How To Securely Outsource Cryptographic Computations

Susan Hohenberger[1,⋆] and Anna Lysyanskaya[2,⋆⋆]

[1] CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139 USA,
srhohen@mit.edu

[2] Computer Science Department, Brown University, Providence, RI 02912 USA,
anna@cs.brown.edu

Abstract. We address the problem of using untrusted (potentially malicious) cryptographic helpers. We provide a formal security definition for *securely outsourcing* computations from a computationally limited device to an untrusted helper. In our model, the adversarial environment writes the software for the helper, but then does not have direct communication with it once the device starts relying on it. In addition to security, we also provide a framework for quantifying the *efficiency* and *checkability* of an outsourcing implementation. We present two practical outsource-secure schemes. Specifically, we show how to securely outsource modular exponentiation, which presents the computational bottleneck in most public-key cryptography on computationally limited devices. Without outsourcing, a device would need $O(n)$ modular multiplications to carry out modular exponentiation for n-bit exponents. The load reduces to $O(\log^2 n)$ for any exponentiation-based scheme where the honest device may use two untrusted exponentiation programs; we highlight the Cramer-Shoup cryptosystem [13] and Schnorr signatures [28] as examples. With a relaxed notion of security, we achieve the same load reduction for a new CCA2-secure encryption scheme using only one untrusted Cramer-Shoup encryption program.

1 Introduction

Modern computation has become pervasive: pretty much any device these days, from pacemakers to employee ID badges, is expected to be networked with other components of its environment. This includes devices, such as RFID tags, that are not designed to carry out expensive computations. In fact, RFID tags do not even have a power source. This becomes a serious concern when we want to guarantee that these devices are integrated into the network securely: if a device is computationally incapable of carrying out cryptographic algorithms, how can we give it secure and authenticated communication channels?

In this paper, we study the question of how a computationally limited device may *outsource* its computation to another, potentially malicious, but much more

⋆ Supported by an NDSEG Fellowship.

⋆⋆ Supported by NSF Career grant CNS-0347661.

computationally powerful device. In addition to powering up from an external power source, an RFID tag would have some external helper entity do the bulk of the computation that the RFID tag needs done in order to securely and authentically communicate with the outside world. The non-triviality here is that, although this external helper will be carrying out most of the computation, it can, potentially, be operated by a malicious adversary. Thus, we need to ensure that it does not learn anything about what it is actually computing; and we also need to, when possible, detect any failures.

There are two adversarial behaviors that the helper software might engage in: *intelligent* and *unintelligent* failures. Intelligent failures occur any time that the helper chooses to deviate from its advertised functionality based on knowledge it gained of the inputs to the computation it is aiding. For example, the helper might refuse to *securely* encrypt any message once it sees the public key of a competing software vendor; it might pass any signature with its manufacturer's public key without checking it; it might even choose to broadcast the honest device's secret key to the world. The first goal of any outsourcing algorithm should be to hide as much information as possible about the actual computation from the helper, thus removing its ability to bias outputs or expose secrets. Obviously, software may also unintelligently fail. For example, the helper might contain a malicious bug that causes it to fail on every 1,000th invocation regardless of who is using it. Thus, we face a real challenge: get helper software to do *most* of the computations for an honest device, without telling it anything about what it is actually doing, and then check its output!

In this paper, we give the definition of security for outsourced computation, including notions of efficiency and checkability. We also provide two practical outsource-secure schemes.

In Section 3, we show how to securely outsource variable-exponent, variable-base modular exponentiation. Modular exponentiation has been considered prohibitively expensive for embedded devices. Since it is required by virtually any public-key algorithm, it was believed that public-key cryptography for devices such as RFID tags is impossible to achieve. Our results show that outsourced computation makes it possible for such devices to carry out public-key cryptography. Without outsourcing, a device would need $O(n)$ modular multiplications to carry out a modular exponentiation for an n-bit exponent. Using two untrusted programs that purportedly compute exponentiations (and with the restriction that at most one of them will deviate from its advertised functionality on a non-negligible fraction of inputs), we show that an honest device can get away with doing only $O(\log^2 n)$ modular multiplications itself – while able to catch an error with probability $\frac{1}{2}$. This result leads to a dramatic reduction in the burden placed on the device to support Cramer-Shoup encryption [13] and Schnorr signatures [28] with error rates of $\frac{1}{8}$ and $\frac{1}{4}$ respectively. (Consider that after a small number of uses, malfunctioning software is likely to be caught.)

In Section 4, we show how to securely outsource a CCA2-secure variant of Cramer-Shoup encryption, using only one Cramer-Shoup encryption program as an untrusted helper. Since this is a randomized functionality, its output cannot generally be checked for correctness. However, suppose we can assume that the untrusted helper malfunctions on only a negligible fraction of adversarially chosen inputs; for example, suppose it is encryption software that works properly except when asked to encrypt a message under a certain competitor's public key. Normally, software that fails on only a negligible fraction of *randomly-chosen* inputs can be tolerated, but in the context of secure outsourcing we cannot tolerate any *intelligent* failures (i.e., failures based on the actual public key and message that the user wishes to encrypt). That is, secure outsourcing requires that the final solution, comprised of trusted and untrusted components, works with high probability for all inputs. Consider that Alice may have unwittingly purchased helper software for the sole purpose of encrypting messages under one of the few public keys for which the software is programmed to fail. Thus, in this scenario, we provide a solution for Alice to securely encrypt *any* message under *any* public key with high probability (where the probability is no longer taken over her choice of message and key). One can easily imagine how to hide the message and/or public key for RSA or El Gamal based encryption schemes; however, our second result is non-trivial because we show how to do this for the *non-malleable* Cramer-Shoup encryption scheme, while achieving the same asymptotic speed-up as before.

Related Work. Chaum and Pedersen [11] previously introduced "wallets with observers" where a third party, such as a bank, is allowed to install a piece of hardware on a user's computer. Each transaction between the bank and the user is designed to use this hardware, which the bank trusts, but the user may not. This can be viewed as a special case of our model.

This work shares some similarities with the TPM (Trusted Platform Module) [29], which is currently receiving attention from many computer manufacturers. Like the TPM, our model separates software into two categories: trusted and untrusted. Our common goal is to minimize the necessary trusted resources. Our model differs from TPM in that we have the trusted component controlling all the input/output for the system, whereas TPM allows some inputs/outputs to travel directly between the environment and untrusted components.

In the 1980s, Ben-Or et al. used multiple provers as a way of removing intractability assumptions in interactive proofs [4], which led to a series of results on hiding the input, and yet obtaining the desired output, from an honest-but-curious oracle [1–3]. Research in program checking merged into this area when Blum, Luby, and Rubinfeld [5, 7, 6] considered checking *adaptive, malicious* programs (i.e., oracles capable of intelligently failing).

The need for a formal security definition of outsourcing is apparent from previous research on using untrusted servers for RSA computations, such as the work of Matsumoto et al. [22] which was subsequently broken by Nguyen and Shparlinski [24]. We incorporate many previous notions including: the idea of an untrusted helper [19], confining untrusted applications and yet allowing a sanitized space for trusted applications to operate [30], and oracle-based checking of untrusted software [23]. Our techniques in Section 4 also offer novel approaches to the area of message and key *blinding* protocols [10, 18, 31].

Secure outsourcing of exponentiations is a popular topic [27, 28, 17, 8, 25, 22, 1–3, 12], but past approaches either focus on fixed-base (or fixed-exponent) exponentiation or meet a weaker notion of security.

2 Definition of Security

Suppose that we have a cryptographic algorithm *Alg*. Our goal is to split *Alg* up into two components: (1) a trusted component T that *sees* the input to *Alg* but is not very computationally intensive; (2) luckily T can make oracle queries to the second component, U, which is an untrusted component (or possibly components) that can carry out computation-intensive tasks.

Informally, we say that T *securely outsources* some work to U, and that (T, U) thereby form an *outsource-secure* implementation of a cryptographic algorithm *Alg* if (1) together, they implement *Alg*, i.e., $Alg = T^U$ and (2) suppose that, instead of U, T is given oracle access to a malicious U' that records all of its computation over time and, every time it is invoked, tries to act maliciously – e.g., not work on some adversarially selected inputs; we do not want such a malicious U', despite carrying out most of the computation for $T^{U'}(x)$, to learn anything *interesting* about the input x. For example, we do not want a malicious U' to trick T into rejecting a valid signature because U' sees the verification key of a competing software vendor or a message it does not like.

To define outsource-security more formally, we first ask ourselves how much security can be guaranteed. The least that U' can learn is that T actually received some input. In some cases, for a cryptographic algorithm $Alg = T^U$ that takes as input a secret key SK, and an additional input x, we may limit ourselves to hiding SK but not worry about hiding x. For example, we might be willing to give a ciphertext to the untrusted component U', but not our secret key. At other times, we may want to hide everything meaningful from U'. Thus, the inputs to *Alg* can be separated into two logical groups: (1) inputs that should remain hidden from the untrusted software U' at all times (for example, keys and messages), and (2) inputs that U' is entitled to know if it is to be of any help in running *Alg* (for example, if *Alg* is a time-stamping scheme, then U' may need to know the current time). Let us denote these two types of input as *protected* and *unprotected*.

Similarly, *Alg* has protected and unprotected outputs: those that U' is entitled to find out, and those that it is not. For example, if $Alg = T^U$ is an encryption program it may ask U' to help it compute a part of the ciphertext, but then wish to conceal other parts of the ciphertext from U'.

However, U' is not the only malicious party interacting with *Alg*. We model the adversary A as consisting of two parts: (1) the adversarial environment E that submits adversarially chosen inputs to *Alg*; (2) the adversarial software U' operating in place of oracle U. One of the fundamental assumptions of this model is that E and U' may first develop a joint strategy, but once they begin interacting with an honest party T, they no longer have a direct communication channel. Now, E may get to see some of the protected inputs to *Alg* that U' does not. For example, E gets to see *all* of its own adversarial inputs to *Alg*, although T might hide some of these from U'. Consider that if U' was able to see some values chosen by E, then E and U' can agree on a joint strategy causing U' to stop working upon receiving some predefined message from E. Thus, there are going to be some inputs that are known to E, but hidden from U', so we ought to formalize how different their views need to be.

We have three logical divisions of inputs to *Alg*: (1) secret – information only available to T (e.g., a secret key or a plaintext); (2) protected – information only available to T and E (e.g., a public key or a ciphertext); (3) unprotected – information available to T, E, and U' (e.g., the current time). These divisions are further categorized based on whether the inputs were generated *honestly* or *adversarially*, with the exception that there is no *adversarial, secret* input – since by definition it would need to be both generated by and kept secret from E. Similarly, *Alg* has secret, protected, and unprotected outputs. Thus, let us write that *Alg* takes five inputs and produces three outputs. This is simplified notation since these inputs may be related to each other in some way. For example, the secret key is related to the public key.

As an example of this notation, consider a signing algorithm *sign* such that we want to hide from the malicious software U' the secret key *SK* and the message m that is being signed, but not the time t at which the message is signed. The key pair was generated using a correct key generation algorithm and the time was honestly generated, while the message may have been chosen adversarially. Also, we do not want the malicious U' to find out anything about the signature that is output by the algorithm. Then we write $sign(SK, \varepsilon, t, m, \varepsilon) \rightarrow (\varepsilon, \sigma, \varepsilon)$ to denote that the signature σ is the protected output, there are no secret or unprotected outputs, *SK* is the honest, secret input, t is the honest, unprotected input, m is the adversarial, protected input, and there are no other inputs. This situation grows more complex when we consider *Alg* operating in a compositional setting where the protected outputs of the last invocation might become the adversarial, unprotected inputs of the next; we will further discuss this subtlety in Remark 2.

Let us capture an algorithm with this input/output behavior in a formal definition:

Definition 1 (Algorithm with outsource-IO). *An algorithm Alg obeys the outsource input/output specification if it takes five inputs, and produces three outputs. The first three inputs are generated by an honest party, and are classified by how much the adversary $A = (E, U')$ knows about them. The first input is called the* honest, secret *input, which is unknown to both E and U; the second is called the* honest, protected *input, which may be known by E, but is protected from U; and the third is called the* honest, unprotected *input, which may be known by both E and U. In addition, there are two adversarially-chosen inputs generated by the environment E: the* adversarial, protected *input, which is known to E, but protected from U; and the the* adversarial, unprotected *input, which may be known by E and U. Similarly, the first output called* secret *is unknown to both E and U; the second is* protected*, which may be known to E, but not U; and the third is* unprotected*, which may be known by both parts of A.*

At this point, this is just input/output notation, we have not said anything about actual security properties. We now discuss the definition of security.

The two adversaries E, U' can only communicate with each other by passing messages through T, the honest party. In the real world, a malicious manufacturer E might program its software U' to behave in an adversarial fashion; but once U' is installed behind T's firewall, manufacturer E should no longer be able to directly send instructions to it. Rather, E may try to establish an *indirect* communication channel with U' via the unprotected inputs and outputs of Alg. For example, if E knows that the first element in a signature tuple is unprotected (meaning, T always passes the first part of a signature tuple, unchanged, to U'), it might encode a message in that element instructing U' to "just tell T the signature is valid" – even though it may not be. Alternatively, an indirect communication channel might be realized by U' smuggling secrets about the computation it helped T with, through the unprotected outputs, back to E. For example, if, in the course of helping T with decryption, U' learned the secret key, it might append that key to the next unprotected output it creates for T. Obviously, T must use U' with great care, or he will be completely duped.

Our definition of outsource-security requires that anything secret or protected that a malicious U' can learn about the inputs to T^U from being T's oracle instead of U, it can also learn without that. Namely, there exists a simulator S_2 that, when told that $T^U(x)$ was invoked, simulates the view of U' without access to the secret or protected inputs of x. This property ensures that U' cannot intelligently choose to fail.

Similarly, our definition of outsource-security must also prevent the malicious environment E from gaining any knowledge of the secret inputs and outputs of T^U, even when T is using malicious software U' written by E. Again, there exists a simulator S_1 that, when told that $T^{U'}(x)$ was invoked, simulates the view of E without access to the secret inputs of x.

Definition 2 (Outsource-security). *Let $Alg(\cdot,\cdot,\cdot,\cdot,\cdot)$ be an algorithm with outsource-IO. A pair of algorithms (T, U) is said to be an outsource-secure implementation of an algorithm Alg if:*
Correctness *T^U is a correct implementation of Alg.*
Security *For all probabilistic polynomial-time adversaries $A = (E, U')$, there exist probabilistic expected polynomial-time simulators (S_1, S_2) such that the following pairs of random variables are computationally indistinguishable. Let us say that the honestly-generated inputs are chosen by a process I.*
Pair One*: $EVIEW_{real} \sim EVIEW_{ideal}$ (The external adversary, E, learns nothing.):*

– *The view that the adversarial environment E obtains by participating in the following REAL process:*

$$EVIEW^i_{real} = \{(istate^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) \leftarrow I(1^k, istate^{i-1});$$
$$(estate^i, j^i, x^i_{ap}, x^i_{au}, stop^i) \leftarrow E(1^k, EVIEW^{i-1}_{real}, x^i_{hp}, x^i_{hu});$$
$$(tstate^i, ustate^i, y^i_s, y^i_p, y^i_u) \leftarrow T^{U'(ustate^{i-1})}(tstate^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au}) :$$

$$(estate^i, y_p^i, y_u^i)\}$$

$EVIEW_{real} = EVIEW^i_{real}$ if $stop^i = TRUE$.
The real process proceeds in rounds. In round i, the honest (secret, protected, and unprotected) inputs $(x^i_{hs}, x^i_{hp}, x^i_{hu})$ are picked using an honest, stateful process I to which the environment does not have access. Then the environment, based on its view from the last round, chooses (0) the value of its $estate_i$ variable as a way of remembering what it did next time it is invoked; (1) which previously generated honest inputs $(x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu})$ to give to $T^{U'}$ (note that the environment can specify the index *j^i of these inputs, but not their* values*); (2) the adversarial, protected input x^i_{ap}; (3) the adversarial, unprotected input x^i_{au}; (4) the Boolean variable $stop^i$ that determines whether round i is the last round in this process. Next, the algorithm $T^{U'}$ is run on the inputs $(tstate^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au})$, where $tstate^{i-1}$ is T's previously saved state, and produces a new state $tstate^i$ for T, as well as the secret y^i_s, protected y^i_p and unprotected y^i_u outputs. The oracle U' is given its previously saved state, $ustate^{i-1}$, as input, and the current state of U' is saved in the variable $ustate^i$. The view of the real process in round i consists of $estate^i$, and the values y^i_p and y^i_u. The overall view of the environment in the real process is just its view in the last round (i.e., i for which $stop^i = TRUE$).*

– *The IDEAL process:*

$$\begin{aligned}
EVIEW^i_{ideal} = \{&(istate^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) \leftarrow I(1^k, istate^{i-1});\\
&(estate^i, j^i, x^i_{ap}, x^i_{au}, stop^i) \leftarrow E(1^k, EVIEW^{i-1}_{ideal}, x^i_{hp}, x^i_{hu});\\
&(astate^i, y^i_s, y^i_p, y^i_u) \leftarrow Alg(astate^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au});\\
&(sstate^i, ustate^i, Y^i_p, Y^i_u, replace^i) \leftarrow S_1^{U'(ustate^{i-1})}(sstate^{i-1}, \ldots\\
&\qquad \ldots x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au}, y^i_p, y^i_u);\\
&(z^i_p, z^i_u) = replace^i(Y^i_p, Y^i_u) + (1 - replace^i)(y^i_p, y^i_u) :\\
&(estate^i, z^i_p, z^i_u)\}
\end{aligned}$$

$EVIEW_{ideal} = EVIEW^i_{ideal}$ if $stop^i = TRUE$.
The ideal process also proceeds in rounds. In the ideal process, we have a stateful simulator S_1 who, shielded from the secret input x^i_{hs}, but given the non-secret outputs that Alg produces when run all *the inputs for round i, decides to either output the values (y^i_p, y^i_u) generated by Alg, or replace them with some other values (Y^i_p, Y^i_u). (Notationally, this is captured by having the indicator variable $replace^i$ be a bit that determines whether y^i_p will be replaced with Y^i_p.) In doing so, it is allowed to query the oracle U'; moreover, U' saves its state as in the real experiment.*

Pair Two*: $UVIEW_{real} \sim UVIEW_{ideal}$ (The untrusted software, U', learns nothing.):*

– *The view that the untrusted software U' obtains by participating in the REAL process described in Pair One.* $UVIEW_{real} = ustate^i$ *if* $stop^i = TRUE$.
– *The IDEAL process:*

$$\begin{aligned} UVIEW^i_{ideal} = \{ (istate^i, x^i_{hs}, x^i_{hp}, x^i_{hu}) &\leftarrow I(1^k, istate^{i-1}); \\ (estate^i, j^i, x^i_{ap}, x^i_{au}, stop^i) &\leftarrow E(1^k, estate^{i-1}, x^i_{hp}, x^i_{hu}, y^{i-1}_p, y^{i-1}_u); \\ (astate^i, y^i_s, y^i_p, y^i_u) &\leftarrow Alg(astate^{i-1}, x^{j^i}_{hs}, x^{j^i}_{hp}, x^{j^i}_{hu}, x^i_{ap}, x^i_{au}); \\ (sstate^i, ustate^i) &\leftarrow S_2^{U'(ustate^{i-1})}(sstate^{i-1}, x^{j^i}_{hu}, x^i_{au}) : \\ &\qquad (ustate^i) \} \end{aligned}$$

$UVIEW_{ideal} = UVIEW^i_{ideal}$ *if* $stop^i = TRUE$.
In the ideal process, we have a stateful simulator S_2 who, equipped with only the unprotected inputs (x^i_{hu}, x^i_{au}), queries U'. As before, U' may maintain state.

There are several interesting observations to make about this security definition.

Remark 1. The states of all algorithms, i.e., I, E, U', T, S_1, S_2, in the security experiments above are initialized to $\emptyset$. Any joint strategy that E and U' agree on prior to acting in the experiments must be embedded in their respective codes. Notice the intentional asymmetry in the access to the untrusted software U' given to environment E and the trusted component T. The environment E is allowed non-black-box access to the software U', since E may have written code for U'; whereas U' will appear as a black-box to T, since one cannot assume that a malicious software manufacturer will (accurately) publish its code. Or, consider the example of an RFID tag outsourcing its computation to a more powerful helper device in its environment. In this case we cannot expect that, in the event that it is controlled by an adversary, such a helper will run software that is available for the purposes of the proof of security.

Remark 2. For any outsource-secure implementation, the adversarial, unprotected input x_{au} must be empty. If x_{au} contains even a single bit, then a covert channel is created from E to U', in which k bits of information can be transfered after k rounds. In such a case, E and U' could jointly agree on a secret value beforehand, and then E could slowly smuggle in that k-bit secret to U'. Thus, $UVIEW_{real}$ would be distinguishable from $UVIEW_{ideal}$, since E may detect that it is interacting with Alg instead of $T^{U'}$ (since Alg's outputs (y^i_p, y^i_u) are always correct), and communicate this fact to U' through the covert channel. A non-empty x_{au} poses a real security threat, since it would theoretically allow a software manufacturer to covertly reprogram its software *after* it was installed behind T's firewall and *without* his consent.

Remark 3. No security guarantee is implied in the event that the environment E and the software U' are able to communicate without passing messages through T. For example, in the event that E captures all of T's network traffic and then steals T's hard-drive (containing the memory of U') – all bets are off!

RFID tags and other low-resource devices require that a large portion of their cryptographic computations be outsourced to better equipped computers. When a cryptographic algorithm *Alg* is divided into a pair of algorithms (T, U), in addition to its security, we also want to know how much work T saves by using U. We want to compare the work that T must do to safely use U to the work required for the fastest known implementation of the functionality T^U.

Definition 3 (α-efficient, secure outsourcing). *A pair of algorithms (T, U) are an α-*efficient *implementation of an algorithm Alg if (1) they are an outsource-secure implementation of Alg, and (2) $\forall$ inputs x, the running time of T is $\leq$ an α-multiplicative factor of the running time of $Alg(x)$.*

For example, say U relieves T of at least half its computational work; we would call such an implementation $\frac{1}{2}$-efficient. The notion above considers only T's computational load compared to that of *Alg*. One might also choose to formally consider U's computational burden or the amount of precomputation that T can do in his idle cycles versus his on-demand load. We will not be formally considering these factors.

The above definition of outsource-security does not prevent U' from deviating from its advertised functionality, rather it prevents U' from intelligently choosing her moments for failure based on any secret or protected inputs to *Alg* (e.g., a public key or the contents of a message). Since this does not rule out unintelligent failures, it is desirable that T have some mechanism for discovering that his software is unsound. Thus, we introduce another characteristic of an outsourcing implementation.

Definition 4 (β-checkable, secure outsourcing). *A pair of algorithms (T, U) are a β-*checkable *implementation of an algorithm Alg if (1) they are an outsource-secure implementation of Alg, and (2) $\forall$ inputs x, if U' deviates from its advertised functionality during the execution of $T^{U'}(x)$, T will detect the error with probability $\geq \beta$.*

Recall that the reason T purchased U in the first place was to *get out of doing work*, so any testing procedure should be *far more* efficient than computing the function itself; i.e., the overall scheme, including the testing procedure, should remain α-efficient. We combine these characteristics into one final notion.

Definition 5 ((α, β)-outsource-security). *A pair of algorithms (T, U) are an (α, β)-*outsource-secure *implementation of an algorithm Alg if they are both α-efficient and β-checkable.*

3 Outsource-Secure Exponentiation Using Two Untrusted Programs

Since computing exponentiations modulo a prime is, by far, the most expensive operation in many discrete-log based cryptographic protocols, much research has been done on how to reduce this work-load. We present a method to securely outsource most of the work needed to compute a variable-exponent, variable-base exponentiation modulo a prime, by combining two previous approaches to this problem: (1) using preprocessing tricks to speed-up *offline* exponentiations [27, 28, 17, 8, 25] and (2) *untrusted* server-aided computation [22, 1–3].

The preprocessing techniques (introduced by Schnorr [27, 28], broken by de Rooij [14, 15, 17], and subsequently fixed by others [16, 9, 21, 8, 25]) seek to optimize the production of random $(k, g^k \bmod p)$ pairs used in signature generation (e.g., El Gamal, Schnorr, DSA) and encryption (e.g., El Gamal, Cramer-Shoup). By *offline*, we mean the randomization factors that are independent of a key or message; that is, exponentiations for a fixed base g, where the user requires nothing more of the exponent k than that it appear random. We leverage these algorithms to speed-up *online* exponentiations as well; that is, given random values $x \in \mathbb{Z}_{ord(G)}$ and $h \in \mathbb{Z}_p^*$, compute $h^x \bmod p$. Generally speaking, given *any* oracle that provides T with random pairs $(x, g^x \bmod p)$ (we discuss the exact implementation of this oracle in Section 3.2), we give a technique for efficiently computing *any* exponentiation modulo p. To do this, we use *untrusted* server-aided (or program-aided) computation.

Blum, Luby, and Rubinfeld gave a general technique for computing and checking the result of a modular exponentiation using four untrusted exponentiation programs – that cannot communicate with each other after deciding on an initial strategy [6]. Their algorithm *leaks* only the size of the inputs (i.e., $|x|, |g|$ for known p) to the programs and runs in time $O(n \log^2 n)$ for an n-bit exponent (this includes the running time of each program). The output of the Blum et al. algorithm is always guaranteed to be correct.

We provide a technique for computing and checking the result of a modular exponentiation using two untrusted exponentiation boxes $U' = (U'_1, U'_2)$ – that again, cannot communicate with each other after deciding on an initial strategy. In this strategy, at most one of them can deviate from its advertised functionality on a non-negligible fraction of the inputs. Our algorithm reveals no more information than the size of the input and the running time is reduced to $O(\log^2 n)$ multiplications for an n-bit exponent.[3] More importantly, we focus on minimizing the computations done by T to compute an exponentiation, which is $O(\log^2 n)$. This is an asymptotic improvement over the $1.5n$ multiplications needed to compute an exponentiation using square-and-multiply. We gain some of this efficiency by only requiring that an error in the output be detected with probability $\frac{1}{2}$. The rationale is that software malfunctioning on a non-negligible amount of random inputs will not be on the market long.

Our $(O(\frac{\log^2 n}{n}), \frac{1}{2})$-outsource-secure exponentiation implementation, combined with previously known preprocessing tricks, yields a technique for using two untrusted programs U_1, U_2 to securely do most of the resource-intensive work in discrete log based protocols. By way of example, we highlight an asymptotic speed-up in the running time of an honest user T (from $O(n)$ to $O(\log^2 n)$) for

[3] Disclaimer: these running times assume certain security properties about the EBPV generator [25] which we discuss in detail in Section 3.2.

the Cramer-Shoup cryptosystem [13] and Schnorr signature verification [27, 28] when using U_1, U_2. Let's lay out the assumptions for using the two untrusted programs more carefully.

3.1 The Two Untrusted Program Model

In the *two untrusted program model*, E writes the code for two (potentially different) programs U_1', U_2'. E then gives this software to T, advertising a functionality that U_1' and U_2' may or may not accurately compute, and T installs this software in a manner such that all subsequent communication between any two of E, U_1' and U_2' must pass through T. The new adversary attacking T (i.e., trying to read T's messages or forge T's signatures) is now $A = (E, U_1', U_2')$.

The *one-malicious* version of this model assumes that at most one the programs U_1', U_2' deviates from its advertised functionality on a non-negligible fraction of the inputs; but we do not know which one and security means that there is a simulator for both. This is the equivalent of buying the "same" advertised software from two different vendors and achieving security as long as one of them is honest without knowing which one.

The concept of an honest party gaining information from two (or more) possibly dishonest, but physically separated parties was first used by Ben-Or, Goldwasser, Kilian, and Wigderson [4] as a method for obtaining interactive proofs without intractability assumptions. Blum et. al. expanded this notion to allow an honest party to check the output of a function rather than just the validity of a proof [7, 6]. Our work, within this model, demonstrates, rather surprisingly, that an honest party T can leverage adversarial software to do the vast majority of its cryptographic computations!

Our $(O(\frac{\log^2 n}{n}), \frac{1}{2})$-outsource-secure implementation of *Exp*, exponentiation modulo a prime function, appears in Figure 1 and Section 3.3. Figure 1 also demonstrates how to achieve an asymptotic speed-up in T's running time, using *Exp* and other known preprocessing techniques [25], for the Cramer-Shoup cryptosystem [13] and Schnorr signature verification [27, 28] (this speed-up was already known for Schnorr signature generation [25]).

3.2 *Rand*1, *Rand*2: Algorithms for Computing $(b, g^b \bmod p)$ Pairs

The subroutine *Rand*1 in Figure 1 is initialized by a prime p, a base $g_3 \in \mathbb{Z}_p^*$, and possibly some other values, and then, on each invocation must produce a random, independent pair of the form $(b, g_3^b \bmod p)$, where $b \in \mathbb{Z}_q$. The subroutine *Rand*2 is the natural extension of *Rand*1 initialized by two bases g_1, g_2 and producing triplets $(b, g_1^b \bmod p, g_2^b \bmod p)$. Given a, perhaps expensive, initialization procedure, we want to see how expeditiously these subroutines can be executed by T.

One naive approach is for a *trusted* server to compute a table of random, independent pairs and triplets in advance and load it into T's memory. Then on each invocation of *Rand*1 or *Rand*2, T simply retrieves the next value in the

Outsource-secure Encryption and Signatures: $Alg = (T, U_1, U_2)$
(in the two untrusted program model)

Global Setup (denoted gp **as honest, unprotected inputs)**

∘ Security parameter: 1^k.

∘ Global Encryption parameters: a group G of prime order q with generators g_1, g_2, a (can be weakly) collision-resistant hash function $H : \{0,1\} \to \mathbb{Z}_q$.

∘ Global Signature parameters: a k-bit prime q, $p = 2q+1$, a generator g_3 for $\mathbb{Z}_p^*$, and a collision-resistant hash function $H : \{0,1\}^* \to \mathbb{Z}_q$.

Advertised Functionality of U_1 **and** U_2

∘ $U_1(b, g) \to g^b$

∘ $U_2(b, g) \to g^b$

Subroutines Executed by T **with access to** U_1, U_2

∘ $Rand1 \to (b, g_3^b)$. T computes alone as in Section 3.2.

∘ $Rand2 \to (b, g_1^b, g_2^b)$. T computes alone as in Section 3.2.

∘ $Exp(a, u) \to u^a$. T uses U_1 and U_2 to compute u^a as in Section 3.3.

Functionality of $Alg = (T, U_1, U_2)$

Outsource-Secure Cramer-Shoup Cryptosystem [13]

Key Generation: Generated by an honest process on input 1^k:

$$PK = (B = g_1^{x_1} g_2^{x_2}, C = g_1^{y_1} g_2^{y_2}, D = g_1^z),\ SK = (x_1, x_2, y_1, y_2, z).$$

Encryption: $Alg.Enc(m, (PK, t), gp, \varepsilon, \varepsilon) \to (\varepsilon, \tau, \varepsilon)$.

On input $PK = (B, C, D)$, $m \in G$, and $t \in \{0,1\}^*$,

1. T computes $Rand2 \to (r, u_1 = g_1^r, u_2 = g_2^r)$.
2. T computes $Exp(r, D) \to D^r, e = D^r m, \kappa = H(u_1, u_2, e, t)$.
3. T computes $Exp(r, B) \to B^r, Exp(r\kappa, C) \to C^{r\kappa}, v = B^r C^{r\kappa}$.
4. T outputs the ciphertext $\tau = (u_1, u_2, e, v, t)$.

Decryption: $Alg.Dec(SK, \varepsilon, gp, \tau, \varepsilon) \to (m, \varepsilon, \varepsilon)$.

(If E generates ciphertext, $Alg.Dec(SK, \varepsilon, gp, \tau, \varepsilon) \to (\varepsilon, m, \varepsilon)$.)

On input $SK = (x_1, x_2, y_1, y_2, z)$ and $\tau = (u_1, u_2, e, v, t)$,

1. T computes $\kappa = H(u_1, u_2, e, t)$.
2. T computes $Exp(x_1 + \kappa y_1, u_1) \to \alpha, Exp(x_2 + \kappa y_2, u_2) \to \beta$.
3. T checks if $\alpha\beta = v$; if not, it outputs "invalid".
4. Otherwise, T computes $Exp(z, u_1) \to \delta$ and outputs $m = e/\delta$.

Outsource-Secure Schnorr Signatures [27, 28]

Key Generation: Generated by an honest process: $SVK = g^a, SSK = a$.

Signature Generation: $Alg.Sign(SSK, m, gp, \varepsilon, \varepsilon) \to (\varepsilon, \sigma, \varepsilon)$.

On input $SSK = a$ and $m \in \{0,1\}^*$,

1. T computes $Rand1 \to (k, r = g_3^k)$.
2. T computes $e = H(r||m)$ and $s = ae + k \bmod q$.
3. T outputs the signature $\sigma = (r, s)$.

Signature Verification: $Alg.Vf(\varepsilon, SVK, gp, (m, \sigma), \varepsilon) \to (\varepsilon, \{0,1\}, \varepsilon)$.

On input $SVK = y$, $m \in \{0,1\}^*$, and $\sigma = (r, s)$,

1. T checks that $1 \le r \le p - 1$, if not, it outputs 0.
2. T computes $e = H(r||m)$, $Exp(s, g_3) \to \alpha$ and $Exp(e, y) \to \beta$.
3. T checks that $\alpha = \beta r$. If so, T outputs 1; otherwise it outputs 0.

Fig. 1. An honest user T, given untrusted exponentiation boxes U_1, U_2, achieves outsource-secure encryption and signatures.

table. (We will see that this table, plus access to the untrusted servers, allows an honest device to compute any exponentiation by doing only 9 multiplications regardless of exponent size!)

For devices that are willing to do a little more work, in exchange for requiring less storage, we apply well-known preprocessing techniques for this exact functionality. Schnorr first proposed an algorithm which, takes as input a small set of truly random (k, g^k) pairs and then, produces a long series of "nearly random" (r, g^r) pairs as a means of speeding-up signature generation in smartcards [27]. However, the output of Schnorr's algorithm is too dependent, and de Rooij found a series of equations that allow the recovery of a signer's secret key [14]. A subsequent fix by Schnorr [28] was also broken by de Rooij [15, 17]. Since then several new preprocessing algorithms were proposed [16, 9, 21, 8, 25]. Among the most promising is the EBPV generator by Nguyen, Shparlinski, and Stern [25], which adds a feedback extension (i.e., reuse of the output pairs) to the BPV generator proposed by Boyko, Peinado, and Venkatesan [8], which works by taking a subset of truly random (k, g^k) pairs and combing them with a random walk on expanders on Cayley graphs to reduce the dependency of the pairs in the output sequence. The EBPV generator, secure against adaptive adversaries, runs in time $O(\log^2 n)$ for an n-bit exponent. (This holds for the addition of a second base in *Rand*2 as well.)

A critical property that we will shortly need from *Rand*1 and *Rand*2 is that their output sequences be computationally indistinguishable from a truly random output sequence. It is conjectured that with sufficient parameters (i.e., number of initial (k, g^k) pairs, etc.) the output distribution of the EBPV generator is statistically-close to the uniform distribution [25]. We make this working assumption throughout our paper. In the event that this assumption is false, our recourse is to use the naive approach above and, thus, further reduce our running time, in exchange for additional memory.

3.3 *Exp*: Outsource-Secure Exponentiation Modulo a Prime

Our main contribution for Section 3 lies in the subroutine *Exp* from Figure 1. In *Exp*, T out-sources its exponentiation computations, while maintaining its privacy, by invoking U_1 and U_2 on a series of (exponent, base) pairs that appear random in the limited view of the software.

The *Exp* Algorithm. Let primes p, q be the global parameters, where $\mathbb{Z}_p^*$ has order q. *Exp* takes as input $a \in \mathbb{Z}_q$ and $u \in \mathbb{Z}_p^*$, and outputs $u^a \bmod p$. As used in Figure 1, *Exp*'s input a may be secret or (honest/adversarial) protected; its input u may be (honest/adversarial) protected; and its output is always secret or protected. *Exp* also receives the (honest, unprotected) global parameters *gp*; there are no adversarial, unprotected inputs. All (secret/protected) inputs are computationally blinded before being sent to U_1 or U_2.

To implement this functionality using (U_1, U_2), T runs *Rand*1 twice to create two *blinding pairs* (α, g^α) and (β, g^β). We denote

$$v = g^\alpha \text{ and } v^b = g^\beta, \text{ where } b = \beta/\alpha.$$

Our goal is to logically break u and a into random looking pieces that can then be computed by U_1 and U_2. Our first logical divisions are

$$u^a = (vw)^a = v^a w^a = v^b v^c w^a, \text{ where } w = u/v \text{ and } c = a - b.$$

As a result of this step, u is hidden, and the desired value u^a is expressed in terms of random v and w. Next, T must hide the exponent a. To that end, it selects two *blinding elements* $d \in \mathbb{Z}_q$ and $f \in G$ at random. Our second logical divisions are

$$v^b v^c w^a = v^b (fh)^c w^{d+e} = v^b f^c h^c w^d w^e, \text{ where } h = v/f \text{ and } e = a - d.$$

Next, T fixes two *test* queries per program by running *Rand*1 to obtain (t_1, g^{t_1}), (t_2, g^{t_2}), (r_1, g^{r_1}) and (r_2, g^{r_2}). T queries U_1 (in random order) as

$$U_1(d, w) \to w^d,\ U_1(c, f) \to f^c,\ U_1(t_1/r_1, g^{r_1}) \to g^{t_1},\ U_1(t_2/r_2, g^{r_2}) \to g^{t_2},$$

and then queries U_2 (in random order) as

$$U_2(e, w) \to w^e,\ U_2(c, h) \to h^c,\ U_2(t_1/r_1, g^{r_1}) \to g^{t_1},\ U_2(t_2/r_2, g^{r_2}) \to g^{t_2}.$$

(Notice that *all* queries to U_1 can take place before any queries to U_2 must be made.) Finally, T checks that the test queries to U_1 and U_2 both produce the correct outputs (i.e., g^{t_1} and g^{t_2}). If not, T outputs "error"; otherwise, it multiplies the real outputs of U_1, U_2 with v^b to compute u^a as

$$v^b f^c h^c w^d w^e = v^{b+c} w^{d+e} = v^a w^a = (vw)^a = u^a.$$

We point out that this exponentiation outsourcing only needs the *Rand*1 functionality; the *Rand*2 functionality discussed in Figure 1 is used for the Cramer-Shoup outsourcing.

Theorem 1. *In the one-malicious model, the above algorithms* $(T, (U_1, U_2))$ *are an outsource-secure implementation of Exp, where the input* (a, u) *may be honest, secret, or honest, protected, or adversarial, protected.*

Proof of Theorem 1 is in the full version of the paper. The correctness property is fairly straight-forward. To show security, both simulators S_1 and S_2 send random (exponent, base) pairs to the untrusted components U_1' and U_2'. One can see that such an S_2 simulates a view that is computationally indistinguishable from the real world view for the untrusted helper. However, it is our construction of S_1 which must simulate a view for the environment that requires the one-malicious model. Consider what might happen in the real world if both U_1' and U_2' deviate from their advertised functionalities. While the event that U_1' misbehaves is independent of the input (a, u), and the same is true for the event that U_2' misbehaves, the event that *both* of them misbehave is *not independent* of the input (a, u).

Lemma 1. *In the one-malicious model, the above algorithms* $(T, (U_1, U_2))$ *are an* $O(\frac{\log^2 n}{n})$*-efficient implementation of Exp.*

Proof. Raising an arbitrary base to an arbitrary power by the square-and-multiply method takes roughly $1.5n$ modular multiplications (MMs) for an n-bit exponent. *Exp* makes six calls to *Rand*1 plus 9 other MMs (additions are negligible by comparison). *Exp* takes $O(\log^2 n)$ MMs using the EBPV generator [25] for *Rand*1 and $O(1)$ MMs when using a table-lookup for *Rand*1. □

Lemma 2. *In the one-malicious model, the above algorithms* $(T, (U_1, U_2))$ *are a* $\frac{1}{2}$*-checkable implementation of Exp.*

Proof. By Theorem 1, U_1 (resp,. U_2) cannot distinguish the two test queries from the two real queries T makes. If U_1 (resp., U_2) fails during any execution of *Exp*, it will be detected with probability $\frac{1}{2}$. □

We combine Theorem 1, Lemmas 1 and 2, and known preprocessing techniques [25] to arrive at the following result. (Schemes differ in β-checkability depending on the number of *Exp* calls they make.)

Theorem 2. *In the one-malicious model, the algorithms* $(T, (U_1, U_2))$ *in Figure 1 are (1) an* $(O(\frac{\log^2 n}{n}), \frac{1}{2})$*-outsource-secure implementation of Exp, (2) an* $(O(\frac{\log^2 n}{n}), \frac{7}{8})$*-outsource-secure implementation of the Cramer-Shoup cryptosystem* [13]*, and (3) an* $(O(\frac{\log^2 n}{n}), \frac{3}{4})$*-outsource-secure implementation of Schnorr Signatures* [27, 28].

4 Outsource-Secure Encryption Using One Untrusted Program

Suppose Alice is given an encryption program that is guaranteed to work correctly on all but a negligible fraction of adversarially-chosen public keys and messages. She wants to trick this software into efficiently helping her encrypt *any* message for *any* intended recipient – even those in the (unknown) set for which her software adversarially fails. This is a non-trivial exercise when one wants to hide both the public key and the message from the software – and even more so, when one wants to achieve CCA2-secure encryption, as we do.

Section 3 covered an $O(\frac{\log^2 n}{n})$-efficient outsource-secure CCA2 encryption scheme using *two* untrusted programs. Here, using only *one* untrusted program, we remain $O(\frac{\log^2 n}{n})$-efficient and CCA2-secure. To efficiently use only one program, one must assume that the software behaves honestly on random inputs with high-probability. After all, it isn't hard to imagine a commercial product that works most of the time, but has a few surprises programmed into it such that on a few inputs it malfunctions. Moreover, some assumption about the correctness of a probabilistic program is necessary since there will be no means of checking its output. (To see this, consider that there is no way for T to know if the "randomness" U' used during the probabilistic encryption was genuinely random or a value known by E.) In Figure 2, present an outsource-secure implementation for CCA2-secure encryption only. We leave open the problem of efficiently outsourcing the decryption of *these* ciphertexts, as well as *any* signature verification algorithm, using only one untrusted program.

The One Untrusted Program Model. This model is analogous to the two untrusted program model in Section 3.1, where only one of U_1, U_2 is available to T and the advertised functionality of U is *tagged* Cramer-Shoup encryption [13]. (Recall that ciphertexts in tagged CS encryption include a public, non-malleable string called a tag.)

Outsource-secure Encryption: $Alg = (T, U)$
(in the one untrusted program model)

Global Setup (denoted gp as honest, unprotected inputs)

- Security parameter: 1^k.
- Global Encryption parameters: a group G of prime order q with generators g_1, g_2, a weakly collision-resistant hash function $H : \{0,1\} \rightarrow \mathbb{Z}_q$, and a statistically-hiding commitment scheme $Com : \{0,1\}^* \rightarrow \phi_C$ with a decommitment of ϕ_D.

Advertised Functionality of U: CCA2-Secure Cramer-Shoup Encryption [13]

- $U(pk, m, t) \rightarrow \tau = (u_1, u_2, e, v, t)$. (See Figure 1 for details.)

Subroutines Executed by T

- $Rand1 \rightarrow (b, g_1^b)$. (See Section 3.2 for details.)

Functionality of $Alg = (T, U)$: CCA2 and Outsource-Secure Cryptosystem

Key Generation: Generated by an honest process on input 1^k:

$PK = (B = g_1^{x_1} g_2^{x_2}, C = g_1^{y_1} g_2^{y_2}, D = g_1^z)$, $SK = (x_1, x_2, y_1, y_2, z)$.

Encryption: $Alg.Enc(m, (PK, t), gp, \varepsilon, \varepsilon) \rightarrow (\varepsilon, \phi_D, \tau)$.

On input $PK = (B, C, D)$, $m \in G$, and $t \in \{0,1\}^*$,

1. T computes $Rand1 \rightarrow (x_1', g_1^{x_1'})$, $Rand1 \rightarrow (y_1', g_1^{y_1'})$, $Rand1 \rightarrow (z', g_1^{z'})$.
2. T computes $PK' = (Bg_1^{x_1'}, Cg_1^{y_1'}, Dg_1^{z'})$.
3. T selects a random $w \in G$ and computes $\beta = wm$.
4. T computes $(\phi_C, \phi_D) = Com(\beta||t||x_1'||y_1'||z')$.
5. T calls $U(PK', w, \phi_C) \rightarrow \tau$, where $\tau = (u_1, u_2, e, v, \phi_C)$.
6. T outputs the ciphertext (τ, ϕ_D).

Decryption: $Alg.Dec(SK, (\tau, \phi_D), gp, \varepsilon, \varepsilon) \rightarrow (m, \varepsilon, \varepsilon)$.

(If E generates ciphertext, $Alg.Dec(SK, \varepsilon, gp, (\tau, \phi_D), \varepsilon) \rightarrow (\varepsilon, m, \varepsilon)$.)

On input $SK = (x_1, x_2, y_1, y_2, z)$ and $(\tau = (u_1, u_2, e, v, \phi_C), \phi_D)$,

1. T computes $(\beta||t||x_1'||y_1'||z') = Decom(\phi_C, \phi_D)$.
2. T computes $\hat{x_1} = x_1 + x_1'$, $\hat{y_1} = y_1 + y_1'$, $\hat{z} = z + z'$.
3. T computes $\kappa = H(u_1, u_2, e, \phi_C)$, $\alpha = u_1^{\hat{x_1} + \kappa\hat{y_1}}$, and $\pi = u_2^{x_2 + \kappa y_2}$.
4. T checks if $\alpha\pi = v$; if not, it outputs "invalid".
5. Otherwise, T computes $w = e/u_1^{\hat{z}}$ and outputs $m = \beta/w$ with tag t.

Fig. 2. An honest user T, given untrusted Cramer-Shoup encryption software U, achieves outsource-secure encryption. Note that the speed-up is for encryption only, not decryption.

4.1 *Com*: Efficient, Statistically-Hiding Commitments

We use Halevi and Micali's commitment scheme based on collision-free hash families [20]. Let $HF : \{0,1\}^{O(k)} \to \{0,1\}^k$ be a family of universal hash functions and let $MD : \{0,1\}^* \to \{0,1\}^k$ be a collision-free hash function. Given any value $m \in \{0,1\}^*$ and security parameter k, generate a statistically-hiding commitment scheme as follows: (1) compute $s = MD(m)$, (2) pick $h \in HF$ and $x \in \{0,1\}^{O(k)}$ at random, so that $h(x) = s$ and (3) compute $y = MD(x)$. (One can construct h by randomly selecting A and computing $b = s - Ax$ modulo a prime set in HF.) The commitment is $\phi_C = (y, h)$. The decommitment is $\phi_D = (x, m)$. Here, we denote the commitment scheme as *Com* and the decommitment scheme as *Decom*.

4.2 CCA2 and Outsource-Security of T^U Encryption

First, we observe that the Cramer-Shoup variant in Figure 2 is CCA2-secure [26]. Here, we only need to look at the *honest* algorithm T^U.

Theorem 3. *The cryptosystem T^U is secure against adaptive chosen-ciphertext attack (CCA2) assuming the CCA2-security of Cramer-Shoup encryption [13] and the security of the Halevi-Micali commitment scheme [20].*

The full proof of Theorem 3 is included in the full version of this paper. It follows a fairly standard reduction from tagged Cramer-Shoup.

Using *Exp*, we achieve the same asymptotic speed-up as in Section 3. Checking the output of this probabilistic functionality is theoretically impossible. Thus, we summarize the properties of this scheme as:

Theorem 4. *The algorithms $(T, U).Enc$ in Figure 2 are an $O(\frac{\log^2 n}{n})$-efficient, outsource-secure implementation of CCA2-secure encryption.*

Proof sketch. All inputs to U, besides the (honest, unprotected) global parameters, are computationally blinded by T. The public key is re-randomized using *Rand*1; a random message $w \in G$ is selected; and the tag ϕ_C, that binds these new values to the old key and message, is a statistically-hiding commitment. Thus, both S_1 and S_2 query U' on random triplets of the form $(PK \in (\mathbb{Z}_p^*)^3, w \in G, t \in \{0,1\}^{|\phi_C|})$. In pair one, S_1 always sets $replace^i = 0$, since the output of $T^{U'}$ in the real experiment is wrong with negligible probability. For efficiency, observe that the commitment scheme *Com* can be implemented with only a constant number of modular multiplications.

Acknowledgments. We are grateful to Ron Rivest for many useful discussions and for suggesting the use of the generators in Section 3.2. We also thank Srini Devadas, Shafi Goldwasser, Matt Lepinski, Alon Rosen, and the anonymous referees for comments on earlier drafts.

References

1. M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. *Journal of Comput. Syst. Sci.*, 39(1):21–50, 1989.
2. D. Beaver and J. Feigenbaum. Hiding instances in multioracle queries. In *Proceedings of STAC '90*, pages 37–48, 1990.
3. D. Beaver, J. Feigenbaum, J. Kilian, and P. Rogaway. Locally random reductions: Improvements and applications. *Journal of Cryptology*, 10(1):17–36, 1997.
4. M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: How to remove intractability assumptions. In *Proceedings of STOC*, pages 113–131, 1988.
5. M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, pages 269–291, 1995.
6. M. Blum, M. Luby, and R. Rubinfeld. Program result checking against adaptive programs and in cryptographic settings. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 107–118, 1991.
7. M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Science*, pages 549–595, 1993.
8. V. Boyko, M. Peinado, and R. Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In *Proceedings of Eurocrypt '98*, volume 1403 of LNCS, pages 221–232, 1998.
9. E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Proceedings of Eurocrypt '92*, volume 658 of LNCS, pages 200–207, 1992.
10. D. Chaum. Security without identification: transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.
11. D. Chaum and T. P. Pedersen. Wallet Databases with Observers. In *Proceedings of Crypto '92*, volume 740 of LNCS, pages 89–105, 1992.
12. D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Speeding up Exponentiation using an Untrusted Computational Resource. Technical Report Memo 469, MIT CSAIL Computation Structures Group, August 2003.
13. R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal of Computing*, 2003. To appear. Available at http://www.shoup.net/papers.
14. P. de Rooij. On the security of the Schnorr scheme using preprocessing. In *Proceedings of Eurocrypt '91*, volume 547 of LNCS, pages 71–80, 1991.
15. P. de Rooij. On Schnorr's preprocessing for digital signature schemes. In *Proceedings of Eurocrypt '93*, volume 765 of LNCS, pages 435–439, 1993.
16. P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Proceedings of Eurocrypt 1994*, volume 950 of LNCS, pages 389–399, 1994.
17. P. de Rooij. On Schnorr's preprocessing for digital signature schemes. *Journal of Cryptology*, 10(1):1–16, 1997.
18. M. Franklin and M. Yung. The blinding of weak signatures (extended abstract). In *Proceedings of Eurocrypt '95*, volume 950 of LNCS, pages 67–76, 1995.
19. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, 1996.

20. S. Halevi and S. Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *Proceedings of Crypto '96*, volume 1109 of LNCS, pages 201–212, 1996.
21. C. H. Lim and P. J. Lee. More flexible exponentiation with precomputation. In *Proceedings of Crypto '94*, volume 839 of LNCS, pages 95–107, 1994.
22. T. Matsumoto, K. Kato, and H. Imai. Speeding up secret computations with insecure auxiliary devices. In *Proceedings of Crypto '88*, volume 403 of LNCS, pages 497–506, 1988.
23. G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.
24. P. Q. Nguyen and I. Shparlinski. On the insecurity of a server-aided RSA protocol. In *Proceedings of Asiacrypt 2001*, volume 2248 of LNCS, pages 21–35, 2001.
25. P. Q. Nguyen, I. E. Shparlinski, and J. Stern. Distribution of modular sums and the security of server aided exponentiation. In *Proceedings of the Workshop on Comp. Number Theory and Crypt.*, pages 1–16, 1999.
26. C. Rackoff and D. Simon. Noninterative zero-knowledge proof of knowledge and chosen ciphertext attack. In *Proceedings of Crypto '91*, volume 576 of LNCS, pages 433–444, 1991.
27. C.-P. Schnorr. Efficient identification and signatures for smart cards. In *Proceedings of Crypto '89*, volume 435 of LNCS, 1989.
28. C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptography*, 4:161–174, 1991.
29. Trusted Computing Group. Trusted computing platform alliance, main specification version 1.1b, 2004. Date of Access: February 10, 2004.
30. D. A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, UC Berkeley, 12, 1999.
31. B. R. Waters, E. W. Felten, and A. Sahai. Receiver anonymity via incomparable public keys. In *Proceedings of the 10th ACM CCS Conference*, pages 112–121, 2003.