Practical Delegation of Computation using Multiple Servers

Ran Canetti[*]
Tel Aviv University & Boston University
canetti@tau.ac.il

Ben Riva[*]
Tel Aviv University
benriva@tau.ac.il

Guy N. Rothblum[†]
Microsoft Research, Silicon Valley Campus
rothblum@alum.mit.edu

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 server, 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 demonstrate a relatively efficient and general solution where the client delegates the computation to *several* servers, and is guaranteed to determine the correct answer as long as even a *single* server is honest. We show:

- A protocol for any efficiently computable function, with logarithmically many rounds, based on any collision-resistant hash family. The protocol is set in terms of Turing Machines but can be adapted to other computation models.

- An adaptation of the protocol for the X86 computation model and a prototype implementation, called Quin, for Windows executables. We describe the architecture of Quin and experiment with several parameters on live clouds. We show that the protocol is practical, can work with nowadays clouds, and is efficient both for the servers and for the client.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Algorithms, Experimentation, Security

[*]Research supported by the Check Point Institute for Information Security.

[†]Most of this work was done while the author was at the Department of Computer Science at Princeton University and Supported by NSF Grant CCF-0832797 and by a Computing Innovation Fellowship.

Keywords

Verifiable Computation, Cloud Computing

1. INTRODUCTION

These days, the IT world is moving towards the *pay-per-use* paradigm (aka Cloud Computing). Companies of all sizes reduce their computing assets and shift to a use of computing resources in the clouds. This shift is predicted to increase in the near future and become a significant portion of the IT market. 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, Tablet PCs and Smartphones. Whenever stronger resources are needed, those devices will use the cloud.

Since cloud services are often given by an outside entity with different interests than those of the client, this model carries within it many security problems. One problem, however, stands out as an inherent and very basic one: How can the client verify the correctness of the cloud's computation? This question is not easily answered by the existing tools of security and cryptography.

There are many possible reasons for a cloud to answer incorrectly. For instance, a cloud would like to improve its revenue by investing less resources while charging for more. Or, a cloud might benefit somehow from certain outputs of the computation, thus it can try to maximize specific results. Or, a disgruntled employee of the cloud provider could modify the executed program. Consequently, the client must be able to verify the correctness of that result. To be effective, we would like the verification process to use considerably *less* resources than those required to actually perform the computation from scratch.

This problem has been considered extensively in the theoretical computer science community, most notably by using Probabilistically Checkable Proofs (e.g., [15, 20]). (See Section 1.2.) However, although those solutions are very efficient in terms of asymptotic complexity, they are currently impractical.

A natural idea, pursued in this work, is to take the basic concept 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 as they are all accessible on the net anyhow.

One simple way of achieving this goal could be to use a number of clouds, and then have the client execute the program by himself in case of inconsistency. However, what about the case where it is impossible or impractical for the client to execute the program even in case of inconsistency?

For this case, the following idea has been proposed: 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 plurality value of those answers to be the correct 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. Indeed, this approach is used in Grid Computing, e.g., BOINC [1].

The main downside of this approach is, of course, the need for an honest majority of clouds. In particular, this method requires at least three clouds to be viable. Can one do better? In particular, can we get practical efficiency improvements over the single cloud case, with access to only *two* clouds, only one of which is honest and for a client that *cannot* compute the function by himself?

We provide a positive answer to this question. Specifically, we are interested in the following model: The client asks for the result of the function $f(x)$ from two (or more) cloud servers. In case they make contradictory claims about $f(x)$, the client engages in a protocol with each of the servers, at the end of which the client can efficiently determine the true claim as long as there is at least *one* honest server. As for efficiency, we require that the computational requirements from an honest server are not much more than those required to compute the function in the first place, and that the client's running time is much smaller than the time required to compute the function. We call this model Refereed Delegation of Computation (RDoC) since the client acts like a referee.

Our model is closely related to the *Refereed Games (RG)* model of Feige and Kilian [10] where they focus on two unbounded competing servers and polynomial time referee/client. However, we are faced with the additional challenges of building protocols with efficient honest servers, with a super-efficient client and for any number of servers. In fact, our model can be also considered as *refereed games with multiple efficient servers* and super-efficient clients.

We remark that although our main motivation is a solution for the setting of cloud computing, our results are also useful for other client-server applications. E.g., Grid computing, or when using several processors for redundancy in realtime systems. (Indeed in the latter case there is a need to determine the identities of the faulty processors.)

1.1 Our Contributions

For the description here we restrict attention to the case when there are exactly two servers, one honest and one malicious (but the client does not know which is honest). We later show how to extend our protocol for more than two servers. We show an efficient and *full-information* RDoC protocol for any efficiently computable function, with logarithmically many rounds, based on any collision-resistant hash function family. Here, by full information we mean that the servers can see the full internal state of the client and the communication between the client and the servers is public. The honest servers' work grows only *quasi-linearly* with the complexity of the computation. This protocol is highly generic and can work with any reasonable computation model.

Previously, Feige and Kilian [10] gave a *private information* but unconditionally sound protocol with similar parameters. It also follows from their results that it is unlikely that an information-theoretically sound full-information protocol with similar performance can be obtained (in particular, this is impossible unless all of $\mathcal{P}$ can be computed in poly-logarithmic space).

Our protocol, which builds directly on the protocol of Feige and Kilian, is qualitatively more practical than known techniques for delegating computation in the single-server setting. In particular, all known protocols rely either on arithmetization and PCP techniques [20, 12], or provide only amortized performance advantages and rely on fully homomorphic encryption [11, 9]. Neither approach is currently viable in practice. Moreover, all known protocols work with the (arguably less practical) circuit representation of the computation.

At high level, in this protocol the client searches (using binary search) for inconsistencies between the intermediate states of the two servers' computations. On finding an inconsistency, the client can detect the cheater by performing only a *single step* of the delegated computation. The collision-resistant hash functions are used to allow the servers to "commit" to the (large) intermediate internal states of the computation using small commitments.

In addition, in contrast with prior protocols, our protocol is full-information (or *public coins*). This means that, in a setting where messages between clients and servers are digitally signed, the protocol guarantees that as soon as a server cheats, the client detects the cheating and obtains a publicly verifiable proof of this fact. This is a strong guarantee: we view the servers as *rational* self-interested parties (say cloud computing service providers). An honest server can convince even third parties that all of the cheating servers are cheaters. Assuming that pointing out cheaters is rewarded and cheating is penalized, playing honestly becomes (always) a dominant strategy for rational servers.

We stress that in case all servers agree upon the result, there are no overheads for the servers, nor for the client. The overhead kicks in only in case the servers do not agree. In this case, the overhead is only poly-logarithmic in the size of the computation.

Quin. We adapt the protocol for the X86 computation model. A simplistic adaptation would be to simply run a Turing Machine simulator for X86; but this would result in highly inefficient code. Instead, we adapt the protocol to the X86 computation model: Instead of Turing Machine transitions we have assembly instructions, and instead of working tapes we have the machine's stack and heap memories. This highly improves the practicality of the protocol, and shows its flexibility.

We present a prototype implementation of this adaptation for the Windows environment. Our implementation, which we call Quin, works directly with X86 assembly instructions. However, 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 it makes the implementation almost transparent for the application programmer. Moreover, with small modifications to our prototype, it allows future support for other languages that compile to X86 assembly.

Since our protocol requires the ability to execute a program for a given number of steps, stop its execution and store its state to a file, and later on, be able to resume execution from a stored state, we use a binary instrumentation framework, Intel's PIN [17], to add assembly instructions for counting the number of executed instructions and comparing it to a given threshold. Then, we combine several low-level techniques that work directly with the process memory in order to efficiently dump/restore a state to/from a file.

Another requirement of our protocol is that the client should be able to simulate one step of the computation by himself, given only a small part of some stored state. While for Turing Machines this seems easy, for high-level languages it can be complicated (e.g., in Java, a single step can be a heavy computation that depends on many variables). However, focusing only on X86 instructions that are more "simple" and depend on a small number of variables (or memory), we can meet this requirement by using an X86 emulator that we feed with the required registers and memory data. We extend the Python X86 emulator PyEmu [3] to support emulation of instructions given only a remote access to the process memory (since we do not want to transfer the entire memory to the client).

Last, our protocol assumes that each execution of the delegated program is fully deterministic (i.e. for a fixed input, the Turing Machine's tableau must be the same for each execution of the machine on this input), and therefore, the delegated program has to use only deterministic system calls. However, useful library calls like `malloc()` are not deterministic and depend on the state of the operating system. We overcome this problem by implementing deterministic versions of `malloc()` and `free()` that use pre-allocated memory that is allocated in a nearly deterministic address. We suggest how to extend the idea of function stubs for other system/library calls.

We experiment with this prototype on live clouds and show that the overhead is almost *reasonable* for real-world applications. For some parameters we get a slowdown factor of "only" 8, compared to the original application. Furthermore, the result implies that a large portion of the overhead is due to implementation issues. Thus, a product-level implementation of the protocol could achieve much smaller overheads. See Section 4.4 for further details regarding the overheads.

1.2 Related Work

Prior work has studied the question of proving the correctness of general computations. (Most previous works focused on interactive proofs between a verifier and a prover. However, the problems are closely related: Given an interactive protocol for proving the correctness of a computation of f, one can easily get verifiable delegation of computation by asking the server for $y = f(x)$ and a proof that y is the correct result.) Babai *et al.* [7] consider this question in a setting where the prover is a non-adaptive oracle. Kilian [15] and Micali [20] build on their techniques and show efficient computationally sound protocols, whose security is based on cryptographic assumptions and where soundness holds only against computationally bounded cheating provers. Micali gets a non-interactive computationally sound proof based on the existence of a Random Oracle whereas Kilian gets a four-message interactive computationally sound proof assuming the existence of collision resistance hash family. Goldwasser *et. al.* [12] present an information theoretically sound interactive proof protocol for verifiable computation for any language in $\mathcal{L}$-uniform $\mathcal{NC}$.

We note that, with the exception of [12], the above works are based on Probabilistic Checkable Proofs (PCP) [6]. Although constructions of PCP are very efficient by means of asymptotic complexity, they are far from being practical.

Gennaro *et al.* [11], Chung *et. al.* [9] and Applebaum *et al.* [5] consider a model with a pre-processing stage. Based on the existence of a fully homomorphic encryption, they construct computationally sound protocols, where in an offline pre-processing stage the verifier runs in time proportional to the size of the computation. Afterwards, in a one-round online stage, the verifier (using the result of the pre-processing stage) runs in time proportional to the size of its inputs and the computation results. In these works, as long as the verifier does not encounter cheating provers, the same pre-processing information can be used in multiple rounds, yielding improved amortized complexity.

We remark that in all the above works, the verifier has to know all the inputs, including any randomness that the server uses for the computation (our protocol also shares this requirement).

A related proof model with several provers is the model of Multi-Prover Interactive Proofs, suggested by Ben-Or *et al.* [8]. In this model, even if *all* of the provers cheat, the verifier will detect that they are cheating. However, soundness is guaranteed assuming that malicious provers *cannot* communicate or coordinate their strategies during the protocol. This is in contrast to the refereed games of Feige and Kilian [10] and to our model, where soundness is guaranteed as long as one server is honest, even if all malicious servers communicate *during* the protocol. In addition, the client learns who are the cheating provers.

As for more applied prior work, there are works that in some sense restrict the computation type (e.g. [21] for computations that have short intermediate states, or, [13] for functions with easy to sample domains) and there are several works that use *trusted hardware* that allow almost any type of computation (e.g. [22, 18]). In the latter one, the client trusts some small tamper-resistant hardware (e.g. a tamper-proof coprocessor) that the server has, and using this hardware the server can prove the correctness of its computation. We note that using tamper-resistant hardware is conceptually similar to trusting another weak third-party (in this case, the hardware manufacturer). Nonetheless, this direction is very promising by means of practicality and generality.

There are works that consider verification of peers in distributed computations (e.g., [14] for verifying peers that are state machines) but usually these works assume the different players (i.e. the servers in our model) communicate among themselves or through a trusted server. In the model of cloud computing, we prefer that all communication will be held only between the client and the servers. That way, the servers do not know whom they are "playing" against (or even against how many) and the client does not have to trust any third party servers.

Our construction is not based on probabilistically checkable proofs or Fully Homomorphic Encryption, which are not practical yet, nor does it rely on trusted hardware. Furthermore, it does not require the *arguably* complex transformation of a Turing Machine program to a boolean circuit.

1.3 Organization

In Section 2 we define the model of Refereed Delegation of Computation (RDoC) and show how to extend RDoC with two servers to N servers. In Section 3.3 we present in detail our RDoC protocol. In Section 4 we describe the difficulties of implementing the protocol, our design choices, the adaptation of the protocol for X86 CPU and the architecture of Quin. In Section 4.4 we show experimental results of Quin on live clouds and in Section 4.5 we outline several future improvements.

2. REFEREED DELEGATION OF COMPUTATION

A refereed delegation of computation for a function f is a protocol between a client (or a referee) R and N servers $P_1, P_2, \ldots, P_N$. All parties may use local randomness. The client and the servers receive an input x. The servers claim different results for the computation of $f(x)$ and the client should be able to determine the correct answer with high probability. We assume that at least one of the servers is honest.

Definition 1 (Refereed Delegation of Computation). *Let $(P_1, P_2, \ldots, P_N, R)$ be an ε-RDoC with N servers for a function f if the following holds:*

- *For any input x and for all $i \in \{1, \ldots, N\}$, if server P_i is honest then for any $P_1^*, \ldots, P_{i-1}^*, P_{i+1}^*, \ldots, P_N^*$ the output of R is $f(x)$ w.p. at least $1 - \varepsilon$.*
- *The complexity of the client is at most quasi-linear in $|x|$ and the complexity of the (honest) servers is polynomial in the complexity of evaluating f.*

If soundness holds only for polynomially bounded (in $|x|$) servers then we say that it is a *computationally sound RDoC*. Furthermore, if the client starts by sending all its local random choices to *all* servers, and if all the communication between the client and the servers is public, we call it a *full-information* RDoC.

For completeness of the description, we briefly review the model of Refereed Games [10]. A refereed game (RG) for a language L is a protocol between a referee R and two competing *unbounded* servers P_1 and P_2. All three parties may use local randomness. The referee and the servers receive $x \in \{0,1\}^*$. Without loss of generality we can assume P_1 claims that $x \in L$ and P_2 claims that $x \notin L$, and the referee should be able to determine the correct answer with probability at least 2/3.

From Two Servers to N Servers. In the next sections we only discuss the case where there are two servers. Here we show how, given any RDoC with two servers and negligible error probability, one can construct a RDoC with N servers and negligible error probability, where we only need to assume that at least *one* of them is honest.

The idea is to execute the RDoC with two servers between each pair of servers. By the soundness of the RDoC with two servers, with high probability there exists an honest server P_i that convinces the client in *all* of his "games". The client outputs the claimed result of P_i.

This solution can be executed in parallel for all pairs, and therefore keeps the number of rounds the same. However, it requires $\frac{N \cdot (N-1)}{2}$ different executions of the protocol.

3. PROTOCOL FOR TURING MACHINES

3.1 Preliminaries: Merkle Hash Tree

Merkle Hash Tree (MHT) [19] is a common primitive that allows one to hash a long string of n characters in a where the hash can later be used to reveal any part of the string and supply a short proof of consistency. Given a collision-resistant hash function H and a string str of length n, the tree has n leaf nodes where leaf node i has the value of $H(str[i])$. The next level has the values of $H(H(str[i]) \circ H(str[i+1]))$ for $i = 1, 3, \ldots n-1$, and so on for the other levels. The highest level, the root, is the hash for the full string str. The proof of consistency for character i is the hash values along the path from the root to $H(str[i])$ and their siblings.

Given a Merkle Hash Tree of string str, denote by $\mathit{MH}_{root}(str)$ the value of the root, by $\mathit{MH}_{proof}(str, i)$ the proof of consistency for i-th character, and by VerifyMHProof$(root, i, str_i, p)$ the verification function that given a claimed $p = \mathit{MH}_{proof}(str, i)$ outputs True if p is a valid proof of consistency, and False otherwise. Note that the size of $\mathit{MH}_{root}(str)$ depends on the output length of the hash function and the size of $\mathit{MH}_{proof}(str, i)$ is logarithmic in the length of str.

Denote by $str_{[i]=x}$ the string that equals to str except for the i-th character that is x. We observe the following property of Merkle Hash Trees: Given a proof of consistency p for the i-th character of str, anyone can efficiently (i.e. in logarithmic time in the length of str) compute $\mathit{MH}_{root}(str_{[i]=x})$. This can be done by computing the hash of x and then iteratively computing the hashes along the path from i to the root.

3.2 Reduced Turing Machine Configuration

Given a Turing Machine configuration $(state, head, tape)$, let the *reduced-configuration* of $(state, head, tape)$ be the tuple

$$(\ state, head, tape[head], \mathit{MH}_{root}(tape)\ ).$$

For simplicity of the description, we denote by k the maximal length of the tape during the execution of the Turing Machine. Note that the size of the reduced-configuration is logarithmic in k, and therefore for computation of size T, it is at most logarithmic in T.

Given two reduced Turing Machine configurations $rc_1 = (s_1, h_1, v_1, r_1)$ and $rc_2 = (s_2, h_2, v_2, r_2)$ that are claimed to be consecutive, and a proof of consistency of the first configuration $p_1 = \mathit{MH}_{proof}(t, h_1)$ where t is the tape of configuration rc_1, one can efficiently verify this claim by checking the following:

1. Verify that VerifyMHProof(r_1, h_1, v_1, p_1) is True.
2. Simulate a single step of the Turing Machine on s_1, h_1, v_1 and get the next state s', new head position h' and the written character v'
3. Verify that $s' == s_2$ and $h' == h_2$.
4. Using p_1 and v', compute $r' = \mathit{MH}_{root}(t_{[h_1]=v'})$. (This can be done without knowing t using the previous observation.)
5. Verify that $r' == r_2$.

If one of the above checks fail, then the claim is false.

Denote by VerifyReducedStep(rc_1, rc_2, p_1) the function that given two reduced configurations rc_1, rc_2 and proof of consistency p_1 outputs False if any of the above checks fails, and True otherwise.

3.3 The Protocol

We base our protocol on the work of Feige and Kilian [10] where they present a refereed game with polynomial number of rounds and private communication channels (therefore not full information) for any *EXPTIME* language. Their protocol uses *arithmetization* for consistency checks and then takes advantage of the *locality* property of a single Turing Machine step (each Turing Machine transition uses only $O(1)$ *local* information: the current state, the current head position and the current character). In their protocol for languages in *EXPTIME*, the referee is polynomial in the length of the input x.

Their construction can be directly scaled-down for languages in P, yielding a protocol where the servers are polynomial in the input size and the referee is quasi-linear. Correctness remains unconditional. However, the protocol requires private communication channels between the referee and the two servers.

We modify their scaled-down protocol (for $\mathcal{P}$ languages) by replacing the use of arithmetization with Merkle Hash Trees. Although it gives only computational soundness, it greatly simplifies the protocol and gives us a negligible error probability for even one execution of the protocol. Since the main overhead of the protocol would be retrieving different states of the execution, this property is highly important. Our protocol is full-information and in particular does not require private communication channels. In a setting where messages between the players are digitally signed, the client can obtain a publicly verifiable proof that a server is cheating. (Note that in case of private channels/private-information, and specifically in the protocol of [10], colluding referee and server can forge together a transcript that incriminates an honest server of cheating. In the full-information model it is not possible.)

Given a Collision-Resistant Hash Function, our protocol is the following. The client requests each server to execute the Turing Machine that computes $f(x)$. In case they answer the same, by the assumption that one of them is honest, the answer is the correct one. Else, the client continues to a *binary-search* phase. The client asks the servers to send him the number of steps it takes to compute $f(x)$), takes the smaller answer as the current *bad row* variable, n_b, and sets to 1 the current *good row* variable, n_g. The client also asks for the maximal length of their stored configurations and takes the bigger answer to be k. Now, the client 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, the client outputs the value of $f(x)$ of the other server (this is the honest server). 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 client continues the binary search in that way till he gets $n_g + 1 = n_b$. Note that the servers 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 server can continue the TM execution from one of those configurations. Overall, in worst case scenario, the servers execution time is not much more than executing the program twice.

Now, the client asks Server 1 for the consistency proof for configuration n_g, i.e. $p = MH_{proof}(t_{n_g}, h_{n_g})$. Denote by rc_{n_g} and rc_{n_b} the reduced configuration that Server 1 sent to the client. If VerifyReducedStep(rc_{n_g}, rc_{n_b}, p) is True, the client outputs the value of $f(x)$ of Server 1. Otherwise, he outputs the value of $f(x)$ of Server 2.

Overall we have:

THEOREM 1. *Assume the hash function in use is collision resistant. Then the above protocol is a computationally sound, full-information, RDoC with two servers and with negligible soundness ε for any function computable in polynomial time. For functions that can be computed by TMs taking $T(n)$ steps and $S(n)$ space on input x with $|x| = n$, the protocol takes $\log T(n) + 3$ rounds, the client runs in time $O(n + \kappa \log T(n) + \kappa \log S(n))$ and the servers run in time $O(T(n) + \kappa S(n) \log T(n))$, where κ is a security parameter.*

Note that in case both servers are honest, there is no overheads.

PROOF (SKETCH). By the specification of the protocol, if Server 1 is honest then it always successfully convinces the client (no matter what a malicious Server 2 does).

A malicious Server 1 can deceive the client to output a false value only if it can generate a fake consistency proof p that has the same root of the Merkle Hash Tree as in the correct reduced configuration n_g. Let π be the above protocol, and let π' be the above protocol with the following change: in the last step the client asks for the consistency proof from *both* servers and outputs the value of the server that was honest (i.e., its consistency proof was consistent with its reduced configurations n_g and n_b) or the output of Server 1 if they were both "honest" (i.e., both proofs were valid). Let ϵ be the probability that Server 1 cheats in π and let ϵ' be the probability Server 1 cheats in π'. Since a malicious Server 1 from protocol π can also cheat in protocol π' then $\epsilon \leq \epsilon'$. Now, assuming Server 1 in π' is malicious, then the client in protocol π' receives two different consistency proofs with the same root, thus, he gets a collision in some node along the path to index h_{n_g}. By the security of the collision resistant hash function, this can only happen with a negligible probability. Therefore, ϵ is negligible. $\square$

3.4 Extensions

Reducing the number of rounds. In some scenarios, the number of rounds might still be the bottle-neck of the protocol. We can reduce the number of rounds by permitting larger messages and longer running time of the servers.

For any constant number t we can reduce the number of rounds to $\log_{t+1} T(n)$ (but slightly increase the communication size, by a factor of t) using the following idea. Instead of asking the servers for only one reduced configuration in each round, the client asks for t reduced configurations. Specifically, the client asks for the t steps that are equally spread between n_g and n_b. I.e., given $n_g = 100, n_b = 200, t = 4$, the client asks for $120, 140, 160$ and 180. Similar to the protocol from Section 3.3, the client updates n_g and n_b according to the servers' answers and continues to the next iteration of the binary (or $(t+1)$-ary) search.

More than two servers. In addition to the general method for extending the protocol to N servers from Section 2, we can extend this specific protocol also in the following way. The client executes a *Playoff* between all servers. In the first round, the client executes the protocol from Section 3.3 with all servers (he can do that because the protocol uses only public communication), where he marks a row as a good row only if all answers for this row match. At the end of the binary search, the client checks if the reduced configurations are consecutive for each one of the servers. After the execution of this protocol, at least one malicious server will be caught lying and will be declared as a cheater. The client continues to the next round with the other servers, again, executes the protocol to find at least one cheater and then excludes him (or them) from the next rounds. The protocol ends when all the remaining servers agree on the output.

Since the client excludes at least one malicious server in each round of the playoff, the number of rounds is bounded by the number of malicious servers.

Supporting large datasets. In order to support computations which use large datasets (e.g. Databases), we can use the Merkle Hash Trees again. We assume the client knows the root of the MHT of the data (either because he delegated the data in the past, or by keeping track of changes of the root). We modify the protocol from Section 3.3 to work with another tape that includes the dataset and we add to the reduced configuration also the root of the MHT of this tape, the current head position and the MHT proof for its current character. The rest of the protocol is the same. Note that the client has to know the root of the MHT at the beginning of the computation in order to be able to verify the initial configuration.

Reducing the server's overhead. In the worst case, the execution time of the servers in the protocol from Section 3.3 is not much more than executing the program twice. The server "pays" one execution time only for counting the number of instructions and then half execution time for getting to the middle configuration.

If we allow the servers to store more configurations then we can reduce the computation time for getting the middle configuration. Instead of just counting the number of instructions, the server also stores intermediate configurations. Let i be the current step of the execution. During the count of instructions, the server remembers the last three configurations for steps $i_3 = 2^{\lfloor \log(i) \rfloor}, i_1 = i_3/2$ and $i_2 = (i_3 - i_1)/2 + i_1$. Note that these values are changed only when $i = 2 * i_3$, and by storing also configuration $i_3 + i_1$, the server can compute them efficiently on-the-fly. In other words, the server always remembers at most four configurations and updates them during the counting (without rewinding the computation). Now, when requested for the middle configuration, the server takes the nearest stored configuration (either for i_1 or i_2) and continues the execution from that configuration. Overall, it reduces the computation of the middle configuration from half to one sixth in the worst case (resulting in overhead of $1\frac{4}{6}$ instead of 2). Repeating this method can reduce this overhead even more but with the price of larger storage.

4. QUIN: ADAPTATION AND IMPLEMENTATION OF THE PROTOCOL FOR X86

We show how to adapt the protocol from Section 3.3 for X86 CPU, and we present a prototype implementation that enables delegation of X86 executables for the Windows environment. Note that Windows is a closed-source OS, and our implementation does not require any changes to the OS. Everything runs in User-Mode. See [4] for the source code of the prototype.

4.1 The Difficulties and Design Choices

Although the protocol in Section 3.3 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:

- **Determinism:** The protocol highly depends on the determinism of the execution, therefore, the framework should be able to execute the program in a completely deterministic way, independently of the OS.

- **Stop, store and continue execution:** The protocol requires the ability 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.

  At first, it seems obvious we can use a debugger. However, this is not the common functionality most debuggers provide. We require the ability to continue from *any* previously stored state, whereas a normal debugger stops on a pre-determined state and can continue execution only from that state.

- **Simulate an instruction:** In order to implement the last step of the protocol (VerifyReducedStep()), 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:

- 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.

- 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 non-deterministic operating system or standard library calls. 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. See Section 4.5 for further discussion. Similarly, multi-threading could ruin the determinism, so we require that our delegated program use only one thread.

4.2 Adaptation of the Protocol for X86

We decided to implement the prototype directly with X86 assembly, for stand-alone programs. 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).

Instead of the Turing Machine's transitions table, the computation is described by assembly instructions. Each step of the protocol is now an execution of a single X86 instruction. When the client needs to compute a single step by himself, he should be able to execute a single X86 instruction given the needed registers and memory. We remark that some X86 instructions are non-deterministic by definition (e.g. CPUID, RDTSC) and we assume the program does not use them (this can be restricted during build).

The equivalent of a Turing Machine state is the current values of all the CPU registers (e.g., `eax`, `ebx`, etc). We initialize those registers before the execution of the program.

Last, we replace the Turing Machine working tapes with the process' stack and heap memories. We assume these memories are initialized to zero, which is the equivalent of an empty tape. Under reasonable assumptions on the cloud operating system we can work with fixed base-addresses for those memories and therefore get fully deterministic memory operations (i.e., even pointers to the memory will have deterministic addresses for each execution of the program).

The reduced-configuration equivalence in this model would be the values of all CPU registers (the equivalent to the Turing Machine's state and head position) and two hash values. Those hash values are the root values of the Merkle Hash Trees of the current stack and heap memories.

The proof of correctness of the resulting protocol is similar to the proof of protocol from Section 3.3.

4.3 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 here 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_program()`. Those restrictions can be easily eliminated (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 operating system's 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 Turing Machine 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. See Figure 1 for a pseudo-code of the wrapper function.

After the client builds his program with the supplied makefile, he sends the executable to each of the servers. Now, the protocol itself starts.

```
int main(){
    int i;
    int result;
    void *heap;
    char *stack;
    /* Init stack memory          */
    /* STACK_SIZE is usually 2^20 */
    _asm{
        mov stack, esp
    }
    for(i=0;i<STACK_SIZE;i++){
        *(stack-i) = 0;
    }
    /* Init heap memory           */
    /* HEAP_SIZE is usually 2^30  */
    heap = calloc(HEAP_SIZE);
    /* Init all registers         */
    _asm{
        mov eax,0
        mov ebx,0
        ...
        /* Setting esp to be the
           start of a memory page  */
        mov eax, esp
        and eax, 0xfff
        sub esp, eax
    }
    /* Execute original program    */
    _asm{
        call client_program()
        mov result, eax
    }
}
```

Figure 1: Quin wrapper function.

The prototype consists of three main tools: `QuinExecuter`, `QuinClient` and `QuinServer`. The client runs the `QuinClient` on his machine and the servers 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. `QuinClient` and `QuinServer` are python implementations of the protocol itself.

The Merkle Hash Tree is computed with granularity of a 4Kbyte (a page-size), i.e., the lowest level of the tree are hashes of 4Kbyte segments. (This is why our wrapper function moves the `esp` register to the beginning of a page.)

4.3.1 QuinExecuter

For executing a program for a given number of instructions, we need a way to count in real-time the number of executed instructions. There are several ways to do that:

- Use the CPU internal instructions counters: this is a very efficient method, but requires to change the OS in order to differentiate between our instructions and other OS instructions.
- Extend the compiler: instead of emitting the programmer's code as is, add to it few assembly instructions that increment a counter before each original assembly instruction.

  The downside is that we have to do it for every high level language compiler although the protocol itself works with the assembly instructions.
- Use OS debug API: execute the program *step-by-step*. Since it is *out-of-process* debugging, it is very inefficient.
- Static instrumentation: given an executable, transform it to a new executable with the other needed instructions (i.e., that increment a counter).
- Dynamic instrumentation: similar to static instrumentation, but done in run-time.

Our method of choice is to use a dynamic instrumentation tool, specifically Intel's PIN [17]. Although dynamic instrumentation has potential for substantial overheads over static instrumentation, PIN has many benefits for our use. PIN is very efficient (compared to other dynamic instrumentation frameworks), very convenient for rapid development, well supported by Intel, and most importantly, works the same in Windows and Linux environments. PIN runs the program inside a Virtual Machine and uses Just-In-Time (JIT) compilation of the instrumented code. The PIN developer writes a piece of code that is called a *PIN-tool*, which is a description of where to instrument and what code to put there. In order to run the PIN-tool, only four binary files are needed (`.exe` and `.dll` files), and no setup is required. The PIN-tool itself is another `.dll` binary.

`QuinExecuter` is basically a PIN-tool. The naive way to get our goal is that `QuinExecuter` adds instrumented code for counting steps and checking whether we reached the needed number of steps. This would be the main overhead of our implementation, as for each executed instruction of the original executable we add code that increments a 64 bit counter and checks whether this counter reached some threshold. This means that *for each* assembly instruction of the original program we add around 10-15 new assembly instructions. Our implementation actually uses a simple heuristic to reduce this overhead. Instead of adding the code for each instruction, we add the code only once for each *basic-block* (a sequence of code with only one entry point and one exit point). For each basic block we increment the counter according to the basic block's number of instructions. Since we want to be correct with a granularity of a single instruction, when we get close enough to our threshold (say, 500 instructions below the threshold), we re-instrument the code and add our code before each instruction (as in the naive way). We note that since we are interested in precise granularity for all types of computations, this heuristic seems to have the best tradeoff between efficiency and accuracy.

After it reaches the needed number of steps, it dumps the current state of the memories (we separate the stack and the heap for efficiency) and all the registers values to a file. When requested, it can do the opposite, start a new process, restore a state from a given file and continue the execution of the process from this state. The way it does those operations is by reading and writing directly the stack and the heap whenever needed.

In order to handle memory, we use the following two methods:

- **For stack memory:** In order to get full determinism, all memory should be initialized before use. Since the operating system do not initialize the stack by itself, our wrapper function does it before the call to `client_program()`. We use the maximal stack size that is defined during the program linkage.
- **For heap memory:** As already noted, we implemented our own `malloc()` and `free()`. Those functions uses a pre-allocated memory, and stores its state in a pre-defined memory areas, so when needed to dump or restore a state, our PIN-tool can work directly with those internal structures.

  For the protocol itself, we define the maximal size of the heap memory to be constant (e.g., 2^{30} bytes). As this memory is pre-allocated, we set this memory to zero before the execution of `client_program()`.

`QuinExecuter` is implemented in C and Assembly, and uses several low-level techniques in order to get efficient instrumented code.

We note that since we assume the client has the source-code of the delegated program, we could have chosen to change the build process in order to create a different executable that can directly give us the above functionalities. However, our future goal is to replace the use of our makefile (that adds the wrapper function) with another binary instrumentation, in a way that our PIN-tool could work directly with any stand-alone executable (e.g. commodity software). This PIN-tool will find `main()` and wrap it with our wrapper function using instrumentation. In addition, it will locate `malloc()` and `free()` and replace them with our implementation of those functions.

4.3.2 QuinServer

`QuinServer` is a pure Python program that takes an executable as input and implements the server side of the protocol. It is a state-machine. It waits for the client's connection and then waits for his commands. The first command the server receives is **RUN**, and, as a response, the server executes `QuinExecuter` with the input executable and returns its result to the client, including the returning value of `client_program()`, the number of steps of the execution and several other low-level information on the execution itself. Then, if the client decides to proceed to the binary search stage (in case of inconsistency between the servers' answers), the server receives the command **GetReducedState** i. As a response, the server now sends 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 and the current heap. This command is received for all the binary search requested configurations.

After the client finishes the binary search stage, it simulates one instruction based on the registers that the servers sent for the n_g step. In case the simulated instruction requires to use some memory, either from the stack or the heap, the client sends the command **GetMemory** i**, address**. Then, the server sends the memory value of the requested address, augmented with a Merkle Hash Tree proof that this value is consistent with the root values it sent before (for step i).

4.3.3 QuinClient

`QuinClient` is a python program that implements the client part of the protocol. It connects to the servers, sends the command **RUN** to all of them and compares their answers. In case of consistency, it can output the correct result of `client_program()` and quit. In case of inconsistency, it executes the binary search and finds the step where the servers disagree (by using the **GetReducedState** i command). Then, it simulates the instruction for step n_g using an open-source X86 emulator called PyEmu [3]. In order to support the use of PyEmu in our prototype, we implemented a new object that encapsulates the verifiable memory accesses using the **GetMemory** i command. When PyEmu needs some memory page, `QuinClient` asks one of the servers for it. The server sends that page along with Merkle Hash Tree proof for this page. Then, `QuinClient` verifies that proof and feeds PyEmu with the received page. Last, `QuinClient` uses the emulation result (the written memory data and the values of the registers) to compute the correct reduced configuration for step n_b, compares it to the servers' answers and declares the correct output. We remark that PyEmu does not support the full X86 instruction set, but it does support the instructions most compilers generate. (Also, it is easy to add other instructions if needed.) Quin inherits this limitation.

4.4 Evaluation

We conducted several experiments with our Quin prototype in order to test the practicality of our protocol. We separated the evaluation of the protocol and the evaluation of our main tool, `QuinExecuter`, in order to better understand where the bottle-necks are. We believe there are many possible directions for improving `QuinExecuter` and therefore we want to get the exact weight of its performance compared to the protocol's overall performance.

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. For our experiments we used the following setup: 1) Laptop installed with Windows Vista, Intel 2.2 GHz CPU, 1 GB of RAM. 2) EC2 virtual machines installed with Windows 2008 32bit, 5 EC2 compute units, 1.7 GB RAM and 160 GB storage. All are located in Amazon's Virginia region. 3) Rackspace virtual machines installed with Windows 2008 32bit, 2 GB RAM, 80 GB storage and default CPU. All are located in Rackspace's Chicago datacenter. The average round-trip time between the laptop and the clouds was 380ms (for packets of size 10K bytes).

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.

4.4.1 QuinExecuter Performance

In order to isolate the overhead of the `QuinExecuter` itself, we ran the following experiments:

1. Execution of the delegated program, `Det_quin.exe`, which is `Determinant.exe` wrapped with our wrapper function. The overhead of our wrapper function itself is negligible.
2. Execution of PIN with `Det_quin.exe`, without any instrumentation (i.e., `pin -- Det_quin.exe`). Since PIN is a dynamic instrumentation tool that uses dynamic translation (JIT), this gives us an estimation of the overhead of this translation (the PIN VM).
3. Execution of PIN with `Det_quin.exe`, with our PIN-tool (i.e., `pin -t quin.dll -- Det_quin.exe`). This is an execution of `Det_quin.exe` when we only count the number of executed instructions (by adding instrumentation code).
4. Execution of PIN with `Det_quin.exe`, with our PIN-tool, for N steps (i.e., `pin -t  quin.dll -- Det_quin .exe -n N`). This is an execution of `Det_quin.exe` for N steps when we count the number of executed instructions and compare it to N. N is the total number of steps of the execution of `Det_quin.exe`. This is the heaviest instrumentation we use in the protocol since it adds code both for counting the number of executed instructions and for the comparison to N.

We ran the above experiments on an Amazon EC2 virtual machine and on a Rackspace Cloud virtual machine. The left side of Table 1 shows the results for those experiments. Each experiment was executed three times. The numbers in the table represent the average running times in seconds on Amazon EC2 and Rackspace Cloud, respectively, separated by slashes.

Matrix Size	Exp. 1	Exp. 2	Exp. 3	Exp. 4	Quin Total	Overhead Factor
10	2/1	3/3	7/7	15/15	381	190
11	21/19	23/21	69/67	156/143	694	33
12	252/230	256/242	829/803	1899/1720	2243	8

Table 1: Evaluation results of PIN-tool experiments and Quin performance

We can observe (from experiment 2) that PIN itself introduces an overhead of at least 1-2 seconds. This overhead is very influencing for the protocol since Quin executes PIN for each round of the binary search and for many short executions, so we get that for x rounds of the binary search we already have an overhead of at least x seconds, and in most cases this overhead is much larger.

Furthermore, we can see that adding even few instrumentation instructions increases the running time by large factors.

Those performance results are important, since it gives us a good estimation of the protocol performance. The protocol starts with executing `Det_quin.exe` and counting the number of steps, and then, continues to an execution of the heavier instrumentation for overall time that is proportional to the running time of the program once. So, in theory (without considering network latency, servers' overheads, etc) we expect that the protocol performance would be something that is proportional to the sum of the fourth and the fifth columns. (Note that these sums are bigger by factors of $10 - 20$ than the plain execution.)

4.4.2 *Performance of the Protocol*

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 $80\%-85\%$ 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. Note that we focus on the efficiency of the clouds since the client's running time is very small (he just sends short TCP messages to each cloud, and receives short answers that are several hundreds of bytes for each round and around 5Kbyte for the last round).

After evaluating the performance of `QuinServer` and noticing that its execution is very expensive, we decided to modify the underlying protocol as following: If $(n_b - n_g)$ is greater than some threshold (e.g., 50M), instead of asking for configuration $(n_b-n_g)/2+n_g$ the client asks for configuration $(n_b-n_g)/4+n_g$. The rest of the protocol is the same. This change is essentially a tradeoff between the number of rounds (that is increased) and the number of executed instructions on the cloud (that is decreased proportionally to the distance of the last n_g configuration from the last configuration).

We used one virtual machine on each cloud provider (as our two clouds) and the laptop as our client. In the right side of Table 1 we show the average running times of the protocol for the program `Determinant.exe`. The numbers represent the total Quin running times as recorded by the client (where only few seconds, combined, are from the client work or from communication latency). We remark that there are major performance differences, both between Amazon EC2 and Rackspace Cloud and between different times of the day. The overhead factors are over the running times of `Det_quin.exe` from experiment 1.

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. E.g., the PIN VM overhead that adds at least 1-2 seconds for each round and the PIN-tool large overheads we saw in the previous section. We believe that a product-level implementation those parts can get much smaller overheads, presumably a factor of 10-20 times slower on average for all computations.

Recall that the protocol from Section 3 has no overhead for the servers in case both of them are honest. Therefore, since most of the time both servers are honest, a more efficient solution would be that the client starts by simply asking both servers for their answers, and only in case of inconsistent answers he proceeds to the full execution of Quin. This gives a solution with no overhead in case both servers are honest.

4.5 Restrictions and Further Improvements

Adding function stubs. Since the program has to be fully deterministic, it can not call any non-deterministic external libraries or OS APIs. This restriction can be overcome (to some extent) by adding function stubs that hide the OS non-deterministic factors. E.g., `fopen()` returns a pointer `FILE*` that points to some non deterministic address. One could implement the functions `ex_fopen()` and `ex_fread()` that instead of working with `FILE*`, they work with some file identifiers that are generated deterministically. Those functions use an internal table to translate that identifier to a matching `FILE*` pointer and call the required `fopen()` or `fread()`. Last, `QuinExecuter` should treat calls to `ex_fopen()` or `ex_fread()` as single instructions. Note that this workaround can work only for functions that are non-deterministic because of the OS implementation, and it can not work with functions that are non-deterministic by definition (e.g., functions that return current time or a packet from the network).

Different Operating Systems. Currently Quin runs only on Windows because we make use of some of the Windows Loader low-level properties. Since PIN supports Windows and Linux, under minor changes to our PIN-tool, `QuinExecuter` can be used also in Linux. As the delegated program is a plain stand-alone C program that can be executed on any OS, and since `QuinClient` and `QuinServer` are OS independent, an interesting improvement would be to generalize `QuinExecuter` to be OS independent.

Support any X86 executable. Our current prototype requires building the delegated executable with our supplied makefile. We stress that it is a design decision since our future goal is to replace the use of our makefile (that adds the wrapper function) with another binary instrumentation, in a way that our PIN-tool could work directly with any stand-alone executable (e.g. commodity software).

Static Instrumentation instead of Dynamic Instrumentation. The functionalities we need from PIN can be achieved by static instrumentation, which is much more efficient for cases of *instrument once, run many times*. We saw in our evaluation that `QuinExecuter` adds the largest overhead of the implementation, therefore, any performance improvement of it will dramatically improve the performance of the overall protocol.

There are several static instrumentation frameworks for Linux environments (e.g., DynInst [2], PEBIL [16]), but we did not find a convenient framework that works also for Windows executables.

Using several computers in each cloud. Since the main bottleneck of our implementation is the executions of `QuinExecuter`, we can use the following trick to reduce the overheads. Instead of running `QuinExecuter` separately for each query, the cloud can use several computers (or other CPU cores) and execute different executions of `QuinExecuter` in parallel. E.g., when queried for configuration i, the cloud executes `QuinExecuter` for i steps on one machine, but also `QuinExecuter` for $i+(n_b-i)/2$ steps on a second machine. Then, if the next query is for step $i+(n_b-i)/2$, the cloud could answer with the result sooner than in the sequential protocol.

Also, using parallel executions of `QuinExecuter`, we can efficiently reduce the number of rounds to $\log_{t+1} T + 3$ as described in Section 3.4.

Implementation for Interpreted Languages. As discussed before, interpreted languages such as Java and Python have complex interpreters that usually maintain an internal state that depends on many factors other then the program itself (e.g., garbage collector's tables). However, those interpreters already do many operations that we can use (e.g., most interpreters do just-in-time compilation, or have stubs for system calls). Therefore, if we could modify them to be more deterministic, we might gain a very efficient solution. During our work we have investigated several interpreters and we believe that the above can be done.

Another environment that is potentially relevant for our protocol is the *Platform as a Service* (e.g., Google App Engine), which usually provides kind of a stand-alone computation delegation service for one process and a restricted set of "system calls". Implementation of our protocol for interpreted languages may be very appealing for such services.

5. REFERENCES

[1] BOINC, Validation and replication. `http://boinc.berkeley.edu/trac/wiki/ValidationSummary`.

[2] DynInst, an application program interface for runtime code generation. `http://www.dyninst.org`.

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

[4] Quin's source code. `http://www.cs.tau.ac.il/~benriva/quin`.

[5] B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: efficient verification via secure computation. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, pages 152–163. Springer-Verlag, 2010.

[6] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. *J. ACM*, 45:70–122, January 1998.

[7] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 21–32. ACM, 1991.

[8] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: how to remove intractability assumptions. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 113–131. ACM, 1988.

[9] K. M. Chung, Y. Kalai, and S. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the 30th annual conference on Advances in cryptology*, pages 483–501. Springer-Verlag, 2010.

[10] U. Feige and J. Kilian. Making games short (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 506–516. ACM, 1997.

[11] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th annual conference on Advances in cryptology*, pages 465–482. Springer-Verlag, 2010.

[12] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 113–122. ACM, 2008.

[13] P. Golle and I. Mironov. Uncheatable distributed computations. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, pages 425–440. Springer, 2001.

[14] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 175–188. ACM, 2007.

[15] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732. ACM, 1992.

[16] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *ISPASS*, pages 175–183, 2010.

[17] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM, 2005.

[18] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

[19] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378. Springer-Verlag, 1988.

[20] S. Micali. Computationally sound proofs. *SIAM J. Comput.*, 30:1253–1298, October 2000.

[21] F. Monrose, P. Wycko, and A. D. Rubin. Distributed execution with remote audit. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, pages 103–113. The Internet Society, 1999.

[22] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 154–168. IEEE Computer Society, 2005.