all voters who have not participated. The use of voter aliases as suggested in the Helios specification makes the lack of eligibility verifiability even more problematic. Conversely, assuming voters are equipped with X.509 certificates and a trustworthy PKI, Helios may ask voters to sign their ballot, thereby cryptographically binding voter identities to ballots. This strengthens verifiability, but precludes the use of aliases.

B. Cinderella at the Polling Station

We design and implement a front-end extension of Helios, providing additional privacy and verifiability about who is actually voting, without affecting the core of the Helios protocol and the guarantees it already provides. Hence, we treat Helios ballots as opaque anonymous messages and, for each election, we ensure that a correct subset of anonymous ballots is passed to Helios for tallying:

1) Each voter contributes at most *one ballot of her choice*.
2) Only the election result and the *total number of ballots* are disclosed—not the identity of the actual voters.

Relying on Cinderella for access control and ballot authentication, we achieve *both* the same level of eligibility verifiability afforded by X.509 certificates *and* voter anonymity, even against fully corrupted election authorities. Neither the Helios servers nor the election audit trail contain useable information about who actually voted.

In more detail, relying on an existing X.509 PKI, we assume each voter is identified by some unique personal certificate, though the certificate need not be specific to voting and may have been issued for some other purpose. Below, we simply use the certificate subject as voter identifier; more generally, we may extract the identifier from other fields and perform some filtering, e.g. to check that the voter is at least 18.

With Helios, each election comes with a fresh identifier (EID) and a list of voters that may take part in the election. In principle, we could generate a fresh set of Cinderella keys for each election; Pinocchio, like Helios, supports distributed key generation [10], which can increase confidence in the

[1] http://documentation.heliosvoting.org/verification-specs/helios-v3-verification-specs

election policy (in particular, if the list of voters is fixed at compile time). For the sake of generality, we implement a generic policy for Helios that works for any election, taking as verifier inputs the EID and list of registered voters. We configure Helios voting servers to run in 'open election' mode with 'voter aliases': instead of using a fixed list of voters, the servers freely register new voter aliases (without any *a priori* authentication) and record their votes, each with a corresponding Cinderella proof, until the end of the election.

Given the election identifier, voter list, and a recorded triple of an alias (N), a ballot (B), and a proof (π), everyone can efficiently verify π to confirm that B is a ballot signed by some authorized voter for the election, and that N is that voter's unique alias for the election. Typically, the voting server will verify π before recording the vote, and an auditor will later verify the entire election log. Hence, although N and π do not reveal the voter's identity, multiple valid ballots from the same voter can be detected and eliminated before the normal Helios vote tallying begins.

We now detail the voting process and the meaning of its proof. Each voter computes her voter alias (N) for the election, prepares a ballot (B), produces a ballot signature (σ), and generates a Cinderella proof of knowledge π of both her certificate and the signature σ such that

1) the certificate subject (id) appears in the list of authorized voters for the election ($voters$);
2) σ is a valid signature on B with the certificate key (vk);
3) N is the result of a (fixed) function of the certificate key and the election identifier (EID).

The voter then anonymously contacts a Helios voting server for the election to register the alias N and cast her vote B with additional data π.

The third proof requirement is the most challenging, as we need N to be unique to each voter and each election (to prevent multiple voting) and to preserve the anonymity of the voter. If the signing key is embedded into a smartcard, we cannot simply use a secure hash of that secret. Instead, using the smartcard's usual API and the fact that PKCS#1 signing is deterministic, we ask for a signature on a reserved constant, e.g., `"Helios Seed"`, and use the resulting signature as a unique, unpredictable secret for the certificate's owner. Finally, we use that secret as the key of a pseudo-random function applied to the election description (including its identifier and voter list) to derive the unique alias N. Both the signature and the derivation of N are verifiable in zero-knowledge.

C. Implementation & Security Analysis

Figure 13 provides an overview of our top-level Helios verifier, using a certificate template for Estonian ID cards. In Appendix A-C, we model our Helios extension as a linkable ring signature scheme [53, 57, 66] and prove its security.

VIII. Performance Evaluation

To evaluate Cinderella's practicality, we measured its performance on micro- and macro-benchmarks. All experiments were performed on a Dell Precision 5810 workstation powered

```
#include "estonia.c" // Compiled template
void validate(char* EID, subject* IDs, hash* N, hash* B)
{
  pubkey ipk0, vk; subject id; signature sig0, sig1;
  load_Modulus(&ipk0, COMPILE_TIME); // Static
  verify_estonia(&ipk0, &vk, &id /* ... */);
  load_Signature(&sig0, RUN_TIME); // Prover input
  PKCSVerify(&vk, "Helios Seed", sig0);
  char x[276]; concat(pseudo, sig0.v, eid);
  zeroAssert(cmp_hash(N, sha1(pseudo)));
  filter(&id, IDs); // Checks that id is in IDs
  load_Signature(&sig1, RUN_TIME); // Prover input
  ballot_concat(x, "Helios Ballot", EID, IDs, B);
  PKCSVerify(&vk, x, sig1);
}
```

Fig. 13. Fragment of the concrete top-level verifier code for Helios

by an Intel Xeon E5-1620v3 3.5 GHz CPU with 16 GB of RAM and running Windows 10.

When reporting key generation times, we include compilation from C. For the verification times, we omit the overhead of loading and initializing the cryptographic engine, assuming that a Pinocchio verifier can be queried as a local service. In all cases, we measure single-threaded execution time, although we note that almost all steps are embarrassingly parallel.

Similar to prior work [59], the largest determinant of our key and proof generation performance is the number of quadratic equations produced when compiling our C programs. Evaluation keys use 320 bytes per equation; verification keys are much smaller, roughly the size of their public I/Os.

A. Micro-benchmarks

To better understand and predict Cinderella's costs, we measure the major components of certificate-chain validation: RSA signature verification (§IV), hashing (§V), and certificate generation from a template (§V).

RSA Key	Equations	KeyGen	ProofGen	Verify
Fixed	164,826	47.4 s	26.6 s	8 ms
Variable	180,774	47.4 s	31.0 s	8 ms

Fig. 14. RSA signature verification when we know the public key in advance (Fixed) and when we learn it online (Variable).

RSA Signature Verification The cost of generating a proof of signature verification depends on whether, when we compile and generate Cinderella keys, we know the RSA public key that will be used. If we do, e.g., when verifying an RSA signature using the public key of a root certificate, then all of the values associated with that key are constants and can be folded into Cinderella's key. If we only learn the RSA key at run time, e.g., when verifying an intermediate certificate, then the prover must perform additional proof work to incorporate it. In particular, such keys are represented as bytes in the certificate and must be converted to our high-precision arithmetic representation. We account for this extra step in the run-time signature verification costs.

Figure 14 summarizes our results for the two conditions using 2048-bit keys. During proof generation, Cinderella produces 58 KB of data representing the computed quotients, residues and carries.

	Equations/byte	KeyGen/byte	ProofGen/byte
SHA1	254.9 / B	377 ms / B	116 ms / B
SHA256	541.4 / B	112 ms / B	84 ms / B

Fig. 15. Costs to verify hashing, reported per byte hashed.

Hashing Figure 15 reports the costs of verifying the computation of the SHA1 and SHA256 hash functions, per input byte (unknown at key generation). Overall, each block (64 bytes) of SHA256 requires around 34.6K equations. Perhaps surprisingly, SHA256 performs better per byte than SHA1. The main distinction is that while SHA256 has a larger internal state, it only performs 64 iterations vs. SHA1's 80.

Since our complex applications (involving multiple intermediate CAs and OCSP proofs) need to hash 1–3 KB of data, in our macro-benchmarks below, we find that the total cost of hashing dominates the cost of formatting, RSA signature validation, and application-specific policies.

Complexity	Eqns/byte	KeyGen/byte	ProofGen/byte
0 B	17.0 / B	8.0 ms / B	3.8 ms / B
100 B	42.8 / B	15.8 ms / B	9.4 ms / B
200 B	51.4 / B	17.9 ms / B	9.5 ms / B
300 B	61.3 / B	19.0 ms / B	10.9 ms / B

Fig. 16. ASN.1 formatting costs per byte as a function of the template's complexity (size difference between the largest and smallest certificate).

ASN.1 Formatting The cost of ASN.1 formatting is highly dependent of the source template. In particular, it depends on the number of variable fields in the template, and on the difference between the upper and lower length bounds of these fields. As a metric, we define a template's *complexity* to be the difference between the maximum and minimum sizes of certificates that match it. In our experiment, starting from a fully constant (0 complexity) template for a typical 960-byte TLS server certificate, we increase its complexity by making more fields variable and by widening the range of the lengths of the variable fields until we reach a highly generic template. Figure 16 reports the results of this experiment for different complexities. The generated equations, key generation time and proof generation times are normalized with respect to the maximum size of a certificate that fits the template; hence, the table reports per-byte values.

While equations per byte increases with template complexity, note that even for relatively generic templates (allowing a total difference of 300 bytes between the smallest and largest certificate it covers), the cost of formatting is still only 11% of the cost of hashing. Hence, the maximum certificate size (and the total number of templates) are by far the most important factors for the prover. Also note that formatting using our custom concatenation routines use approximately 40-60 equations per byte, whereas generic memory techniques (§II-A) would require approximately 153 equations per byte for a 2048-byte certificate [68, Figure 5].

	Equations	KeyGen	ProofGen	Verify
Estonian EID	530,389	480 s	160 s	8 ms
S/MIME (SHA256)	967,740	252 s	152 s	8 ms
TLS server (SHA1)	547,940	496 s	165 s	8 ms
TLS server (SHA256)	858,855	219 s	137 s	8 ms
OCSP proof (SHA1)	267,135	174 s	60 s	8 ms
OCSP proof (SHA256)	357,878	85 s	58 s	8 ms
Pseudo-cert (SHA256)	367,488	84 s	61 s	8 ms

Fig. 17. Overall cost (formatting, hashing, signature verification) of certificate validation for various templates

Certificate Validation Combining all of the steps above, Figure 17 summarizes the overall cost of certificate validation for various types of templates. The reported costs include ASN.1 formatting, hashing, and RSA signature validation (assuming the signer's key is not known at compile time). Our tests were conducted on valid X.509 certificates obtained from various CAs, as detailed below.

For client credentials, we use a template based on a public test certificate from the Estonian ID system. This template is moderately constrained but also quite large (with a length range of 977 to 1130 bytes). Its main variable fields are in the subject (name, surname, and social security number). We also build a client certificate template based on the StartCom authority, intended to be used for S/MIME and TLS client authentication. This is also a rather large template (covering certificates from 1223 to 1399 bytes long) signed using a SHA256 hash, resulting in a large number of equations.

For server credentials, we use a TLS template based on the AlphaSSL authority. It is a relatively constrained template, allowing certificates sizes from 856 to 1128 bytes. The main variable fields of the template are the subject (which can include from 1 to 3 variable length fields) and the subject alternative names. We also evaluate the SHA256 version of this template, which is quite similar.

Lastly, we look at the OCSP proofs returned by the AlphaSSL CA and the pseudo-certificates we use for TLS. As these are both short and constant-length, their templates are significantly faster to check than other certificates.

B. Macro-benchmarks

Figure 18 summarizes our evaluation of the complete certificate validation policies for our applications in §VI and §VII.

1) TLS: Recall from §VI that our TLS application involves many templates: one for the endpoint certificate, one for the OCSP certificate, one for the pseudo-certificate, and optionally, several more for any intermediates included in the chain. Furthermore, the TLS policy also performs hostname validation and expiration checks.

According to the 2010 Qualys SSL survey [60], based on a sample of 600,000 trusted chains, 44% of sites use one intermediate CA certificate, 55% use two, while the remaining 1% use even longer chains. Thus, in our experiments, we vary the number of intermediate CAs from 0–2.

Application	CAs	Equations	KeyGen	ProofGen	Verify
TLS SHA256	2	3,033,071	697 s	531 s	9 ms
TLS SHA256	1	2,314,811	530 s	421 s	9 ms
TLS SHA256	0	1,588,340	411 s	266 s	9 ms
TLS SHA1	0	1,095,386	1,207 s	328 s	9 ms
Helios	0	434,177	196 s	90 s	8 ms

Fig. 18. Evaluation of complete application policies with varying numbers of intermediate CAs.

As shown in Figure 18, for our most general policy (with two intermediate CAs, using SHA256), it takes the prover nine minutes (offline) to create a single pseudo-certificate. On the other hand, the verifier (e.g. a web browser) can verify the Cinderella proof contained in the pseudo-certificate in 9 ms.

The comparison of TLS handshake performance using the pseudo-certificate vs. the original chain depends on the client and server configuration for the baseline. For instance, applying OCSP with Cinderella removes the bandwidth and online computational overhead of OCSP stapling. In terms of raw signature performance, the cost of natively hashing and verifying the signatures in the certificate chain is comparable to the time to verify Cinderella's proof. In terms of bandwidth, typical RSA certificate chains with one intermediate take 2 to 3 KB, with one additional KB per extra intermediate. Even ECDSA certificates currently issued by CAs are typically over 700 bytes long. Pseudo-certificates, in contrast, are a flat 564 bytes. Thus, Cinderella improves on bandwidth by 1.2–5.4$\times$ even for short chains; this gain scales with the number of additional intermediates. The overall performance impact of this gain depends on the application protocol: in HTTPS, request pipelining and session resumption amortize these savings.

2) Helios: For our Helios application, we use the Estonian identity card template with no further intermediates. Although an OCSP service is provided, we do not believe checking revocation as part of the Cinderella policy is useful, as we support a per-election registered voter list. The voter pseudonym computation and ballot signature are otherwise implemented as described in §VII. We use as a voter identifier the social security number found in the subject of the certificate.

Since our policy only needs to verify two RSA signatures, the computational costs for Helios (listed in Figure 18) are much smaller than for TLS: it only takes a minute and a half to build a proof of the ballot's signature.

Although our tests were performed on small voter lists, our approach would scale up to lists with millions of voters represented as a Merkle tree using an algebraic hash function [16], at a negligible cost compared with the two RSA signature verifications.

Tallying an election now requires the Helios servers (or anyone who wishes to verify the election) to check all Cinderella proofs attached to all the ballots. At 8 ms per proof verification, we are able to verify over 120 ballots per second, which greatly exceeds the tallying capacity of Helios (reported to be around 7 ballots per second just for decryption [2]).

IX. Related Work

We refer to §II-A for related work on general-purpose verifiable computation. Although recent work provides substantial cryptographic implementations and claims 'near-practicality', few real-world applications have been attempted. The most notable exception is privacy-enhanced variants of Bitcoin [7, 31]. Several papers also evaluate simple MapReduce and data processing applications, but proof-generation overhead is a significant bottleneck [3, 16, 27, 30].

We refer to §II-B for related work on X.509 certificates and PKI. The use of zero-knowledge proofs in public-key infrastructures was pioneered by Chaum [24] and Brands [15]. Wachsmann et al. [67] extend TLS with anonymous client authentication by integrating an anonymous-credential-based signature scheme directly into TLS using a custom extension. Camenisch et al. [21] extend federated identity management protocols with anonymous credentials based on [19]. Our approach differs from classic anonymous credentials and other custom PKI elaborations [4, 49, 65], as we do not rely on the cooperation of CAs to deploy Cinderella, and we only change the usage of plain, existing certificates.

Regarding voting protocols, Kremer et al. [50] distinguish between individual, universal, and eligibility verifiability and note that Helios 2.0 does not guarantee eligibility verifiability and is vulnerable to ballot stuffing by dishonest administrators. Cortier et al. [29] address this problem by adapting the Helios protocol. They envision an additional registration authority that generates signature key pairs for users that then sign their ballots. This corresponds to using X.509 certificates directly to sign the ballot and does not allow for voter anonymity. Springall et al. [62] analyzed the security of Estonia's online elections and noted their lack of end-to-end verifiability.

X. Conclusion

We propose, implement, apply, and evaluate a radically different use of existing X.509 certificates and infrastructure. Taking advantage of recent advances in cryptographically verifiable computation, we outsource the enforcement of flexible X.509 certificate validation policies from certificate verifiers to certificate owners, thereby simplifying the task of the verifier and improving the privacy of the owner. Our prototype implementation supports complex policies involving multiple certificates and application checks. It includes a template compiler and carefully-crafted libraries to fit standard-compliant X.509 processing within the constraints of verifiable computation.

Our applications to TLS and electronic voting show excellent performance for the verifier and substantial overhead for the prover. Cinderella is applicable when policies can be evaluated and turned into proofs offline, or when the burden of producing a proof can be amortized by many faster verifications. It is not a full replacement for X.509, but it already enables the deployment of new, interesting policies, and offers a graceful integration of old and new cryptography.

Acknowledgements

The authors thank the anonymous reviewers for insightful comments, and Joseph Bonneau for shepherding the paper.

References

[1] M. Abe, G. Fuchsbauer, J. Groth, K. Haralambiev, and M. Ohkubo. Structure-preserving signatures and commitments to group elements. In *Proc. of CRYPTO*, 2010.

[2] B. Adida. Helios: Web-based open-audit voting. In *Proc. of USENIX Security*, 2008.

[3] M. Backes, D. Fiore, and R. M. Reischuk. Nearly practical and privacy-preserving proofs on authenticated data. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2015.

[4] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski. ARPKI: Attack resilient public-key infrastructure. In *Proc. of ACM CCS*, 2014.

[5] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Innovations in Theoretical Computer Science (ITCS)*, Jan. 2013.

[6] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proc. of CRYPTO*, 2013.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

[8] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proc. of CRYPTO*, 2014.

[9] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proc. of USENIX Security*, 2014.

[10] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2015.

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

[12] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen. On the existence of extractable one-way functions. In *Proc. of ACM Symposium on the Theory of Computing (STOC)*, 2014.

[13] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[14] M. Blum, A. D. Santis, S. Micali, and G. Persiano. Noninteractive zero-knowledge. *SIAM J. on Computing*, 20(6), 1991.

[15] S. Brands. *Rethinking Public Key Infrastructures and Digital Certificates*. MIT Press, 2000.

[16] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proc. of the ACM SOSP*, 2013.

[17] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

[18] CA/Browser Forum. Baseline requirements for the issuance and management of policy-trusted certificates, v.1.1.5.

[19] J. Camenisch and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *Proc. of EUROCRYPT*, 2001.

[20] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *Proc. of the Conference on Security in Communication Networks (SCN)*, 2002.

[21] J. Camenisch, T. Groß, and D. Sommer. Enhancing privacy of federated identity management protocols: anonymous credentials in ws-security. In *Proc. of the ACM Workshop on Privacy in the Electronic Society (WPES)*, 2006.

[22] J. Camenisch, S. Mödersheim, G. Neven, F. Preiss, and D. Sommer. A card requirements language enabling privacy-preserving access control. In *Proc. of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2010.

[23] Canadian Institute of Chartered Accountants. WebTrust for certification authorities.

[24] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[25] D. Chaum and E. van Heyst. Group signatures. In *Proc. of EUROCRYPT*, 1991.

[26] Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proc. of the ACM Symposium Foundations of Software Engineering*, 2015.

[27] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *Proc. of EUROCRYPT*, Apr. 2015.

[28] J. Clark and P. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2013.

[29] V. Cortier, D. Galindo, S. Glondu, and M. Izabachène. Election verifiability for Helios under weaker trust assumptions. In *Proc. of ESORICS*, 2014.

[30] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2015.

[31] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: Building Zerocoin from a succinct pairing-based proof system. In *ACM PETShop*, 2013.

[32] A. Delignat-Lavaud and K. Bhargavan. Network-based origin confusion attacks against HTTPS virtual hosting. In *Proc. of the ACM Conference on World Wide Web (WWW)*, 2015.

[33] A. Delignat-Lavaud, M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie. Web PKI: closing the gap between guidelines and practices. In *Proc. of the ISOC NDSS*, 2014.

[34] C. Diaz, E. Kosta, H. Dekeyser, M. Kohlweiss, and G. Nigusse. Privacy preserving electronic petitions. *Identity in the Information Society*, 1(1):203–209, 2009.

[35] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *Proc. of ACM Internet Measurement Conference (IMC)*, 2013.

[36] Electronic Frontier Foundation. The EFF SSL Observatory. https://www.eff.org/observatory, 2010.

[37] European Telecommunications Standards Institute. Policy requirements for certification authorities issuing public key certificates.

[38] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for HTTP. Technical report, 2015.

[39] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proc. of ACM Conference on Computer and Communications Security*, 2012.

[40] D. Fiore and A. Nitulescu. On the (in)security of SNARKs in the presence of oracles. Cryptology ePrint Archive, Report 2016/112, Feb. 2016.

[41] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. of CRYPTO*, 2010.

[42] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proc. of EUROCRYPT*, 2013.

[43] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proc. of ACM Symposium on the Theory of Computing (STOC)*, 2011.

[44] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proc. of ACM Conference on Computer and Communications Security*, 2012.

[45] Google. Certificate transparency. https://sites.google.com/site/

certificatetransparency/.

[46] J. Groth and A. Sahai. Efficient non-interactive proof systems for bilinear groups. *SIAM Journal on Computing*, 41(5), 2012.

[47] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of USENIX Security*, 2012.

[48] J. Kasten, E. Wustrow, and J. A. Halderman. Cage: Taming certificate authorities by inferring restricted scopes. In *Financial Cryptography and Data Security*. 2013.

[49] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Transparent Key Integrity (TKI): A Proposal for a Public-Key Validation Infrastructure. Technical Report CMU-CyLab-12-016, Carnegie Mellon University, July 2012.

[50] S. Kremer, M. Ryan, and B. Smyth. Election verifiability in electronic voting protocols. In *Proc. of ESORICS*, 2010.

[51] A. Langley. Revocation still doesn't work. https://www.imperialviolet.org/2014/04/29/revocationagain.html, Apr. 2014.

[52] J. Liang, J. Jiang, H. Duan, K. Li, T. Wan, and J. Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

[53] J. K. Liu, V. K. Wei, and D. S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Proc. of the Australasian Conference on Information Security and Privacy (ACISP)*, 2004.

[54] Y. Liu, W. Tome, L. Zhang, D. R. Choffnes, D. Levin, B. M. Maggs, A. Mislove, A. Schulman, and C. Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *Proc. of the ACM Internet Measurement Conference*, 2015.

[55] M. Marlinspike. More tricks for defeating SSL in practice. Black Hat USA, 2009.

[56] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[57] T. Nakanishi, T. Fujiwara, and W. H. A linkable group signature and its application to secret voting. *Transactions of Information Processing Society of Japan*, 40(7), 1999.

[58] Netcraft. SSL survey. http://www.netcraft.com/internet-data-mining/ssl-survey/, May 2013.

[59] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2013.

[60] I. Ristic. Internet SSL survey. *Black Hat USA*, 3, 2010.

[61] R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret: Theory and applications of ring signatures. In *Theoretical Computer Science, Essays in Memory of Shimon Even*, 2006.

[62] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman. Security analysis of the Estonian Internet voting system. In *Proc. of ACM Conference on Computer and Communications Security*, Nov. 2014.

[63] E. Stark, L.-S. Huang, D. Israni, C. Jackson, and D. Boneh. The case for prefetching and prevalidating TLS server certificates. In *Proc. of the ISOC NDSS*, 2012.

[64] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Proc. of CRYPTO*, 2009.

[65] P. Szalachowski, S. Matsumoto, and A. Perrig. PoliCert: Secure and flexible TLS certificate management. In *Proc. of ACM Conference on Computer and Communications Security*, 2014.

[66] P. P. Tsang and V. K. Wei. Short linkable ring signatures for e-voting, e-cash and attestation. In *Information Security Practice and Experience Conference (ISPEC)*, 2005.

[67] C. Wachsmann, L. Chen, K. Dietrich, H. Löhr, A. Sadeghi, and J. Winter. Lightweight anonymous authentication with TLS and DAA for embedded mobile devices. In *Information Security Conference (ISC)*, 2010.

[68] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proc. of the ISOC NDSS*, Feb. 2015.

[69] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. http://eprint.iacr.org/.

[70] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2013.

Appendix A
Proofs of Security

In this appendix we explain our cryptographic security arguments for the Cinderella protocol transformation and its applications to S/MIME, TLS, and Helios. For Helios we also express our notion of security as linkable ring signatures.

First, we restate *knowledge soundness* of proofs-of-knowledge in a style more amenable to splitting a larger system into those parts that rely on verifiable computations and those that do not.

- The adversary consists of two parts $\mathcal{A}_1$ and $\mathcal{A}_2$. $\mathcal{A}_1$ runs on input 1^λ before $\mathbf{KeyGen}$ and generates F and auxiliary input z.
- Then $\mathbf{KeyGen}$ runs, and $\mathcal{A}_2$ is passed EK_F, VK_F, z, and randomness r.

A proof system is knowledge-sound if for every *benign* $\mathcal{A}_1$ and every PPT $\mathcal{A}_2$ that outputs a verifying y, π_y with some probability, there exists an extractor $\mathcal{E}$ that when run on the same input as $\mathcal{A}_2$, including r, produces u, w such that $y = F(u, w)$ with almost the same probability. The randomness is taken over the choices of $\mathcal{A}_1$, $\mathbf{KeyGen}$, and r.

The benign restriction arises from the possibility of $\mathcal{A}_1$ providing an obfuscator as part of z that creates a proof from which $\mathcal{E}$ cannot extract [12].

The auxiliary input z may for instance contain the signatures of a PKI. It, together with its benign restriction, may however not always be sufficient in settings in which certificates and signatures are generated on the fly and the adversary $\mathcal{A}_2$ has access to a signing oracle. This setting was recently analyzed by Fiore and Nitulescu [40] who introduced the notion of *$\mathcal{O}$-SNARKs*, which allow us to assume the existence of extractors even against more powerful adversaries that have access to oracles $\mathcal{O}$. In terms of the definitions above, this means that adversary $\mathcal{A}_2$ is given access to signing oracles $\mathcal{O}$. We conjecture that Pinocchio [59] is an *$\mathcal{O}$-SNARK* with signing oracles if the knowledge assumption underlying Pinocchio holds against adversaries with access to oracles $\mathcal{O}$.

A. Security of Cinderella Generic: Exemplary for S/MIME

Our general approach is to prove that Cinderella, when employed in a system X, is (almost) as secure as system X in which the certificate-chain validation is performed at the point of signature verification.

Cryptographically, the argument relies on *$\mathcal{O}$-SNARK* knowledge soundness, since emails can be signed after Cinderella key generation. Alternatively, we may modify our scheme to communicate both the signature, the verification modulus, and a proof validating the chain for this modulus.

We refer to the system in which Cinderella is used to extend X as $\tilde{X}$. In the first step of the proof, we split the system $\tilde{X}$ into four parts: the part $\mathcal{A}_1$ of the system that is executed before the generation of Pinocchio keys; the signing authorities $\mathcal{O}$ that provide certificates and signatures generated after the generation of Pinocchio keys; the adversary $\mathcal{A}_2$ that generates the proof π; and the verifier that verifies proofs.

For every pair $(\mathcal{A}_1, \mathcal{A}_2)$, we then consider a different experiment in which we make use of the extractor $\mathcal{E}$ which exists given the *$\mathcal{O}$-SNARK* knowledge soundness. In this experiment, we abort whenever $\mathcal{E}$ fails to extract inputs such that `validate` succeeds but the proof verifies. The difference between the success probabilities of $(\mathcal{A}_1, \mathcal{A}_2)$ in the original experiment and the new experiment is bounded by the *$\mathcal{O}$-SNARK*'s knowledge soundness.

We are now in a position to reduce the security of $\tilde{X}$ to the security of X. We assume that part $\mathcal{A}_1$ executed before the generation of Pinocchio keys and that the signing authorities $\mathcal{O}$ are unchanged. We define the adversary $\mathcal{A}'$ to include the generation of Pinocchio keys, as well as algorithms $\mathcal{A}_2$ and $\mathcal{E}$. $\mathcal{A}'$ also continues to query $\mathcal{O}$. Instead of outputting the proof π, $\mathcal{A}'$ outputs valid inputs for `validate` which break the security of X whenever they break the security of $\tilde{X}$.

We conjecture that the security of S/MIME where the verifier runs the `validate` function reduces to INT-CMA security for PKCS#1 signatures.

B. Security of Cinderella for TLS

The security argument for TLS follows the generic argument, except that we have to consider the additional signature verification introduced by the pseudo-certificate.

We can thus apply the generic security argument to reduce the security of TLS with Cinderella to a system X' in which pseudo-certificates are verified locally by the verifier. It remains to be shown that this adapted system, which now no longer involves SNARKs is secure. Because of the addition of the pseudo-certificate and its signature verification step, any security proof for the TLS protocol and its PKI would need to be extended. As pseudo-certificates are used only once, or at most briefly and for a specific purpose, we argue that such an extension of the certificate chain, although non-standard, can be soundly reduced to the INT-CMA security of PKCS#1 signatures. Most existing security proofs for TLS (e.g., [11]) assume the PKI provides honest keys and are thus unaffected by this change. The formal soundness of the X509 PKI as used by TLS, however, is much in doubt; to our knowledge no realistic end-to-end formal treatment has been attempted.

C. Security of Cinderella for Helios

In addition to the *$\mathcal{O}$-SNARK* knowledge soundness of Pinocchio, we will need an additional assumption on the pseudo-randomness of hashed PKCS#1 signatures. Consider the following game.

Definition 1 (Hash Pseudo-randomness): A signature scheme $\texttt{PKCSGen}, \texttt{PKCSSign}, \texttt{PKCSVerify}$ is hash pseudo-random if for all probabilistic polynomial time adversaries $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l}(vk, sk) \leftarrow \texttt{PKCSGen}(1^\lambda);\\ b \leftarrow \{0,1\}:\\ b = \mathcal{A}^{\texttt{PKCSSign},\texttt{F}}(vk)\end{array}\right] \approx \frac{1}{2},$$

where

- $\texttt{PKCSSign}(m)$ calls $\texttt{PKCSSign}(sk, m)$ if m does not start with the prefix `"Helios Seed"`.
- $\texttt{F(EID)}$ returns, depending on b, either the result of calling $\texttt{H}(\texttt{PKCSSign}(sk, \texttt{"Helios Seed"}||EID))$ or a random function with the same input and output domains.

We model our e-voting extension as a linkable ring signature scheme [53, 57, 66]. For each election, the authorized voters can sign anonymously once. Subsequent signatures are linkable to the same signer and can thus be filtered out. Our scheme has 4 algorithms and models legacy key usage:

- $(ipk, isk) \leftarrow \texttt{Setup}(1^\lambda)$. Generates public parameters ipk available to all users and a private issuer key isk.
- $usk \leftarrow \texttt{Reg}^U(ipk, id) \leftrightarrow \texttt{Reg}^I(isk, id)$. Generates and registers a user signing key for identifier id. For honest registration, we write $usk \leftarrow \texttt{Reg}(ipk, isk, id)$.
- $(\pi, N) \leftarrow \texttt{Sign}(usk, EID, IDs, B)$. Signs the message B with respect to ring (EID, IDs) and returns signature π. EID is the ring identifier and corresponds to the election identifier in our election setting. IDs is the set of identities allowed to sign, sometimes called the ring. B is the message to be signed, in our case the ballot. N is a unique, pseudo-random pseudonym computed from EID and usk. It makes repeated signatures linkable while protecting the signer's identity.
- $\{0,1\} \leftarrow \texttt{Verify}(ipk, EID, IDs, B, \pi, N)$. Verifies the ring signature.
- $\sigma \leftarrow \texttt{Legacy}(usk, m)$. Generates a legacy signature.

We discuss how these algorithms are employed in our Helios front-end extension. `Setup` is run once the Cinderella voting application policy is agreed on. The keys ipk, isk include the Pinocchio verification and evaluation keys, as well as the certificate issuer's public and private keys respectively. One could split `Setup` into two algorithms to isolate the legacy X.509 keys, or refine it with an explicit X.509 template. The `Reg` protocol models certificate issuance. For Cinderella, the ids correspond to the subject identifier encoded in X.509 certificates. The value usk contains both the user's certificate and his RSA private keys. `Sign` corresponds to the vote submission process of our front-end, while `Verify` is used for ballot validation.

The linkable ring-signatures literature already discusses similar voting applications [53, 57, 66]. However, they often require a freshly generated user key for each election, while we reuse long-term legacy keys. A similar primitive was also employed in [34] to implement a primitive anonymous petition system. Here we achieve the same security guarantees but piggyback on the client certificates of National ID cards, which is very appealing for e-government scenarios.

We formally define correctness, unforgeability, and unlinkability properties of linkable ring signatures and prove that they are met by our construction.

Security definitions The scheme $\mathcal{R} = (\texttt{Setup}, \texttt{Reg}, \texttt{Sign}, \texttt{Verify}, \texttt{Legacy})$ is a linkable ring signature scheme if it is *correct*, *unforgeable* and *anonymous*, as defined next.

Users may sign any messages in any ring they belong to.

Definition 2 (Correctness): $\mathcal{R}$ is correct if, for all adversaries $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \texttt{Setup}(1^\lambda);\\ usk \leftarrow \texttt{Reg}(ipk, isk, id)\\ (EID, IDs, B) \leftarrow \mathcal{A}(ipk)\\ (\pi, N) \leftarrow \texttt{Sign}(usk, EID, id \cup IDs, B):\\ \texttt{Verify}(ipk, EID, id \cup IDs, B, \pi, N) = 1\end{array}\right] = 1.$$

When defining unforgeability, we give the adversary access to Corrupt queries that reveal user secret keys and Legacy queries that request the use of usk in legacy algorithms, paradigmatically PKCS#1 signing. Intuitively, $\mathcal{R}$ is unforgeable (with respect to insider corruption and legacy algorithms `Legacy`) if an adversary cannot create signatures with respect to more one-time pseudonyms than he controls.

Definition 3 (Unforgeability): $\mathcal{R}$ is unforgeable when for all probabilistic polynomial-time adversaries $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \texttt{Setup}(1^\lambda)\\ EID, IDs, \Pi \ \leftarrow \mathcal{A}^{\texttt{Reg},\texttt{Legacy},\texttt{Sign},\texttt{Corrupt}}(ipk):\\ \neg\texttt{Cond}(EID, IDs, \Pi) \wedge \forall(N, B, \pi) \in \Pi.\\ \qquad\qquad \texttt{Verify}(ipk, EID, IDs, B, \pi, N) = 1\end{array}\right] \approx 0,$$

where

- $\texttt{Reg}(i, id)$ checks that $id \notin \mathcal{I}$, otherwise aborts; adds id to set $\mathcal{I}$; if $i = \perp$, runs $\texttt{Reg}^I$ with the adversary (enabling it to register his own identifiers); otherwise runs $usk_i \leftarrow \texttt{Reg}(ipk, isk, id)$, adds usk_i to set $\mathcal{C}$ and id to set $\mathcal{H}$.
- $\texttt{Legacy}(i, m)$ calls the $\texttt{Legacy}(usk_i, m)$ signing algorithm if usk_i exists.
- $\texttt{Sign}(i, EID, IDs, B)$ returns $(\pi, N) \leftarrow \texttt{Sign}(usk_i, EID, IDs, B)$, provided usk_i has been generated by Reg and was not leaked using $\text{Corrupt}(i)$. The oracle records (EID, IDs, B) in a set $\mathcal{T}$.
- $\texttt{Corrupt}(i)$, provided usk_i has been generated by $\texttt{Reg}(i, id)$, returns usk_i and removes id from $\mathcal{H}$.
- $\texttt{Cond}(\Pi)$ holds when there is an injective function ϕ from the names N recorded in Π to $IDs \cap \mathcal{I}$ such that

$$\forall(N, B, \pi) \in \Pi \,.\, \phi(N) \in \mathcal{H} \Rightarrow (EID, IDs, B) \in \mathcal{T}.$$

Anonymity means that signatures by different users in the same ring on the same message have the same distribution.

Definition 4 (Anonymity): $\mathcal{R}$ is anonymous when, for any adversary $\mathcal{A}$, we have

$$\Pr\left[\begin{array}{l}(ipk, isk) \leftarrow \texttt{Setup}(1^\lambda);\\ (EID, i_0, i_1, IDs, B) \leftarrow \mathcal{A}^{\texttt{Reg},\texttt{Legacy},\texttt{Sign}}(ipk)\\ b \leftarrow \{0, 1\};\\ (\pi, N) \leftarrow \texttt{Sign}(usk_{i_b}, EID, IDs, B):\\ \mathcal{A}(\pi, N) = b \mid \texttt{Cond}(EID, i_0, u_1)\end{array}\right] \approx \frac{1}{2},$$

where $\texttt{Cond}(EID, i_0, i_1)$ holds if

- usk_{i_0} and usk_{i_1} have been honestly generated by calling $\texttt{Reg}(i_0, id_{i_0})$ and $\texttt{Reg}(i_1, id_{i_1})$;
- $id_{i_0}, id_{i_1} \in IDs$; and
- neither (EID, i_0) nor (EID, i_1) were queried to `Sign`.

On realizing the algorithms Figure 19 realizes the algorithms $(\texttt{Setup}, \texttt{Reg}, \texttt{Sign}, \texttt{Verify})$ using a verifiable computation system for a function $\texttt{validate}(ipk0, EID, IDs, N, B)$ whose concrete code is partially shown in Figure 13, and any ordinary INT-CMA signing scheme. Pragmatically, we will assume that PKCS#1 is INT-CMA secure.

`Setup` creates an issuer key pair $ipk0, isk0$ and an evaluation key pair $ipk1, isk1$ for certificates of fixed template and issuer public key. `Reg` is just the legacy issuing process for the users' X.509 certificates. Certificates of eligible voters must match the template fixed in `Setup`. `Sign` computes the inputs for `validate`, a pseudonym N derived from σ_{id} and the signature σ on (EID, B), and a proof that they satisfy the computation using EK. `Verify` checks the proof.

We instantiate $\texttt{Legacy}(usk, m)$ by PKCS#1 signing, but messages with the prefix `"Helios"` are never signed.

Theorem 1: This realization is correct, anonymous, and unforgeable, if PKCS#1 is INT-CMA secure and hash pseudo-random, and Pinocchio is *$\mathcal{O}$-SNARK* knowledge sound.

Proof sketch:

Correctness follows from inspection.

Anonymity follows from the perfect zero-knowledge property of Pinocchio, and hash pseudo-randomness of PKCS#1. The proof proceeds in two steps. First, we replace real Pinocchio proofs by simulated proofs. Second, we replace pseudonyms N by random values. The first step is justified by the zero-knowledge property of Pinocchio and the second by hash pseudo-randomness.

Unforgeability relies on the *$\mathcal{O}$-SNARK* knowledge soundness of Pinocchio, which allows extraction of valid signatures from the proof. Either extraction fails and we break the security of Pinocchio, or we obtain values $cert[id], \sigma_{id}, \sigma$ such that the certificate is valid and the signatures verify with respect to the subject public key in $cert$. If the certificate was not generated by `Reg` or if for any id in $\mathcal{H}$, σ signs a hitherto fresh message (EID, B) we give a reduction to the unforgeability of PKCS#1 signatures.

```
Setup(){
  ipk0, isk0 = X509Setup();
  store_Modulus(ipk0);
  EK, VK = KEYGEN(validate);
  return ({ipk0, EK, VK}, isk0); }
Reg(ipk, isk, id){
  vk, sk = PKCSGen();
  cert = X509Issue(isk0, vk, id);
  return {ipk, cert, sk}; }
Legacy(usk{ipk, cert, sk}, T) {
  assert(notPrefix("Helios", T));
  return PKCSSign(sk, T);}
Sign(usk{ipk, cert, sk}, EID, IDs, B){
  sig0 = PKCSSign(sk, "Helios Seed");
  sig1 = PKCSSign(sk, "Helios Ballot"||EID||\IDs||B);
  N = H(sig0||EID);
  π = COMPUTE(EK, EID, IDs, N, B, cert, sig0, sig1);
  return N,π; }
Verify(ipk, EID, IDs, B, N, π){
  return VERIFY(VK, EID, IDs, N, B, π); }
```

Fig. 19. Pseudocode for realizing our linkable ring-signature scheme $\mathcal{R}$