Verifiable Computation with Two or More Clouds

Ran Canetti Ben Riva Guy Rothblum
Tel Aviv University Princeton

February 23, 2011

Abstract

The current move to Cloud Computing raises the need for verifiable delegation of computations, where a weak client delegates his computation to a powerful cloud, while maintaining the ability to verify that the result is correct. Although there are prior solutions to this problem, none of them is yet both general and practical for real-world use.

We propose to extend the model as follows. Instead of using one cloud, the client uses two or more different clouds to perform his computation. The client can verify the correct result of the computation, as long as at least *one* of the clouds is honest. We believe that such extension suits the world of cloud computing where cloud providers have incentives not to collude, and the client is free to use any set of clouds he wants.

Our results are twofold. First, we show two protocols in this model:

1. A computationally sound verifiable computation for any efficiently computable function, with logarithmically many rounds, based on any collision-resistant hash family.
2. A 1-round (2-messages) unconditionally sound verifiable computation for any function computable in log-space uniform $\mathcal{NC}$.

Second, we show that our first protocol works for essentially any sequential program, and we present an implementation of the protocol, called QUIN, for Windows executables. We describe its architecture and experiment with several parameters on *live* clouds.

1 Introduction

These days, the IT world is moving towards the *pay-per-use* paradigm named Cloud Computing. Companies of all sizes reduce their computing assets and shift to a use of computing resources in the clouds. One consequence of this shift is that the IT world outside the clouds is moving to a use of weaker and smaller computer devices, like Virtualized Thin Desktops and Smartphones. Whenever stronger resources are needed, those devices can use the cloud.

Since cloud services are given by an outsider entity, probably with different motivations than of the client's, this model has many security and integrity problems. However, one basic problem is inherent in the model: How can a weak client verify the correctness of the cloud's computation? Can the client be assured that the cloud server follows its declared strategy? These questions are not easily answered by the existing tools of security and cryptography.

There are many possible reasons for a cloud to cheat on his answers. For instance, a cloud would like to improve its revenue by computing things with minimal resources while charging for more. Or, a cloud might benefit somehow from the output of the computations, and therefore it can try to maximize specific results. Or, a disgruntled employee of the cloud provider could modify the executed program. Thus, the client must have some way of verifying the cloud's computation.

This problem of verifiable computation was tackled in many previous works in the theoretical computer science community, most notably by using Probabilistically Checkable Proofs (e.g., [Kil92, Mic00]). Other recent works (e.g., [GGP10, CKV10]) use fully homomorphic encryption and get amortized performance advantages. However, although those solutions are very efficient in terms of asymptotic complexity, they are currently unpractical.

A natural idea is to take the basic idea behind cloud computing, the pay-per-use paradigm, and extend it also for *integrity*. If a client wants to get better assurance of the integrity of his cloud computations, he can pay a little more to get such assurance. And if he already pays a little more, why should it be to the same cloud? He can split his payment among several clouds. They are all accessible on the net anyhow.

One simple way of achieving this goal could be the following (this idea is used also in Grid Computing): Instead of executing his program on one specific cloud provider, the client picks N different cloud providers. Next, the client

asks each of those cloud providers to execute his program and return the output. Now, the client takes the majority of those answers to be his answer. As long as there is a majority of honest cloud providers (even if the client does not know which ones), the client gets the correct answer. The cost of this protocol is $N*$[cost of running one instance of the program]. The main downside of the this protocol is, of course, the need for an honest majority of clouds. In particular, this method requires at least three clouds to be viable. We would prefer a weaker assumption.

2 Our Contributions

A closely related model to ours is the *Refereed Games (RG)* model of Feige and Kilian [FK97], where two unbounded players make contradictory claims and "play" against each other in a protocol where a weak referee can efficiently determine the true claim. This is precisely the same soundness guarantee we require from our protocols for verifiable computation. However, they focus on unbounded players, whereas we are faced with the additional challenge of building protocols where the computational requirements from the servers (i.e., the players) are not much more than those required to compute the function in the first place. In this work we will be interested in refereed games where the complexity of the referee is small (specifically, quasi-linear in the input size), and the complexity of the players is polynomial in the complexity of deciding L. We call such protocols *efficient-players refereed games* (epRG).

We adapt the model of refereed games to the setting of verifiable computation. Stick to the [FK97] notation, we call the cloud's client the *referee* and the cloud servers the *players*. Those names are morally aligned with the verifier and the provers in the *interactive-proof* model. Also, for the description here we restrict attention to the case when there are exactly two players, one honest and one malicious (but the referee does not know which is honest). Our protocols can be extended for more than two players using a simple "tournament" among the servers (see Section 3.2).

We show two new protocols in this model for polynomial-time computations. Both of these protocols have servers/players that are polynomial in the time to compute the function in question, and clients/referees that are *quasi-linear* in the input size. More specifically, we show:

A Practical Computationally-Sound epRG for any $L \in P$. The game requires logarithmically many rounds, and is based on any collision-resistant hash family. The players' work scales only *quasi-linearly* with the complexity of the computation. This protocol is highly generic and can work with any reasonable computation model. Specifically, we describe it with Turing Machines (TM) but it can be easily adapted for real-world models.

This new protocol seems to be qualitatively more practical than known techniques for delegating computation in the single-prover setting. In particular, all known protocols rely either on arithmetization and PCP techniques [Mic00, GKR08], or provide only amortized performance advantages and rely on fully homomorphic encryption [GGP10, CKV10]. Current constructions of both PCP and fully homomorphic encryption are far from being practical. Moreover, all known protocols work with the (arguably less practical) circuit representation of the computation.

Implementation of This Protocol for Windows environment running on X86 CPU. Our implementation, which we call QUIN, works directly with assembly instructions (instead of TM transitions), and we do not require the programmer to write his code in assembly. The programmer can write his code in C language and later on build the program to run with our framework. This is an important feature, since the implementation is almost transparent for the application programmer. We experiment with this prototype on live clouds and show that the overhead is almost *reasonable* for real-world applications. For some applications we get a slowdown factor of "only" 7, compared to the original application. See Section 5 for further details regarding the overheads.

A 1-round (2-message) unconditionally-sound epRG for any function computable in $\mathcal{L}$-uniform $\mathcal{NC}$. To the best of our knowledge, all previous single-round protocols for reliably delegating computation in the single server model [Mic00, GKR08, GGP10, CKV10] require cryptographic assumptions and provide only computational soundness.

Our protocol adapts techniques from the work of Feige and Kilian [FK97], which construct a refereed game but with *inefficient* servers (even, for as low as log-space computations) along with ideas and techniques from the work of Goldwasser, Kalai and Rothblum [GKR08], and some new techniques.

3 Computationally Sound epRG for Any Polynomial Time Computation

We scale down the result of Feige and Kilian [FK97] to get a poly-logarithmic time referee (the client) for any polynomial time program, with unconditionally soundness. Then, we replace their use of arithmetization with Merkle Hash Trees. Although it gives us only computational soundness, replacing arithmetization with Merkle Hash Trees has several advantages. First, hash functions are very efficient, both in software and in general hardware. Second, using Merkle Hash Tree gives us negligible soundness error probability for one execution of the protocol, as opposed to a relatively high (constant) soundness error probability the previous solution has. Third, the resulting protocol is arguably easier to understand and to implement, and therefore might be adopted for real-world uses. Last, [FK97] requires private channels between the referee and the two players. Our protocol uses only public communication.

Given a TM configuration $(state, head, tape)$, let the tuple $(\ state, head, tape[head], \textit{MHT}(tape, head)\ )$ be a *reduced-configuration*, where $\textit{MHT}(tape, head)$ is the Merkle Hash Tree hash values along the path to $head$ and their siblings (this is a Merkle Hash Tree consistency proof). Note that the size of a reduced configuration is poly-logarithmic in the time of the computation.

3.1 The Protocol

We describe the protocol in the Turing Machine model. Given a Collision-Resistant Hash Function, our protocol is the following. The players and the referee have a TM for a language L. The referee sends x to both players and asks whether $x \in L$ or not. In case they answer the same, by the assumption that one of them is honest, the answer is the correct one (and they both declared as winners). Else, the referee continues to a *binary-search* phase. The referee asks the players to send him the number of steps it takes to compute TM(x), takes the smaller answer as the current *bad row* variable, n_b, and sets to 1 the current *good row* variable, n_g. Now, the referee asks for the reduced configuration of the $(n_b - n_g)/2 + n_g$ configuration. If one of the answers is not a valid reduced configuration, he trivially declares the other player as the winner. If answers match, he sets $n_g = (n_b - n_g)/2 + n_g$, otherwise, he sets $n_b = (n_b - n_g)/2 + n_g$. The referee continues the binary search in that way till he gets $n_g + 1 = n_b$. Note that the players do not have to remember *all* the configurations, instead, they can remember only two configurations, one for the last n_g and one for the last $(n_b - n_g)/2 + n_g$. Then, when asked for the next configuration, the player can continue the TM execution from one of those configurations. Overall, in worst case scenario, the players execution time is not much more than executing the program twice.

Now, the referee takes the reduced configuration n_g and the reduced configuration that player 1 sent for row n_b and checks whether those two reduced configurations are consecutive. This can be done in logarithmic time by simulating a single TM transition and checking the path in the Merkle Hash Tree. If they are the consecutive, he declares player 1 as the winner. Otherwise, he declares player 2 as the winner.

Theorem 1 *Assume the hash function in use is collision resistant. Then the above protocol is a computationally sound, full-information, efficient-players refereed game for any language in P. For languages decidable by TMs taking $T(n)$ steps and $S(n)$ space on input x with $|x| = n$, the protocol takes $\log T(n)$ rounds, the referee runs in time $O(n + \kappa \log T(n) \log S(n) + \kappa \log S(n))$ and the players run in time $O(T(n) + \kappa S(n) \log T(n))$, where κ is a security parameter.*

We have several extensions to the basic protocol, such that we can reduce the number of rounds and the communication size.

3.2 More than Two Players

Although we presented our protocol only for two players, this protocol can be extended to any number of players using a *Tournament*, where the referee executes the protocol between each pair of players, in parallel. The winner will be the player that won *all* of his "games". This solution keeps the number of sequential rounds the same but requires $\frac{N \cdot (N-1)}{2}$ different executions of the protocol, and increases the players' running time quadratically.

4 One-round epRG for Any L-uniform $\mathcal{NC}$ Computation

Due to space limitations we only describe the intuition behind our protocol. [GKR08] shows a way to check the entire computation of a circuit C in $depth(C)$ steps, where in each step they reduce the correctness of a lower circuit layer to

the correctness of an higher layer. When they reach the input layer, the verifier can check the correctness of the layer by himself. The protocol uses the standard sum-check protocol as a sub-protocol, and therefore is highly interactive and requires $depth(C)$ steps, each one with $depth(C)$ rounds of interaction. [FK97], shows a one-round refereed game for the sum-check task, but with inefficient players. Our idea is to combine those two protocols and run the protocol of [GKR08] for all layers together, where we use the protocol from [FK97] instead of the standard sum-check protocol. However, there are many subtle difficulties with this idea. Moreover, in order to get a refereed game for $\mathcal{L}$-uniform $\mathcal{NC}$ languages, [GKR08] runs sequentially the basic protocol for several different circuits (that depend on C). Since our goal is one-round refereed game, we show how to compose the basic refereed game in several parallel executions. For a $\mathcal{L}$-uniform $\mathcal{NC}$ circuit C, our refereed game (P_1, P_2, R) requires polynomial time (in $|C|$) players and quasi-linear time referee.

Theorem 2 *Let L be a language in $\mathcal{L}$-uniform $\mathcal{NC}$. For any input x, the protocol $(P_1(x), P_2(x), R(x))$ is an efficient-players refereed game.*

5 Quin: Implementation of the Protocol

We show an implementation of the protocol from Section 3 that enables delegation of X86 executables for Windows environments. Note that Windows is a closed-source OS, and our implementation does not require any changes to the OS. Everything runs in User-Mode.

5.1 The Difficulties and Design Choices

Although the protocol in Section 3 seems easy to describe with Turing Machines, its adaptation for real-world use is quite delicate. An implementation of it must have the following key properties: 1) The framework should be able to execute the program in a completely deterministic way, independently of the OS. 2) We need to be able to execute a program for a given number of steps, stop its execution and store its current state to a file. Later, we need to be able to continue the program execution from any previously stored state. 3) In order to implement the last step of the protocol the client should be able to simulate any instruction given the instruction's operands.

When we were looking for candidate high-level languages we had two key observations: 1) Since our client has to simulate one instruction by itself, we prefer to work with a language that has simple instructions. By simple we mean that any single instruction takes a small and bounded amount of time to compute, as opposed to, for example, a single line of Java code which can theoretically hide a very heavy computation. Ideally, we would like to work with something that is similar to RISC assembly or Java Bytecode. 2) Interpreted languages like Java and Python have complex interpreters that have their own internal states, which are usually not deterministic. E.g., their native code cache or their internal memory management processes like the Garbage Collector. Therefore, storing a state of an interpreted-language program requires subtle changes of its interpreter to somehow make it more deterministic.

In addition, there are many non deterministic events that depend on the operating system, e.g., many OS interfaces return handles to some of the OS internal structures like a pointer to an opened file. Since most operating system calls are also non-deterministic, we require that the program will not make any operating system calls (or at least not non-deterministic ones). We remark that this restriction can be bypassed by writing *function stubs* that preserve that determinism of the program. Currently, we implemented such stubs only for the essential `malloc()` and `free()` functions. Similarly, multi-threading could ruin the determinism, so we require that our delegated program use only one thread.

We decided to implement a prototype directly with X86 assembly, for stand-alone programs. Under reasonable assumptions on the cloud operating system we can achieve all the above functionalities. The atomic instructions are assembly instructions. We believe that using X86 assembly is both cleaner to use and general for further development (e.g., using C++ programs instead of only C).

5.2 System Architecture

The client has a source code in C of a program `prog.exe` that he wishes to delegate. The programmer does not have to write his program in some new or restricted language, he can write his program in the same way he writes any C program. For simplicity of the description we assume that the input to the program is part of the program itself (hard-coded) and that the result of the program is an integer. Specifically, we assume there is a function with the prototype

`int client_main()`. Those restrictions can be easily eliminated if needed (e.g., by using a pre-allocated buffers for input/output before/after the program execution).

Given the source code, the client builds the program using a supplied makefile. This makefile basically links the program with a wrapper code, sets the code base-address to be static and statically links all libraries. We set the code base-address to be static so the loader will load the program to the same memory address for all executions. Similarly, since shared libraries can be loaded to any memory address, we statically link all libraries so they will be (again) loaded to specific memory addresses.

The wrapper code corresponds to the TM initialization. It initializes to zero all the general use registers and the required stack memory, it allocates a large amount of memory to be used as the program heap and calls `client_program()`. Also, the makefile links a code for `malloc()` and `free()` that uses the pre-allocated memory instead of the regular heap. Here we use the fact that Windows allocates large memory segments (e.g. 2 GB) in almost deterministic addresses. After the client builds his program with the supplied makefile, he sends the executable to each of the clouds. Now, the protocol itself starts.

The prototype consists of three main tools: `QuinExecuter`, `QuinClient` and `QuinServer`. The client runs the `QuinClient` on his machine and the clouds run `QuinServer` and `QuinExecuter` on their machines. `QuinExecuter` is a tool for executing a program for a given number of instructions. It can store the program state or continue execution from a previously stored state, it uses a dynamic instrumentation tool named Intel's PIN [LCM+05] and is able to count the number of executed instruction, stop execution and store the state to a file, and, restore a state from a file and continue execution. `QuinClient` and `QuinServer` are python implementations of the protocol itself. The reduced-configuration equivalence in this implementation would be the values of all CPU registers and two hash values. Those hash values are the root values of the Merkle Hash Trees of the current stack memory and the current heap memory. Last, `QuinClient` uses an open-source X86 emulator called PyEmu [PyE] in order simulate X86 instructions.

5.3 Evaluation

Since our goal is to check practicality of the protocol in real-world scenarios, we experimented with live cloud providers, Amazon EC2 and Rackspace Cloud, which are currently among the largest cloud providers. As for the delegated program, we used a simple but very useful program, `Determinant.exe`, that computes the determinant of a given matrix. Although there are algorithms for computing determinant that run in time $O(n^3)$ (for $n * n$ matrix), we used the naive algorithm that runs in time $O(n!)$, and uses $O(n^3)$ space.

We executed several experiments of the full protocol. For each experiment we ran the protocol several times with one cheating cloud that cheats on one out of three randomly chosen states. Those states were chosen to be close to the end of the computation (around 85% of the total number of steps). We added to `QuinServer` a code that, when asked, tries to cheat on all configurations from some given step.

Since there are performance differences between Amazon EC2 and Rackspace Cloud, we conducted two different experiments. In the first (AR), we used one virtual machine on each cloud provider, and in the second (RR), we used two virtual machines that run on the same cloud provider (specifically, Rackspace Cloud). We used a simple laptop as our client.

In Table 5.3 we show the average running times of the protocol for the program `Determinant.exe`. The numbers in the left part of the table represent the running times in seconds on Amazon EC2 and Rackspace Cloud, respectively, separated by slashes. BS-PT denotes the `QuinExecuter` overhead during the binary search and MHT denotes the computation time of the Merkle Hash Tree roots. The overhead factors are over original running times of `Determinant_quin.exe`.

Matrix Size	BS-PT	MHT	AR Total	AR Overhead Factor	RR Total	RR Overhead Factor
10	230/145	121/27	381	190	188	188
11	487/358	165/42	694	33	439	23
12	2017/1773	181/43	2243	8	1676	7

Table 1: QUIN performance for three different matrix sizes, using Amazon EC2 and Rackspace Cloud (AR), and using only Rackspace Cloud (RR).

We can observe that the overhead of the protocol is reduced when the original computation time grows. This suggests that the overhead of the protocol itself is lower, and there are many implementation overheads, mainly from `QuinExecuter` (e.g., because of the internal PIN VM). We believe that a product-level implementation can get much smaller overheads, presumably a factor of 10-20 times slower on average for all computations.

References

[CKV10] Kai Min Chung, Yael Kalai, and Salil Vadhan, *Improved delegation of computation using fully homomorphic encryption*, CRYPTO '10: Proceedings of the 30th annual conference on Advances in cryptology, Springer-Verlag, 2010, pp. 483–501.

[FK97] Uriel Feige and Joe Kilian, *Making games short (extended abstract)*, STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, ACM, 1997, pp. 506–516.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno, *Non-interactive verifiable computing: outsourcing computation to untrusted workers*, CRYPTO '10: Proceedings of the 30th annual conference on Advances in cryptology, Springer-Verlag, 2010, pp. 465–482.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum, *Delegating computation: interactive proofs for muggles*, STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing, ACM, 2008, pp. 113–122.

[Kil92] Joe Kilian, *A note on efficient zero-knowledge proofs and arguments (extended abstract)*, STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, ACM, 1992, pp. 723–732.

[LCM+05] Chi Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM, 2005, pp. 190–200.

[Mic00] Silvio Micali, *Computationally sound proofs*, SIAM J. Comput. **30** (2000), 1253–1298.

[PyE] *PyEmu, a python IA-32 emulator,* `http://code.google.com/p/pyemu`.