Toward practical and unconditional verification of remote computations

Srinath Setty*, Andrew J. Blumberg*, and Michael Walfish*   *UT Austin

1 Introduction

This paper revisits a classic question: how can a machine specify a computation to another one and then, without executing the computation, check that the other machine carried it out correctly? The applications of such a primitive include cloud computing (a computationally limited device offloads processing to the cloud [11] but does not assume the cloud's correctness [12]); volunteer computing (some 30 projects use the BOINC [1] software platform to leverage volunteers' spare cycles, but some "volunteers" return wrong answers [2]); and high-assurance computing (a machine may be remotely deployed and subject to physical tampering).

Today, a common way to verify computations is replication [9, 17, 23, 27]. However, replication may not be viable (say if the performing computer is an embedded robot). It also requires assumptions about failure independence. Another technique is auditing [17, 24], but if the performer understands the computation better than the requester, the performer can alter strategic bits, undetected by an audit. A final technique is trusted computing [10, 21, 26, 28, 29], but it assumes that some component—the hardware, the hypervisor, a higher layer—is not physically altered. Can we instead make *no correctness assumptions* about the performer?

In theory, such *unconditional* verification has been achievable since the 1980s, using both interactive proofs [15] and probabilistically checkable proofs (PCPs) [3, 5]. Informally, the central theorem in the area states that a client can—with a suitably encoded proof and under a negligible chance of viewing a wrong answer as correct—check an answer's correctness in constant time [3]. Unfortunately, this astonishing body of theory is considered by folklore to be impractical. This skepticism is well-founded: it is based on the complexity of the algorithms and long experience trying to use general-purpose cryptographic results in practical systems.

This brings us to the purpose of this paper, which is to propose a new line of systems research: using the machinery of PCPs, can we build a system that (a) has practical performance, (b) is simple to implement, and (c) provides unconditional guarantees? Note that (a) and (b) contrast with PCPs as used in the theory literature and (c) contrasts with current systems approaches. To illustrate the promise of this line of research, we do the following:

(1) Identify work in the PCP literature that provides a base for systems research (§3.1–§3.2). We first looked to the PCP literature, then observed that efficient argument systems [19, 22] (PCP variants in which the server proves that it has a proof by answering questions interactively) are promising, and then noticed that a particular argument system [18] could lead to a practical solution.

(2) Refine the approach of [18] into a design that is practical over a limited domain (§3.3). We applied refinements to shrink program encoding (via arithmetic circuits instead of Boolean circuits), enable batched proofs (which enhances performance for computations that can be decomposed into parallel pieces), and improve amortization (by moving more of the work to a setup phase). These innovations are essential to practical performance.

(3) Implement this design to demonstrate its practicality (§4). To our knowledge, PCP theory has never before found its way into any efficient implementation. Thus, we believe that our implementation, though limited, is a contribution. Our implementation is also comparatively simple; it could conceivably be formally verified.

(4) Articulate a research agenda for extending the reach of our approach (§5). Our ultimate goal is a practical system for general-purpose verified computation.

The four contributions above provide a concrete foundation for our position, which is that *PCP-based verifiable computation can be a systems problem, not just a theory problem.* We need this foundation because PCPs are thought to be impractical; indeed, our prior designs were too expensive by over 11 orders of magnitude. Even our prototype achieves goals (a)–(c) above only over a limited domain.

Our initial demonstration is $m \times m$ matrix multiplication over (large) finite fields. We chose this as our core example for two reasons. First, matrix multiplication was our initial test: it's parallelizable and efficiently encodable as a circuit, so we knew that if we couldn't make it perform well, we were out of business. Beyond that, matrix multiplication has practical significance. It is a core primitive in many applications: image processing (e.g., filtering, rotation, scaling), signal processing (e.g., Kalman filtering), data mining (e.g., recommendation engines based on collaborative filtering [6]), etc.

Fortunately, our results for this computation are encouraging. In our implementation, the client's measured work per computation is Km^2 operations, with K on the order of several hundred (the exact value depends on the desired confidence), and the server's is roughly $12m^4$ operations Thus, relative to the naive m^3 algorithm, the prototype's costs at the client are cheaper than computing locally for $m > K$ (which holds for even small image files). The situation with the server is grimmer. Its costs are undoubtedly high: a factor of $12m$ greater than (naively) executing the computation with no verification.

Nevertheless, we are willing to label this practical, for two reasons. First, one should expect a cost for provable assurance, and a linear factor (with good constants) is perhaps tolerable. Second, our view may be colored by our prior designs: the worst of them called for m^9 server work with constants greater than 10^{16} (seriously). We have made a lot of progress since then, and we no longer worry about being laughed out of the room.

Apart from the server's cost, our scheme has other significant restrictions. The most serious is the program encoding: while arithmetic circuits achieve efficiency comparable to a C program for many linear algebra computations, they are much less efficient for computations that have many conditionals and control flow. Moreover, our performance relies on the circuit being efficiently decomposable into independent parallel-executable chunks. Another issue is that our scheme requires an expensive one-time setup phase, the cost of which is amortized over all instances of the same computation; while reasonable in some scenarios, this is obviously not ideal for generic outsourced computation. Finally, for smaller computations (e.g., $m \times m$ matrix multiplication for $m < K$), even the online phase is more expensive for the verifier than executing the computation locally, which undermines the usual motivation for verified computations: saving the client work [10, 12, 14, 26].

However, even for small computations, our scheme could be useful in certain operating regimes. If computing remotely is unavoidable (because of the deployment scenario), if the server is expensive, and if the context is mission-critical, then assurance is important. In these circumstances, the client's verification work is permitted to cost more than computing locally (within reason), and an expensive pre-deployment step is acceptable.

2 Related work

As mentioned above, replication and trusted hardware rely on assumptions about the remote computer, in contrast to our goal of unconditional verification. The introduction also mentioned auditing, and here we highlight the contrast between spot-checking and PCP-based guarantees. Unless the audit coverage is nearly perfect (which degenerates to executing the computation), a performing computer that alters a key bit (e.g., in an intermediate step) is unlikely to be detected. Under PCPs, however, *any* deviation from a correct answer (even in a single bit!) is detectable with high probability with only a small number of queries to the proof. This unintuitive aspect of PCPs is described further in Section 3.1.

The potential of PCPs and interactive proofs to serve as a foundation for verified computations has long been known to theorists [3, 5, 8, 15]. We borrow from this theory, but our goal is a practical system, which has biased our choice of techniques. For instance, we built upon [18], which admits a remarkably easy implementation, rather than an intricate scheme by Goldwasser et al. [14] even though it is asymptotically more efficient.

For verified computations, theorists have also turned to secure multi-party protocols, where parties compute an agreed-upon function of private data, revealing only the result [32]. These protocols alone do not provide verifiable computations, but Gennaro et al. [12] combine Yao's construction with Gentry's homomorphic encryption [13] to provide verifiable non-interactive computing. However, Gentry's scheme is not yet practical on today's computers, and secure multi-party protocols themselves are prohibitive [20]. An efficient version of these techniques specialized to arithmetic circuits [16] has different asymptotics (significantly more work for the verifier, better overall burden) than our approach, and it is unclear if it is amenable to the amortizations we rely on.

Some approaches to verified computation work over specific problem domains [4, 31]. For that matter, Freivalds' technique [25] verifies a matrix multiplication in $O(m^2)$ time with a randomized algorithm. However, our ultimate goal is to support arbitrary functions, so we head in a different direction from special-purpose protocols.

3 Approach

We first sketch a totally impractical scheme and then sketch a series of refinements that reduce it to practice.

3.1 Verified computations and PCPs

We wish to implement the following protocol: A computer we control, the *verifier*, sends a program Ψ and an input x to a remote computer, the *prover*. The prover returns a result y and a proof P, where P establishes that y is the result of running Ψ on x. Moreover, it must be cheaper to check P than to compute y locally. Such asymmetric checking is precisely what PCPs enable.

Since many PCP constructions are geared to proving the satisfiability of Boolean circuits, we now cast our problem in this language.[1] There is a Boolean circuit B (which depends on Ψ, x, and y) such that B is satisfiable (that is, evaluates to 1) iff y is the correct output of Ψ run on x. Assume for now that the prover and verifier can each efficiently derive B, given Ψ, x, and y. Then, if the prover could prove that B is satisfiable, the verifier would know that y is the correct output of Ψ run on x. Of course, the assignment $\vec{a}$ (that is, the values of all of the wires in B) constitutes an (obvious) proof that B is satisfiable: the verifier could check $\vec{a}$ against every gate in B.

The remarkable content of the PCP theorem, however, is that there is a proof, P, that the verifier can check by examining only a *constant number* of bits of P. Specifically, for any satisfiable Boolean circuit B, there is a

[1]A Boolean circuit is a set of interconnected gates, each with input wires and an output wire, with wires taking 0/1 values.

proof that convinces a constant-time verifier; if B is not satisfiable, then the probability that such a verifier will accept B as satisfiable is upper-bounded by a negligible constant—for *any* purported proof.

We want to emphasize the power and somewhat counter-intuitive nature of this theory. One way to understand it is that the information content in the assignment $\vec{a}$ is spread over the proof, which acts as an error-correcting code. Thus, even if y differs from the correct output by *a single bit* (or if one bit is flipped in the intermediate computation), thereby making B not satisfiable, the verifier will catch this fact with high probability.

Unfortunately, this approach is totally impractical:

- *The proof is too long.* It is much too large for the verifier to handle, or for the prover to compute.
- *The protocol is too complicated.* State-of-the-art PCP protocols [7] partially address the concern about proof length, but they are so intricate that a bug-free implementation is unlikely. Unfortunately, in this context, even small bugs can be security-critical.
- *The phrasing of the computation is too primitive.* Even if they have simple control flow, most general-purpose computations are far longer when expressed as Boolean circuits than in, say, C++. Such expansion translates into prohibitive implementation expense.

The rest of this section addresses the above issues in turn.

3.2 Argument systems

A key refinement was proposed by Kilian [19], who constructed an efficient *argument system*. Kilian observed that the verifier need not receive the proof and instead can ask the prover to send the bits of the proof only at the locations where the verifier will check the proof. To prevent the prover from changing answers to later queries to spuriously match earlier answers, the verifier requires the prover to cryptographically *commit* to the proof. This interactive approach alleviates the verifier's burden in dealing with the proof. The prover, however, must now pay for the proof (as before) *and* the substantial overhead of the (Merkle tree) commitment scheme.

We next turn to work by Ishai et al. [18], who had a key insight: for the purposes of an argument system, one can avoid the efficient PCP constructions, which are complex, by exploiting the interactive nature of the protocol. That is, since we're not shipping the proof around, we can do much better by using that fact from the beginning. In more detail, Ishai et al. use a simple PCP construction from [3] (also described in [7, 18]) in which the proof is a linear function; querying the proof means asking the prover to evaluate the function at a verifier-chosen point. If written down fully, the proof would be exponentially-sized: it would be the function's values at every point in the domain. However, the prover does not materialize the proof; it derives the coefficients for its linear function while executing the computation.

For this approach to work, the prover must not modify the function during the protocol. Thus, Ishai et al. propose a primitive: *commitment to a linear function*. The idea is that the prover pre-evaluates the function at points chosen by the verifier, and then the prover's subsequent responses must be consistent with this pre-evaluation.

The advantages of the Ishai et al. scheme are twofold. First, it is very simple (as least compared to other PCP-based protocols). Second, it admits a version that has an expensive setup phase but is efficient during normal use. This version can be made practical through various refinements, which we describe in the next subsection.

3.3 Our refinements

Our version and the base protocol [18] have the same structure, which we outline immediately below.

Setup phase (one time):

1. Verifier sends to prover a circuit, C, that encodes Ψ.
2. Verifier generates random encoded query *helper vectors* and sends them to prover.

Online phase (for each computation instance):

1. Verifier sends input, x, to prover, which executes Ψ on x and returns answer, y.
2. At verifier's request, prover issues a linear commitment to a proof for the following statement: *When C is augmented with input x and purported output y, that new circuit (which is akin to B in §3.1) is satisfiable*. If true, this statement means that Ψ run on input x truly produces the claimed y.
3. Verifier asks prover to respond to the encoded queries. If the proof is correct, a correct verifier certainly accepts the prover's responses and declares y to be correct. If the proof or purported output is incorrect, the verifier rejects y with high probability.

Before describing our refinements, we answer a natural question: if the verifier must send C (or write it down), then how can this process save the verifier work? The answer is amortization. The verifier pays *once* to materialize C, which is roughly as much work as executing C (or Ψ). The verifier then retains only a digest of C. In the online phase, the verifier augments the digest with x and y, and this augmented digest allows the verifier to query the prover and check its results.

(1) Program encoding. We reworked the scheme to encode computations as *tailored arithmetic circuits*, instead of Boolean circuits. The input and output wires of arithmetic circuits take values from a large set (e.g., a finite field or the integers), and the gates are low-level operations like AND or ADD. In our tailoring, gates can also encapsulate a single-output degree-2 function of potentially many inputs (e.g., a dot product). Note that a gate here does not represent a low-level hardware element but rather a modular piece of the computation that enters the verification algorithm as an algebraic constraint.

Although unsurprising theoretically, this refinement is critical practically. For parallelizable numerical computations (e.g., matrix multiplication or FFT), arithmetic circuits can be tailored until they cost nothing: as we demonstrate empirically in §4, executing matrix multiplication as a circuit costs the same as a C++ implementation (so the overhead in this case derives only from proving and verifying). Even computations not so amenable to tailoring are vastly more concise when represented as arithmetic circuits than as Boolean circuits.

To give a sense of our savings in getting to this no-overhead point for matrix multiplication, the move from a Boolean to a non-tailored arithmetic circuit saves, for a fixed m, four orders of magnitude in the number of circuit wires and eight orders of magnitude in the prover's work (which is quadratic in the number of wires). We decrease the proof work by another factor of m^2 with dot product gates (the tailored gates reduce the number of wires from m^3+2m^2 to $2m^2$, lowering the number of required prover operations from more than m^6 to $4m^4$).

(2) Decomposition and batching. If a computation has modular components (e.g., the m^2 dot products in matrix multiplication), the prover can separately encode these *sub-computations* and, with a small change to the proof encoding, allow them to be verified in batch. The prover's work then goes from quadratic in the computation size to (a) quadratic in the sub-computation size plus (b) a cost for combining. For matrix multiplication, the constant factor saved is two orders of magnitude. The verifier, however, pays for this approach (but far less than if it computed locally).

(3) Optimization of cryptographic primitives. To extract query responses from the prover while hiding the queries themselves, the base scheme uses homomorphic encryption. This primitive, though powerful, is fairly expensive, and the base scheme invokes it incessantly. Our refinement is to eliminate these invocations with a special-purpose protocol that blinds queries, radically reducing the prover's costs (by a huge constant).

(4) Amortization and query reuse. Ishai et al. sketched an amortization scheme and a way to trade costs between the online phase and the setup phase. This suggestion inspired us. However, the original did not work out the details, and it required either frequent instances of the setup phase or unacceptably long latency (since verification could happen only once per setup instance). In contrast, we developed a precise specification of an amortization scheme with a one-time setup (we reuse queries

Resource	(A) General	(B) Matrix multiplication
Baseline CPU	N/A	m^3
Circuit CPU	$d \cdot c$	m^3
Verifier CPU (online)	$O((\lvert x\rvert+\lvert y\rvert)\cdot d)$	$(7m^2+44m+52)\cdot r$
Prover CPU (online)	$O((n^2+c)\cdot d)$	$12m^4+m^3+360m^2r$
Network (online)	$O(d)$	$(60+60m)\cdot r$
Verifier CPU (setup)	$O(n^2+d)$	$23\cdot(4m^2+2m)\cdot r+m^2$
Prover storage	$O(n^2+d)$	$30\cdot(4m^2+2m)\cdot r+2m^2$

Figure 1—Costs in our scheme for (A) any computation expressed as an arithmetic circuit and (B) tailored $m \times m$ matrix multiplication. Column (A) gives asymptotic bounds; column (B) gives exact counts. The CPU lines refer to number of operations; network and storage are in terms of 4-byte words. x is the computation's input, y its output. $d \geq 1$ is the number of sub-computations. c denotes the number of operations in each sub-computation, and n the number of wires in a sub-computation. For column (B), the sub-computations are dot products, and the operations counted are field multiplications only (not additions). Thus, $d = m^2$, $c = m$, and $n = 2m$. (We omit additions for simplicity and because multiplications are far more expensive. However, the vertical comparison is still apples-to-apples.) Derivations are in [30].

by keeping them hidden, using the primitive from (3)).

The sketch above is detailed in [30]. We have proved the completeness and soundness of (1) and (2), but (3) and (4) are only heuristics for now.

4 Are the costs plausible?

We now assess the costs of our scheme, first analytically and then by measuring a C++ prototype. Our example computation is matrix multiplication over the field $GF(2^{32})$, and our baseline for comparison is this computation when executed as a compiled C++ program.

Figure 1 lists the costs. The parameter r relates to error probability: a correct verifier always classifies a correct output as such but, with probability p, fails to detect an incorrect output. If executed once, the scheme sketched in §3 has $p < \frac{7}{8}$. To make the overall error, e, negligible, the verifier and prover repeat the scheme r times in parallel.[2] For example, $e < 3 \cdot 10^{-5}$ for $r \approx 80$. Note that reducing e further is relatively inexpensive. For example, when $m = 200$, to bound e by 10^{-4}, 10^{-6}, and 10^{-9}, the verifier's estimated CPU usage, relative to the baseline, ascends 2.6, 3.8, and 5.8, and the percentage effect on the prover is far less.

We measured CPU cycles (using `rdtsc`) consumed by our prototype. Figure 2 shows the measurements, scaled by $r = 80$; they agree with our estimates above.

Take-aways. For matrix multiplication, required storage is a constant ($\approx 60r$) greater than the input; the

[2]Unlike in the theory literature, our error cannot be driven arbitrarily close to zero. It is bounded away from zero by $\frac{1}{2^{32}}$, owing to the way that we implement refinement (2). The details are described in [30].

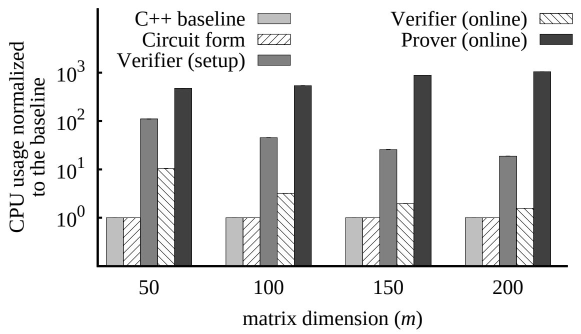

Figure 2—CPU measurements (scaled) of $m \times m$ matrix multiplication, for varying m. The y-axis is log-scaled. The bar heights depict the means of 10 runs, scaled by $r = 80$ (which makes $e \leq 3 \cdot 10^{-5}$); std. deviations are $< 3\%$ of the means.

prover's online CPU cost is $\approx 12m$ times executing locally; and the verifier's online CPU usage is less than executing locally for $m^3 \geq 7m^2r + O(m)$, which, for $r = 80$, holds for $m > 600$. The main costs, then, are storage and CPU at the prover. Are they plausible? Possibly for high-assurance scenarios like those mentioned in §1. For everyday use, likely not. However, as noted before, our costs are dramatically reduced from a naive version, and we are optimistic about more improvements.

5 Where do we go from here?

Our scheme achieves our goals of practicality, simplicity, and unconditional assurance but only over a limited domain. The ultimate goal is perhaps far off, but there are a number of tractable improvements we hope to pursue:

Reducing storage costs. During setup, the verifier installs on the prover many random strings. We think that pseudo-random functions may help alleviate these costs.

Floating point support. Our prototype supports arithmetic circuits over (large) finite fields. But for most applications, efficient floating point arithmetic is essential.

Automatically compiling circuits. Our example involves a hand-tailored circuit for which the decomposition into parallel chunks was apparent. A key next step is a compiler to turn amenable computations expressed in a higher-level formalism into representations of this form.

Longer-term (but more speculative) projects include:

Complicated control flow. A central inquiry is to find a representation of computation that, unlike our current one, deals well with looping or significant control logic.

Prover efficiency. While our results suggest that the prover's current burden is arguably practical in some scenarios, it would obviously be great to reduce these costs.

Larger inputs. An extension in which the verifier handles only a digest of the input, instead of the full input, would be highly useful for large problem instances.

Transferability. Our verifier cannot convince a third party of the correctness of a computation, unless that third party trusts the verifier.

Acknowledgments

We thank Chao Ruan for a very helpful suggestion about decomposition. Marcos K. Aguilera, Lon Ingram, and Ivo Popov gave helpful comments. This work was supported by AFOSR grant FA9550-10-1-0073, ONR grant N00014-09-10757, and NSF grant 1040083.

References

[1] Berkeley Open Infrastructure for Network Computing (BOINC). `http://boinc.berkeley.edu`.

[2] D. P. Anderson et al. SETI@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.

[4] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *ASIACCS*, 2010.

[5] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.

[6] R. M. Bell and Y. Koren. Improved neighborhood-based collaborative filtering. In *KDD Cup and Workshop*, 2007.

[7] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. 36(4):889–974, Dec. 2006.

[8] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[10] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, 2010.

[11] B.-G. Chun and P. Maniatis. Augmented smart phone applications through clone cloud execution. In *HotOS*, 2009.

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

[13] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[14] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

[15] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Comp.*, 18(1):186–208, 1989.

[16] J. Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.

[17] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[18] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Computational Complexity*, 2007.

[19] J. Kilian. Improved efficient arguments (preliminary version). In *CRYPTO*, 1995.

[20] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: A secure two-party computation system. In *USENIX Security*, 2004.

[21] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[22] S. Micali. Computationally sound proofs. *SIAM Journal on Comp.*, 30(4):1253–1298, 2000.

[23] N. Michalakis, R. Soulè, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI*, 2007.

[24] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *NDSS*, 1999.

[25] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 2007.

[26] A.-R. Sadeghi, T. Schneider, and M. Winandy. Token-based cloud computing: secure outsourcing of data and arbitrary computations with lower latency. In *TRUST*, 2010.

[27] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, Burlington, MA, 2009. Chapter 8.

[28] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *USENIX HotCloud*, 2009.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *SOSP*, 2005.

[30] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations (supplement). Technical Report TR-11-16, Dept. of CS, UT Austin, Apr. 2011.

[31] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2005.

[32] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, 1986.