How to Validate a Verification?

Houda Ferradi

Freelance Consultant
Ferradih@gmail.com

Abstract. This paper introduces *signature validation*, a primitive allowing any third party T (Théodore) to verify that a verifier V (Vadim) computationally verified a signature s on a message m issued by a signer S (Sarah).
A naive solution consists in sending by Sarah $x = \{m, \sigma_s\}$ where σ_s is Sarah's signature on m and have Vadim confirm reception by a signature σ_v on x.
Unfortunately, this only attests *proper reception* by Vadim, i.e. that Vadim *could have checked* x and not that Vadim *actually verified* x. By "actually verifying" we mean providing a proof or a convincing argument that a program running on Vadim's machine checked the correctness of x. This paper proposes several solutions for doing so, thereby providing a useful building-block in numerous commercial and legal interactions for proving informed consent.

1 Introduction

In many practical scenarios, it is very useful for all parties to dispose of primitive allowing any recipient to ascertain that a given party actually verify the digital signatures that they get. By "actually verified" we mean that that receiving party checked *computationally* the signatures on its machine. A classical example is that of a user sub-contracting verification to a distrusted third party or the willingness to ascertain full informed consent of the received information.

This paper introduces *signature validation*, a primitive allowing any third party T (Théodore) to verify that a verifier V (Vadim) *computationally* verified a signature s on a message m issued by a signer S (Sarah).

In our scenario Sarah, sends a signed document m to Vadim. Théodore (T) wishes to check that the Sarah's signature σ_S was computationally verified by Vadim.

A classical solution consists in sending to Théodore $x = (m, \sigma_S)$ as well as Vadim's signature on x (hereafter σ_V). This however only attests the *proper reception* of x by Vadim and not actual verification. In other words, σ_V attests that Vadim *could have verified* x and not that Vadim *actually verified* it. Here "actually checking" means "providing a mathematical proof or a convincing argument that a program running on Vadim's computer checked the correctness of x".

Is it possible at all to provide such a proof?

As we will see in the subsequent sections, prior art provides several solutions for doing so but all are involved and frequently interactive constructions that depart in terms of complexity and efficiency from the constructions provided in this paper.

To avoid useless and tedious re-description of some building-blocks and prior art, we reproduce in this article (sections 3 and 2.1) two sections of the second author's paper [CGSH+21].

2 Notations & Building-Blocks

We will use the following standard notations:

Table 1. Conventions used in this paper.

notation	**meaning**
$=$	mathematical identity
$:=$	definitional equality
$\leftarrow$	assignment
$x \overset{\$}{\leftarrow} X$	sample an element x uniformly at random from a set X
$\mathsf{HASH}(x)$	any (unspecified) hash function
$\sigma_P \leftarrow \mathsf{Sig}_P(m)$	a classical signature of message m by party P
$b \leftarrow \mathsf{Ver}_P(\sigma_P, m)$	the verification process corresponding to $\mathsf{Sig}_P(m)$
$\sigma_P \leftarrow \mathsf{MRSig}_P(m)$	a message recovery signature of message m by party P
$\{m, b\} \leftarrow \mathsf{MRVer}_P(\sigma_P)$	the verification process corresponding to $\mathsf{MRSig}_P(m)$

2.1 Machine Model and State

As part of some of our protocols we assume that the Théodore and the Sarah agree on a concrete computational model, that can be simulated using finitely many resources by a deterministic Turing machine. We only need a way to describe this model unambiguously (so that its parameters can be agreed upon) and that, when running a program P on input x, we can obtain the sequence of states τ that the machine $\mathcal{M}(P, x)$ goes through during computation.

Any such model could be used, but for the sake of compactness and applicability, we may want to use higher-level semantics.

One well-studied model that can be used is TinyRAM, introduced by Ben-Sasson et al. [BCG+13] for the very purpose of proving program execution. TinyRAM is close enough to real programs that it can be translated and compiled on most computer architectures, yet it enjoys a full specification together with a small instruction set, making it easier to prove statements about. In particular, for our needs, the TinyRAM assembly is very succinct, having only 29 opcodes, but most importantly its state is straightforward to capture.

We recall here some elementary facts about TinyRAM, for the sake of completeness[1]. A TinyRAM machine is described by two integers (W, K) together with a state $(P, \mathsf{pc}, \{r_1, \dots, r_K\}, \mathsf{f}, \mathsf{mem}, x)$ where:

Notation	Definition
P	the program to be executed (considered as a read-only sequence of elementary operations)
pc	is a W-bit integer (indicating which instruction is currently being executed)
$r_1, \dots, r_K$	are W-bit registers
f	a one-bit flag
mem	an array of 2^W bytes
x	a string of W-bit integers, representing the input

At every clock cycle, TinyRAM fetches the instruction in P indicated by pc, and reads if necessary from the input tape x. A special instruction answer takes a single argument and acts as the return value of program P it immediately terminates execution. Before the execution of P all registers, all memory cells, the flag and the program counter pc are set to zero. Any other computational model could be used, but TinyRAM strikes a nice balance between usability and compactness.

In addition to the above, we enrich $\mathcal{M}$ by an additional feature. At every clock cycle t the machine concatenates its state, i.e. the data:

$$\text{state}_t = (P, \mathsf{pc}_t, \{r_1, \dots, r_K\}_t, \mathsf{f}_t, \mathsf{mem}_t, x)$$

to a global string:

$$\text{state} = \{\text{state}_0, \dots, \text{state}_{\tau-1}\}$$

When $\mathcal{M}$ halts it outputs $h = \mathsf{HASH}(\text{state})$.

Remark 1. The above assumes that P, x remain invariant during execution, if such is not the case replace in the above definition P, x by P_t, x_t.

2.2 Pollard's Kangaroo Algorithm

One of our solutions will use Pollard's kangaroo algorithm (also called the λ algorithm) [Pol78,Pol00] as a black-box. Given $\Delta = r^a \bmod p$ and r, the algorithm will find a in a complexity of $O(\sqrt{a})$. The advantage of this algorithm is its generic nature, i.e., it will work in any finite cyclic group.

[1] The current specifications can be found here: http://www.scipr-lab.org/doc/TinyRAM-spec-2.000.pdf.

3 Related Work

The topic of verifiable computing was kickstarted by Babai et al. [BFLS91] in the context of monitoring large computations performed by a powerful, but fallible, supercomputer. This contrasts with the more traditional approach of majority or quorum computation, where a single task is repeated several times with the hope that not all computers conspire to lure the verifier [CRR11, CL02]. The new paradigm relies on providing a *proof of validity* together with a computational result: the celebrated PCP theorem [ALM+98, AS98, AS92, Hås01] states that with a suitably encoded proof, it is sufficient for the verifier to check three randomly chosen bits! Unfortunately, this theorem does not provide a practical, usable protocol that can be implemented. Furthermore, the PCP proof might be very long (potentially too long for the verifier to process).

An interactive protocol for verifiable computation was proposed by Ishai et al. [IKO07] and the first *non-interactive* primitives were very limited [Mic94]. A history of these developments can be found in Goldwasser et al. [GKR15]; several implementations are also available [SMBW12, VSBW13, PHGR16].

Most of the protocols above start by translating a program into a *circuit*, then translating this circuit into a polynomial (arithmetization). The verifier supplies the input and the prover executes the circuit, producing a transcript from intermediate values. Rather than sending the transcript to the verifier (who could then run the circuit themselves, and thus check the transcript's validity) the key idea is to *convince the verifier that a valid transcript exists* by encoding the transcript in some way, then having the verifier *probe* some parts of that encoded transcript. This makes it possible for a computationally weaker verifier to nevertheless check the work of a computationally stronger prover.

In the non-interactive setting literature this is achieved by either extracting a commitment [IKO07, Blu11, SBV+13, SMBW12, SVP+12, VSBW13] or by using encrypted queries [GGPR13, BCI+13, BCG+13, BCTV14], in both cases using PCP under the hood. The above thread of research shows that it is possible for the verifier to check the prover's claim — for a given program and a given input — without running the full program itself.

We may also consider that prior art consists of building blocks that, whilst not resorting to verifiable computation, has as a *side effect* similar verifiability properties. A natural primitive that comes to mind while doing so is signcryption [Zhe97]. A careful look into [Zhe97] reveals that while this primitive is close to what we wish to achieve, the step performing the verification of the signature, namely the comparison $\mathrm{KH}_{k_2}(m) = r$ comes only *after* the decryption step. It is hence possible to decrypt m without actually verifying the signature on it.

Our contribution. Our approach achieves a similar goal, albeit much more efficiently, in the sole context of digital signatures, and by different means. One advantage of our protocol over traditional verifiable computing approaches is that it is straightforward to implement, that it is representation-agnostic (in the sense that any machine model can be used, not only circuits), and that it does not rest on new mathematical assumptions.

4 Generic Solutions

We start by presenting three generic ways allowing to achieve the desired function. Here the term generic refers to protocols that can be instantiated using any underlying digital signature scheme.

4.1 Virtualization

We start by a first construction achieving the stated goals.

Setup. We assume that both parties agree on a program P performing the verification Ver_S and a serialization of it written $[P]$. At start $[P]$ includes the public-keys of S in its code (hard-coded) or is given the public-keys within the input parameter.

– Sarah signs the message m and sends to Vadim:

$$x := \{m, \sigma_S\} \leftarrow \{m, \mathsf{Sig}_S(m)\}$$

– Vadim runs $h \leftarrow \mathcal{M}(P, x)$.
– Vadim signs:

$$\sigma_V = \mathsf{Sig}_V(\{[P], x, h\})$$

and sends σ_V to Théodore.
– Théodore computes h independently using his own instance of $\mathcal{M}$ and then checks that:

$$\mathsf{Ver}_V(\sigma_V, \{[P], x, h\}) = \mathsf{True}$$

Remark 2. In many cases, e.g. DSA, the protocol can be considerably simplified by performing on $\mathcal{M}$ only a final crucial verification step. For instance, if we are given a standard DSA signature r, s it appears unnecessary to implement the entire Ver_S on $\mathcal{M}$. Vadim and Théodore may pick a moderate-size (e.g. 80 bit) prime γ and compute by any desirable means (not necessarily using $\mathcal{M}$) the quantity:

$$\rho = g^{m/s} y^{r/s} \bmod p \bmod q\gamma$$

and then run on $\mathcal{M}$ only the final operation $\rho \bmod q$ and the comparison of the obtained result to r.

In the case of RSA, this applies as well by computing by any means the quantity $\rho = \sigma_S^e \bmod \gamma n$ and using $\mathcal{M}$ only for the operation $\rho \bmod n$ and the comparison of final result to m.

Note that in both cases $\mathcal{M}$ can use a reduced instruction-set, i.e. only the few opcodes required to run the above operations.

Interestingly, those techniques are remind Shamir's countermeasure against fault attacks [Sha99].

Remark 3. It is important to note that the devil is in $\mathcal{M}$'s implementation's details. Assume that $\mathcal{M}$ is an 8-bit architecture, then, in theory, Vadim may run the operation $\mathcal{M}(P, x)$ for all cycles except one (say j), guess the correct value of the result register at step j and continue the calculation. Hence, such a strategy will comply with attack requirement that σ_S was not completely verified by Vadim using the program P and have a success probability larger than 2^{-H} where H is the digest size of HASH. We consider this as a theoretical risk that can be mitigated in a several ways, for instance, the same signature can be checked μ times using different programs $P_0, \ldots, P_{\mu-1}$ whose results are concatenated to form the final output. This forces the attacker to successfully guess all the missing js in all executions. Another mitigation (that does not solve the problem but makes cheating harder) consists in using an $\mathcal{M}$ whose word size is large and ascertain that all opcodes return a result having a high enough entropy.

This solution requires the implementation of virtual machine $\mathcal{M}$ on both ends and performing trace collections that might be more or less tedious depending on the complexity of Ver_S.

4.2 Perturbing

A second solution consists in forcing Vadim to solve an easy challenge derived from the signature. Here we assume that underlying signature scheme has a (plausible) conjectured property which is that the signature cannot be derived from the challenge in a way not that avoids verifying the signature.

Let $\ell \in \mathbb{N}$ be a security parameter (typically $\ell = 20$).

The protocol is the following:

- Sarah signs the message m and sends to Vadim:
$$x := \{m, \sigma_S\} \leftarrow \{m, \mathsf{Sig}_S(m)\}$$
- Vadim generates a random ℓ-bit integer u and sends to Théodore:
$$(m, \sigma_S) = (m, \mathsf{Sig}_S(m) + u)$$
- Vadim determines (using exhaustive search) the value i such that:
$$\mathsf{Ver}_S(\sigma_S - i, m) = \mathsf{True}$$
- Vadim signs:
$$\sigma_V = \mathsf{Sig}_V(m, \sigma_S)$$
and sends σ_V to Théodore who checks that σ_S was properly determined. Note that Théodore must also check that σ_S verifies correctly with respect to Sarah's public key (otherwise Sarah and Vadim may collude to mislead Théodore).

The crux of this solution lays in the fact that Vadim identified the correct signature by removing the noise u and that this identification is assumed to be doable using Ver_S. A cheating Vadim may avoid verifying the signature and bet on a given i value but his odds in succeeding to do so are $2^{-\ell}$.

Remark 4. Note that if the protocol is repeated t times using t different signatures on the same message using smaller ℓ values, security is still $2^{-t\ell}$. This speeds-up verification but costs of additional transmission.

Remark 5. Note that if the signature σ_S is publicly available (e.g. the certificate of a known administration) then this protocol is insecure because Vadim can just retrieve σ_S and forge a correct answer without doing any exhaustive search work. Hence, this protocol is usable only in the settings where σ_S is unknown to Vadim. If such is not the case, the protocol can be slightly modified using randomized signatures (e.g. [PS16]) to have Théodore challenge Vadim with an equivalent yet different signature σ'_S of the same message m. This requires from Théodore some work (randomizing is in essence equivalent to the verification of a signature) but can be sub-contracted to some trusted randomizing entity. Note that if the messages need to be hidden, validation applies to SoRCs as well [BF20].

Remark 6. In many settings a complete re-verification is not necessary. e.g. if Sarah signs m using RSA, she can provide as noisy signature the quantity:

$$\left(\frac{m}{2^u}\right)^d \bmod n$$

We see that this allows Vadim to avoid repeated exponentiations given that to spot the correct u all he needs to do is exponentiate once, divide by m and perform successive modular multiplications by 2 until he gets 1 (or spots a valid redundancy if m is padding using a probabilistic padding scheme), which is a very simple operation.

Remark 7. In some *ad hoc* cases, u can silently exploit internal redundancy already existing in the protocol. For instance if PSS is used, the signature σ_S can be replaced by the quantity:

$$\sigma_S^{d^u \bmod \phi(n)} \bmod n$$

Vadim can then detect u by repeated elevations to the power e until a correct signature is found.

Remark 8. If we relax genericity, *ad hoc* protocols can be more efficient as they reduce the search space quadratically. We give one example here. Consider a DSA-like signature scheme where:

$$r = g^k \bmod p \quad \text{and} \quad s = \frac{m + xr}{ka} \bmod q$$

We note two differences with standard DSA: the first is that r is not reduced modulo q and the second is the appearance of a new parameter a in the equation describing s.

To verify such a signature Vadim must check that:

$$\Delta = g^{m/s} y^{r/s} = r^a \bmod p$$

We hence see that verifying the signature requires solving for a the discrete logarithm problem:

$$\Delta = r^a \bmod p$$

However, as we saw previously the Kangaroo algorithm allows to solve this equation in $O(\sqrt{a})$. In other words, if a is a ℓ-bit number, Vadim only needs to perform $O(2^{\ell/2})$ operations to implement the protocol for a security level of 2^ℓ.

The second solution is conceptually simpler and does not require any virtualization of Ver_S. It does, however, require on the Vadim's side, 2^ℓ native code re-runs of Ver_S where ℓ is a security parameter (typically $\ell = 20$).

4.3 Message Recovery Signatures

The third generic solution is based on message recovery signatures. Interestingly, it differs from the previous (and the next) solutions in the fact that Vadim learns m only *ipso facto*, by performing the verification. In other words, this protocol is very close to the delivery of a physical registered letter where the receiver is required to sign the postal receipt *before* receiving the letter and actually knowing what information the letter contains. This quasi-perfect simulation of the physical world makes the proposed protocol promising in digital postage applications.

– Sarah signs m using a message recovery signature scheme:

$$\sigma_S = \mathsf{MRSig}_S(m)$$

– Sarah sends σ_S to Vadim.
– Vadim verifies σ_S and hence retrieves m.
– Vadim signs:

$$\sigma_V = \mathsf{Sig}_V(m, \sigma_S)$$

and sends σ_V to Théodore.
– Théodore has all the information required to check that:

$$\mathsf{Ver}_V(\sigma_V, (m, \sigma_S)) = \mathsf{True} \text{ and } \{\mathsf{True}, m\} = \mathsf{MRVer}_S(\sigma_S)$$

This solution requires no virtualization or exhaustive search. It assumes however that the message recovery signature scheme has the property that it is impossible to extract m otherwise than by verifying the signature. We regard this conjecture as plausible but it should be carefully checked for each instance of message recovery signature algorithm used in this generic construction.

5 Conclusion

The techniques described in this paper can find applications in a variety of practical situations where a sender needs the proof that a signature was not only received but also *effectively verified* by a remote machine. All presented solutions are simple to model and implement.

The observation of and the techniques in this paper can be extended in various directions. Here are a three:

ℓ-order validation: Because Vadim signs his validation, nothing prevents from validating the validation or validating the validation of the validation etc. Such as recursion gives a more refined, yet somewhat philosophical, notion of trust and informed consent in electronic exchanges.

Batch validation: We note that batch verification and screening (e.g. [KP17], [BGR98]) also lends themselves to validation. In this case Théodore can aggregate the signatures to be checked and get from Vadim a proof of validation for the entire batch.

Zero-resource zero-knowledge/signature verification: Consider a scenario where Vadim is nearly resourceless. His goal is to identify Sarah using a ZKP. To that end, Vadim resorts to the help of Xavier, in whom Vadim has limited trust. By "limited trust" we encompass the following assumptions:

- Sarah and Xavier do not collude to mislead Vadim.
- We tolerate a scenario where Xavier does not answer (DoS on Vadim).
- We tolerate a scenario where Xavier induces Vadim into falsely rejecting a honest Sarah.
- We do not tolerate a scenario where Xavier induces Vadim into falsely accepting a dishonest Sarah.
- We wish the protocol to be resilient to an attacker mispresenting himself as Xavier to Vadim.

Vadim conducts the full ZKP and gets a trace $\{x, c, y\}$[2]. Vadim sends to Xavier $\{x, y\}$ and asks Xavier to exhaustive search c. If Xavier replies with the correct c, Vadim accepts Sarah. Xavier may perform a DoS attack and reply with a fake c (or not answer at all), but Xavier cannot mislead Vadim into accepting a fake Sarah unless Sarah and Xavier collude[3].

[2] Here x denotes the commitment, c the challenge and y the response in the ZKP.
[3] or if Xavier poses as being Sarah

[4]. To minimize this DoS and/or collusion risk Vadim may resort to several independent Xaviers and implement a majority choice thereby reducing the risk at wish. The same works with digital signatures and provides Vadim with both verification and validation for free (here it suffices to perturb σ_S as explained before). Hence, this idea can prove very useful in scenarios where Vadim has nearly no computational resources but where communication is cheap.

References

ALM$^+$98. Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998. (cited on page 4)

AS92. Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs; A new characterization of NP. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 2–13. IEEE Computer Society, 1992. (cited on page 4)

AS98. Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998. (cited on page 4)

BCG$^+$13. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013. (cited on pages 2, 4)

BCI$^+$13. Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, 2013. (cited on page 4)

BCTV14. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 781–796. USENIX Association, 2014. (cited on page 4)

BF20. Balthazar Bauer and Georg Fuchsbauer. Efficient signatures on randomizable ciphertexts. Cryptology ePrint Archive, Paper 2020/524, 2020. https://eprint.iacr.org/2020/524. (cited on page 7)

BFLS91. László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 21–31. ACM, 1991. (cited on page 4)

[4] In other words we assume a secure channel between Vadim and Théodore allowing to Vadim to ascertain that his communications are indeed readable only by Xavier. Here Rabin encryption with Xavier's public-key does the job with one squaring.

BGR98. Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 236–250, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. (cited on page 9)

Blu11. Andrew J. Blumberg. Toward practical and unconditional verification of remote computations. In Matt Welsh, editor, *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011*. USENIX Association, 2011. (cited on page 4)

CGSH$^+$21. Clémence Chevignard, Rémi Géraud-Stewart, Antoine Houssais, David Naccache, and Edmond de Roffignac. How to claim a computational feat. Cryptology ePrint Archive, Paper 2021/1554, 2021. https://eprint.iacr.org/2021/1554. (cited on page 2)

CL02. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. (cited on page 4)

CRR11. Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 445–454. ACM, 2011. (cited on page 4)

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013. (cited on page 4)

GKR15. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015. (cited on page 4)

Hås01. Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001. (cited on page 4)

IKO07. Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short pcps. In *22nd Annual IEEE Conference on Computational Complexity (CCC 2007), 13-16 June 2007, San Diego, California, USA*, pages 278–291. IEEE Computer Society, 2007. (cited on page 4)

KP17. Apurva S. Kittur and Alwyn Roshan Pais. Batch verification of digital signatures: Approaches and challenges. *Journal of Information Security and Applications*, 37:15–27, 2017. (cited on page 9)

Mic94. Silvio Micali. CS proofs (extended abstracts). In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 436–453. IEEE Computer Society, 1994. (cited on page 4)

PHGR16. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, 2016. (cited on page 4)

Pol78. John Pollard. Monte Carlo methods for index computation ($\bmod p$). *Mathematics of Computation*, 32, 1978. (cited on page 3)

Pol00. John Pollard. Kangaroos, Monopoly and Discrete Logarithms. *Journal of Cryptology*, 12:437–447, 2000. (cited on page 3)

PS16. David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016*, pages 111–126, Cham, 2016. Springer International Publishing. (cited on page 7)

SBV$^+$13. Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 71–84. ACM, 2013. (cited on page 4)

Sha99. Adi Shamir. Method and apparatus for protecting public key schemes from timing and fault attacks. US Patent 5,991,415, 1999. (cited on page 5)

SMBW12. Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. (cited on page 4)

SVP$^+$12. Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 253–268. USENIX Association, 2012. (cited on page 4)

VSBW13. Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013. (cited on page 4)

Zhe97. Yuliang Zheng. Digital signcryption or how to achieve cost(signature & encryption) $\ll$ cost(signature) + cost(encryption). In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 165–179, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. (cited on page 4)