Direct Anonymous Attestation on a Secure Coprocessor

Experience Report

Eslam Elnikety Edit Kapcari Vineet Rajani David Swasey

April 7, 2013

Abstract

We describe the Direct Anonymous Attestation protocol and our experience implementing it on the IBM 4758 family of cryptographic coprocessors.

1 Introduction

We implemented Direct Anonymous Attestation (DAA) on top of a member of the IBM 4758 family of cryptographic coprocessors [1, 6].

Attestation is a core concept in so-called trusted computing. It enables trust decisions based on the identity of a machine and its software stack. Why identity? Consider the following scenario. Suppose server S houses content C and wants to only ever release C to a trusted application A; for example, S might supply music online, C might comprise songs, and A might be a trusted application that lets its users listen to—but not copy—their purchases. Certainly, S would employ cryptographic protocols to protect its content during download, but what about *after* download? How does S ensure it's really talking to A? Moreover, how does it ensure that no other software running alongside A makes an illicit copy of C? Attestation attempts to mitigate such concerns; for example, prior to sending C, the server S might identify all of the software running alongside A to ensure that only "trustworthy" programs are running.

As this scenario suggests, we are concerned with *remote* attestation. Remote attestation involves authenticating a trusted component (e.g., a trusted platform module—TPM) on a remote machine prior to running an attestation protocol with it. Authentication matters: S would like to know that when it performs a remote attestation, its peer is *honest.* By communicating with a TPM, S avoids trusting the rest of a user's machine. But authentication poses a problem: A TPM's identity serves as *personally identifying information.* In our scenario, for example, if remote attestation reveals a TPM's identity, then online companies might misuse that information to track a user and profile her preferences.

DAA authenticates a TPM while offering anonymity guarantees. DAA performs the necessary authentication *without* revealing the TPM's identity, sending S a zero-knowledge proof instead.

In this project, we adapt DAA to work with an IBM secure coprocessor rather than a TPM. Such coprocessors serve as dedicated machines, designed to run arbitrary application-specific code in a controlled, secure environment. The TPM specification takes a far less interesting and (in the long-term) less maintainable approach to secure coprocessing: One size fits all. More important, the IBM coprocessors have been verified to FIPS 140-1 Level 4. Among other things, this means they offer laboratory-tested physical security against environmental attacks. Keys and other sensitive data stored in an IBM coprocessor's battery-backed RAM are automatically 'zeroized' when such attacks are detected [5]. To our knowledge, no TPM has been so verified.

Our contributions include

- An implementation of DAA for the IBM 4758 family of cryptographic coprocessors. To the best of our knowledge, no out-of-the-box implementations exist. This should enable trusted computing systems to harness the numerous hardware and software features provided by cryptocards. As we discuss in Section 4, we provide implementations in ML and C.

- Simplification. Due to the hardware limitations of the TPM, traditional DAA implementations offload expensive computations to the relatively more powerful processor of the host machine. However, the results of offloaded computations have to be verified by the TPM since the host is not trusted. In our model, we eliminate the distinction between the host and the cryptocard.

In Section 2, we describe DAA. In Section 3, we elaborate by describing the cryptographic building blocks used in DAA. In Section 4, we describe our implementation. A brief appendix discusses our individual contributions.

2 Protocol overview

In our setting, the DAA protocol involves three participants:

Issuer An issuer runs a setup procedure to establish parameters for itself and other participants. The issuer publishes these parameters along with a *zero-knowledge proof* that the parameters are well-formed. (We discuss such proofs in Section 3.3. Intuitively, they relate secret values to public values, without leaking the secrets.)

Other protocol participants check an issuer's parameters once, amortizing that cost over all subsequent uses of those parameters. In the following, we presuppose that other protocol participants have checked an issuer's parameters.

Cryptocard After mutual authentication, a cryptocard and an issuer run the *join protocol* to create a *certificate* for later use by the cryptocard. The certificate serves as evidence that the issuer has authenticated the cryptocard.

The join protocol employs blinding factors and two zero-knowledge proofs to (a) prevent the issuer from learning the secret key material, f, used by the cryptocard for DAA and (b) convince the issuer that the rogue tag sent by the cryptocard relates to f.

A cryptocard's *rogue tag* is a value calculated from an issuer's DAA parameters, the cryptocard's secret f, and the identity of its communication peer. Rogue tags serve as pseudonyms that permit a limited form of tracking sufficient for issuers and verifiers (q.v.) to heuristically detect misbehaving cryptocards.

Verifier A verifier and a cryptocard run the *signing protocol.* In it, the cryptocard uses a zero-knowledge proof to convince the verifier that (a) a certificate over f exists and thus the cryptocard is authentic and (b) the rogue tag sent by the cryptocard relates to f.

Brickell et al. describe a fourth protocol participant, the untrusted host housing a TPM. They do so in order to offload some computations from a resource-starved TPM to its host. As cryptocards are more powerful than TPMs, we avoid the distinction. In our implementation, a cryptocard performs all TPM and host calculations.[1]

Before describing the underlying cryptographic operations, we offer a bird's-eye view of the join and signing protocols.

Join protocol The join protocol presupposes that the issuer and the cryptocard have authenticated one another and established a secure channel. The issuer knows it's talking to a legitimate cryptocard and vice versa. Messages arriving at the card are authentic. The protocol proceeds as follows.

$$\begin{tikzcd}
\text{Cryptocard} \arrow[r, "{U,\ N_I}"] & \text{Issuer} \\
{} & {} \arrow[l, "n_i"'] \\
{} \arrow[r, "{P,\ n_h}"] & {} \\
{} & {} \arrow[l, "{P',\ C,\ v''}"']
\end{tikzcd}$$

[1]We do not avoid computations relevant to the Host-TPM divide.

The cryptocard initiates the protocol, sending a blinded signature request U and rogue tag N_I. The request U is built up from the issuer's parameters, the cryptocard's secret f, and a fresh blinding factor v'. Upon receiving the issuer's nonce n_i, the cryptocard sends a zero-knowledge proof P that U and N_I were well-formed along with a nonce n_h. The cryptocard's proof mentions n_i. Upon verifying P and confirming that N_I isn't blacklisted, the issuer sends an unblinding factor v'', a certificate C over f and the quantity $v := v' + v''$, and a zero-knowledge proof P' that C is well-formed. The issuer's proof mentions n_i and n_h.

Signing protocol The signing protocol is much simpler:

$$\begin{tikzcd}
{} \arrow[r, "{m,\, n_v,\, b}"] & \text{Cryptocard} \\
\text{Verifier} & {} \arrow[l, "{\sigma,\, N_V}"']
\end{tikzcd}$$

The cryptocard receives a message m to sign along with a nonce n_v and a bit b indicating whether m originated with the verifier or within the cryptocard. Hidden inputs include the relevant issuer's parameters and an optional basename for the verifier. The cryptocard responds with its rogue tag N_V for this verifier and a "signature" σ comprising a zero-knowledge proof that (a) the cryptocard knows a certificate C from the issuer and (b) both C and N_V relate to the cryptocard's secret f. The proof mentions m, n_v, and b.

3 Crypographic building blocks

We offer an informal account of the building blocks used by DAA. DAA comprises a group signature scheme (Section 3.1) paired with a rogue tagging scheme (Section 3.2), using *efficient* zero-knowledge proofs to achieve anonymity (Section 3.3).

DAA's need for both computational and communication efficiency should be clear: Devices with limited resources communicate remotely to participate in DAA. Thus, a constraint: One may design DAA-like protocols around any group signature and rogue tagging schemes for which one can define suitably efficient proof protocols.

3.1 Group signature scheme

The Camenisch and Lysyanskaya (CL) signature scheme supports group creation, the ability for a group's creator to add a member to the group, and the ability for a group member to prove membership to a third party [3]. The group of interest is the set of cryptocards that have proved their identity to a DAA issuer.

The CL signature scheme involves three operations built up over the group QR_n of quadratic residues for a special RSA modulus n:

- **Setup:** To create a new CL key, one picks a special RSA modulus n, one of its prime factors p, and members $a_1, \ldots, a_L$, b, and c of QR_n where L is a parameter. All but p are public.
- **Sign:** A signature for a block of messages $m_1, \ldots, m_L$ and a key as above comprises a random prime e, a random s, and a value v satisfying
$$v^e \equiv a_1^{m_1} \cdots a_L^{m_L} b^s \pmod{n}. \tag{1}$$
- **Verify:** To verify a signature (e, s, v) over a message block $m_1, \ldots, m_L$, one need only compute both sides of (1) and check for equality:
$$v^e \bmod n \stackrel{?}{=} a_1^{m_1} \cdots a_L^{m_L} b^s \bmod n.$$

A signer—knowing the secret p and thus how to factor n—can easily compute v satisfying (1) whereas any algorithm to forge such a signature reduces to an algorithm to solve the flexible RSA problem.

For important restrictions (*i.e.*, parameters governing the lengths of n, the m_i, e, and s) and a proof of unforgeability by reduction to the flexible RSA problem, please see [3].

3.2 Rogue tagging scheme

The rogue tagging scheme defines pseudonyms for cryptocards and offers a limited form of linkability. The goal is to permit an issuer or verifier to apply heuristics to detect "rogue" cryptocards and add such cryptocards to a blacklist.

To our knowledge, Brickell et al. introduced DAA's rogue tagging scheme when they defined DAA. That's rather unfortunate for an outsider attempting to understand the scheme in isolation: The rogue tagging scheme and its security properties are only implicitly defined as a part of the larger (and much more complicated) DAA protocol.

During setup, an issuer picks a group G such that the discrete logarithm problem in G has about the same difficulty as factoring CL RSA moduli. Many issuers and verifiers can share the same rogue-tagging group. DAA assigns distinct tags to issuers/verifiers via so-called basenames. As far as the protocol is concerned, basenames are uninterpreted bit strings; see §6.2 in [7] for a critique and practical advice.

While running the join protocol with an issuer or the signing protocol with a verifier that supplies its basename, a cryptocard computes a base $\zeta \in G$ from its peer's name and commits to the rogue tag $N := \zeta^f$, where f is the cryptocard's secret key for use in DAA signatures. (When signing with a verifier that does not supply a basename, the cryptocard picks a random $\zeta \in G$.)

DAA's rogue tagging key comprises primes Γ and ρ and a member γ of the multiplicative group mod Γ such that $\langle\gamma\rangle$ is a subgroup of order ρ and ρ is a large prime factor of $\Gamma - 1$. All of Γ, ρ, and γ are public.

There are at least two important points to make regarding the length of ρ:

- It's the maximum bit length of cryptocard secrets f.
- It must be chosen large enough to make it difficult to compute discrete logarithms in $\langle\gamma\rangle$; see the discussion around Algorithm 4.84, "Selecting a k-bit prime p and a generator α of $\mathbb{Z}_p^\star$" in [4].

We offer no "check if this tag is a rogue" heuristic. In theory, issuers and verifiers use tags to track legitimate users (pseudonymously) and to identify rogues. We offer tags, but no heuristics. Each issuer/verifier must settle on a policy. (Brickell et al. suggest that when talking to a platform with tag N, a verifier checks N and any earlier N' used by that platform against a blacklist. If found, the verifier should abort the protocol.)

3.3 Zero-knowledge proofs

DAA's zero-knowledge proofs ensure that all identifying information about a cryptocard rests in the rogue tags it supplies to its protocol peers.

The DAA protocol is interesting, in part, because it is a practical and widely-deployed protocol that relies heavily on zero-knowledge proofs of knowledge. We introduce such proofs using an extended example: The proof of knowledge of a discrete logarithms modulo a composite.[2]

3.3.1 The discrete log problem

Let the group G and g, $h \in G$ be given. Suppose a prover P and verifier V agree on these and P wants to convince V that

$$\begin{aligned} \exists\alpha.\, \alpha = \log_h g &\iff \exists\alpha.\, h^\alpha = g \\ &\iff g \in \langle h\rangle. \end{aligned}$$

without revealing the witness α.

[2]This is a special case of one of the zero-knowledge proofs used in DAA; see [2, Appendix A].

3.3.2 Making the verifier happy: Basic protocol

Consider the following protocol. It has the form "commit, challenge, response" (a so-called Σ-protocol).

$$\begin{array}{ccc}
P & \xrightarrow{\quad t := h^r \quad \text{for } r \geq 1 \text{ fresh} \quad} & V \\
P & \xleftarrow{\quad b \quad \text{for } b \in \{0,1\} \text{ fresh} \quad} & V \\
P & \xrightarrow{\quad s := r - b\alpha \quad} & V
\end{array}$$

After running the protocol, the verifier knows t, b, and s and so can check

$$t \stackrel{?}{=} g^b h^s.$$

From the verifier's perspective, the protocol is great. The point:

$$\begin{aligned}
t = g^b h^s &\iff h^r = g^b h^{r-b\alpha} \\
&\iff r = b \log_h g + r - b\alpha \\
&\iff b\alpha = b \log_h g \\
&\iff b = 0 \lor (b = 1 \land \alpha = \log_h g).
\end{aligned}$$

Thus a prover cannot answer both challenges $b = 0$ and $b = 1$ correctly without knowing $\log_h g$. Intuitively, the protocol satisfies *soundness*: If a (potentially adversarial) prover survives k runs of the protocol, then with probability 2^{-k}, the prover knows $\log_h g$.

3.3.3 Making the prover happy: An RSA group

Here's another intuition: A Σ-protocol satisfies *witness indistinguishability* if there exists a simulator for the prover such that (a) the simulator doesn't know the prover's secrets and yet (b) the distribution of "conversations" between the verifier and the prover matches the distribution of conversations between the verifier and the simulator.

If we could restrict our parameters G, g, and h in order to deny a (potentially adversarial) verifier any "computational back doors" to the prover's secret α, then we might attempt to prove our protocol satisfies witness indistinguishability.

We restrict our parameters as follows. Let n be a special RSA modulus; that is, there exist primes p, p', q, and q' satisfying $n = pq$, $p = 2p' + 1$, $q = 2q' + 1$, and $q \neq p$. Let G be the multiplicative group mod n and h a random generator of the group of quadratic residues mod n. Thus, $|\mathrm{QR}_n| = p'q'$. Our protocol specializes to

$$\begin{array}{ccc}
P & \xrightarrow{\quad t := h^r \bmod n \quad \text{for } r \in \{1, \ldots, p'q'\} \text{ fresh} \quad} & V \\
P & \xleftarrow{\quad b \quad \text{for } b \in \{0,1\} \text{ fresh} \quad} & V \\
P & \xrightarrow{\quad s := (r - b\alpha) \bmod p'q' \quad} & V
\end{array}$$

where the verifier computes

$$v := g^b h^s \bmod n$$

and checks

$$t \stackrel{?}{=} v.$$

Now our imaginary proofs can argue that no such "computational back doors" exists by reduction to the strong RSA assumption.

3.3.4 Making the protocol practical: Avoiding communication

We now introduce the so-called Fiat-Shamir heuristic, used to avoid communication between the prover and the verifier. The idea is to use a hash function as a source of challenge bits.

Let a hash function H with output length ℓ_H be given. The prover picks ℓ_H random values; computes the corresponding commitments $t_1, \ldots, t_{\ell_H}$; hits the common inputs and the commitments with H; then computes ℓ_H responses using bits from the hash as challenges. In practical terms, the prover hopes that no adversary can defeat H to discover computational back doors to α. Now our imaginary proofs must make the random oracle assumption, in addition to the strong RSA assumption.

3.3.5 On efficiency

The zero-knowledge protocol discussed in the preceding sections has one drawback: Relying as it does on a list of ℓ_H commitments, it's not very efficient.

The CL signature scheme and DAA use this inefficient protocol, but only to establish that long-lived parameters (*e.g.*, the CL group parameters) are well-formed. Each protocol participant checks such parameters once, amortizing that cost.

CL and DAA use several other zero-knowledge protocols to prove knowledge of relations among discrete logarithms (amongst other relations). These zero-knowledge protocols are much more efficient: Presupposing that parameters have been checked, they "get away with" one commitment. An account of these protocols is beyond the scope of this report; please refer to [3, §5] and [1, §3.2] for references.

4 Implementation

We briefly describe our implementation of the DAA protocol on the IBM 4758 PCI-X Cryptographic Coprocessor. We developed (1) a full implementation of DAA—issuer, verifier, and cryptocard algorithms—in the functional programming language OCaml and (2) a separate implementation of the cryptocard-specific algorithms in C, a language we can readily compile for execution on cryptocards.

We decided to divide our efforts because the potential benefits seemed to outweigh the obvious cost. First, our ML implementation is easier to scrutinize, being relatively high-level and thus "close" to the protocol specification. It served as the protocol reference when we started writing C code. As important, several categories of bugs (*e.g.*, memory safety errors) are easily avoided in ML; we sought to avoid, as much as possible, routine debugging chores related to verifier/issuer code. Second, our goal was to perform an attestation. We felt that using separate code bases would likely rule out some "false positives" caused by bugs common to both.

Status: Our ML implementation is complete, but bugs remain. We know of at least one, wherein the code to verify an issuer's parameters reject parameters that should be well-formed.

Our C implementation is incomplete, lacking some portions of the DAA signing protocol. It has not been debugged, either in isolation or in combination with our ML implementation. We have not performed an attestation involving both code bases.

Our C implementation relies on the GMP library to perform big integer operations that are not supported by the cryptocard; for example, linear arithmetic. However, some functionalities that are implemented now as utility functions need to be reimplemented to use similar functionalities offered by the cryptocard.

References

[1] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, 2004.

[2] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. Cryptology ePrint Archive, Report 2004/205, 2004. Full version of [1].

[3] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *Proceedings of the 3rd international conference on Security in Communication Networks*, 2002.

[4] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[5] Sean Smith, Ron Perez, Steve Weingart, and Vernon Austel. Validating a high-performance, programmable secure coprocessor. In *National Information Systems Security Conference*, 1999.

[6] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999.

[7] Ben Smyth, Mark D. Ryan, and Liqun Chen. Formal analysis of privacy in direct anonymous attestation schemes. Cryptology ePrint Archive, Report 2012/650, 2012.

A Contributions

We describe the work we have done, jointly and individually.

Elnikety: Eslam helped define the problem. He read related work in trusted computing, trying to pin down which capabilities a system can provide when integrated with a secure cryptocard. He helped write our project proposal. Later, during the implementation, he worked on the cryptocard DAA code. He helped write this report.

Kapcari: Edit helped define the problem. She investigated the available capabilities for TPM on Linux machines. She presented our proposal to the class. During the implementation, she worked on the network communication between the issuer, the cryptocard, and the verifier.

Rajani: Vineet helped define the problem. During the implementation, he investigated access to the cryptocard and how to compile, deploy, and communicate with custom-built binary images on the cryptocard. He worked on the cryptocard DAA code.

Swasey: David helped define the problem. He read related work on trusted computing, IBM cryptocards, and zero-knowledge proofs. He helped write our project proposal. He closely read the DAA paper and related literature and explained them to the group. He implemented the full DAA protocol in ML. He helped write this report.