One-Time Programs

Shafi Goldwasser[1,3]*, Yael Tauman Kalai[2]**, and Guy N. Rothblum[3]***

[1] Weizmann Institute of Science, Rehovot, Israel
[2] Georgia Tech, Atlanta, USA
[3] MIT, Cambridge, USA

Abstract. In this work, we introduce *one-time programs*, a new computational paradigm geared towards security applications. A one-time program can be executed on a *single* input, whose value can be specified at run time. Other than the result of the computation on this input, nothing else about the program is leaked. Hence, a one-time program is like a black box function that may be evaluated once and then "self destructs." This also extends to k-time programs, which are like black box functions that can be evaluated k times and then self destruct.

One-time programs serve many of the same purposes of program obfuscation, the obvious one being software protection, but also including applications such as temporary transfer of cryptographic ability. Moreover, the applications of one-time programs go well beyond those of obfuscation, since one-time programs can only be executed once (or more generally, a limited number of times) while obfuscated programs have no such bounds. For example, one-time programs lead naturally to electronic cash or token schemes: coins are generated by a program that can only be run once, and thus cannot be double spent.

Most significantly, the new paradigm of one-time computing opens new avenues for conceptual research. In this work we explore one such avenue, presenting the new concept of "one-time proofs," proofs that can only be verified once and then become useless and unconvincing.

All these tasks are clearly impossible using software alone, as any piece of software can be copied and run again, enabling the user to execute the program on more than one input. All our solutions employ a secure memory device, inspired by the cryptographic notion of interactive oblivious transfer protocols, that stores two secret keys (k_0, k_1). The device takes as input a single bit $b \in \{0, 1\}$, outputs k_b, and then self destructs. Using such devices, we demonstrate that for every input length, any standard program (Turing machine) can be efficiently compiled into a functionally equivalent one-time program. We also show how this memory device can be used to construct one-time proofs. Specifically, we show how to use this device to efficiently convert a classical witness for any NP statement, into "one-time proof" for that statement.

* Supported by NSF Grants CCF-0514167, CCF-0635297, NSF-0729011, the RSA chair, and by the Weizmann Chais Fellows Program for New Scientists, shafi@theory.csail.mit.edu.

** Supported in part by NSF grant CCF-0635297, yael@cc.gatech.edu.

*** Supported by NSF Grants CCF-0635297, NSF-0729011, CNS-0430336 and by a Symantec Graduate Fellowship, rothblum@csail.mit.edu.

1 Introduction

In our standard computing world (and the standard theoretical computing models), computer programs can be copied, analyzed, modified, and executed in an arbitrary manner. However, when we think of security applications, such complete transfer of code is often undesirable, as it complicates and inhibits tasks such as revocation of cryptographic ability, temporary transfer of cryptographic ability, and preventing double-spending of electronic cash. Other tasks such as general program obfuscation [BGI+01,GK05] are downright impossible.

In this paper, we propose to study a new type of computer program, called a *one-time program*. Such programs can be executed only once, where the input can be chosen at any time. This notion extends naturally to k-time programs which can be executed at most k times on inputs that can be chosen by the user at any time. These programs have immediate applications to software protection, electronic tokens and electronic cash. Even more interestingly, they open new avenues for conceptual contributions. In particular, in this work they allow us to conceive of, define and realize the new cryptographic concept of *one-time zero-knowledge proofs*: zero-knowledge proofs that can be verified exactly once, by any verifier, without the prover being present. After the proof is verified once by any single verifier, it becomes useless and cannot be verified again.

Clearly a one-time program cannot be solely software based, as software can always be copied and run again, enabling a user to execute the program more times than specified. Instead, we suggest the use of a secure memory devices, *one-time memory* (OTM), as part of the one-time program. In general, when using a hardware device it is crucial that the device be as simple as possible, so that it can be scrutinized more easily. In particular, side-channel attacks have emerged as a devastating threat to the security of hardware devices.

The memory device used in this work is very simple and withstands even extremely powerful side-channel attacks. An OTM does not perform any computation, but its memory contents are assumed to be somewhat protected, i.e. they cannot be read and written arbitrarily. All we assume is that memory locations that are *never accessed* by the device are *never leaked* via a side channel (whereas memory that *is* accessed may be immediately leaked), and that the device has a *single* tamper-proof bit, see Section 3 for a fuller discussion. In particular, our device meets the desirable property laid out in the work of Gunnar *et. al.* [GLM+04], and can be decoupled into two components: the first component is tamper-proof but readable, and consists of a single bit. The second component is tamperable but read-proof. As mentioned above, in our case the read-proof requirement is only for memory locations that are never accessed by the device. These assumptions seem minimal if any non-trivial use is to be made of the secure device (see the illuminating discussions in Micah and Renin [MR04] and in Gunnar *et. al.* [GLM+04]). Also, the device is very inexpensive, low energy and disposable, much like RFID tags used in clothing. Thus, a one-time program can be realized by a combination of standard software and such minimally secure memory devices.

We construct a universal one-time compiler that takes *any* standard polynomial-time program and memory devices as above, and transforms it (in polynomial time) into a one-time program which achieves the same functionality, under the assumption that one-way functions exist. This compiler uses techniques from secure function evaluation, specifically Yao's garbled circuit method [Yao86], as its starting point. These techniques, however, only give solutions for settings in which adversaries are honest-but-curious, whereas we want the security of one-time programs to also hold against malicious adversaries. Unlike the setting of secure function evaluation, we need to overcome this difficulty without the benefit of interaction. This is accomplished (non-interactively) using the secure memory devices (see Section 4 for details).
While we cannot show that this compiler is optimal in terms of the efficiency of the one-time programs it produces, we do argue that significant improvements would resolve a central open problem in cryptography. Specifically, significant complexity improvements would imply significant improvements in the communication complexity of secure-function-evaluation protocols. See Section 4 for further details.
Continuing in the spirit of one-time computing, we also define and construct *one-time proofs*. A one-time proof system for an NP language L allows the owner of a witness for the membership of some input x in L to transform this witness into a one-time proof token (a device with the above secure memory components). This proof token can be given to any efficient prover, who does not know a witness for x. The prover can use this token *exactly once* to prove to any verifier that $x \in L$ using an interactive proof. The prover does not learn anything from the proof token, and in particular cannot prove that $x \in L$ a second time. The witness owner does not need to be involved in the interaction between the prover and verifier. We show how to construct a one-time proof system with negligible soundness for any NP language. Achieving constant soundness is relatively straightforward, but amplifying the soundness is not. The technical difficulties are similar to those encountered in parallel composition of zero-knowledge proofs. We are able to resolve these difficulties (again, in a non-interactive manner) using the secure memory devices. See Section 5.2 for an overview.
We proceed to describe our contributions in detail.

2 One-Time Programs and One Time Compilers

Informally, a *one-time program for a function* f: (1) Can be used to compute f on a single input x of one's choice. (2) No efficient adversary, given the one-time program, can learn more about f than can be learned from a single pair $(x, f(x))$, where x is chosen by the adversary. Hence, it acts like a black-box function that can only be evaluated once.
Several formal definitions can be proposed for condition 2 above. We chose a definition inspired by Goldreich and Ostrovsky's [GO96] work on software protection, and the work of Barak *et al.* [BGI$^+$01] on program obfuscation. Informally, for every probabilistic polynomial time algorithm A given access to a one-time program for f on inputs of size

n, there exists another probabilistic polynomial time algorithm $S(1^n)$ which can request to see the value $f(x)$ for an x of its choice where $|x| = n$, such that (for any f) the output distributions of A and S are computationally indistinguishable, even to a machine that knows f!

The notion of one-time programs extends naturally to k-time programs which can be provably executed at most k times on input values that can be chosen by the user at any time. For simplicity of exposition we mostly deal with the one-time case throughout this paper.

As previously mentioned, a one-time program cannot be solely software based, and we propose to use secure hardware devices as building blocks in the constructions. In general, we model *secure hardware devices* as black-boxes with internal memory which can be accessed only via its I/O interface. A *one-time program* is a combination of *hardware*: (one or many) hardware devices $\mathcal{H}_1, \ldots, \mathcal{H}_m$; and *software*: a (possibly non-uniform) Turing machine $\mathcal{M}$, where the machine $\mathcal{M}$ accesses the hardware devices via their I/O interface. It is important to note that the Turing machine software component *is not secure*: it can be read and modified by the user whereas the access to the secure hardware is only via the I/O interface. Thus, we view one-time programs as software-hardware packages. An execution of one-time program $\mathcal{P} = \mathcal{M}^{\mathcal{H}_1,\ldots,\mathcal{H}_m}$ on input $x \in \{0,1\}^n$, is a run of $\mathcal{M}^{\mathcal{H}_1,\ldots,\mathcal{H}_m}(x)$, where the contents of $\mathcal{M}$'s output tape is the output. Throughout this work we use a new type of secure hardware device which we name a *one-time memory* (OTM), see the introduction and Section 3 for more details.

One-Time Compiler. To transform a standard computer program into a one-time program computing the same functionality, we propose the notion of a *one-time compiler*. The compiler takes a computer program, modeled as a (possibly non-uniform) Turing machine $\mathcal{T}$, an input length n, and a collection of OTM devices (the number of devices may depend on n). The compiler then creates a one-time program computing the same functionality as $\mathcal{T}$ on inputs of length n by initializing the internal memory of the OTMs and also outputting (in the clear) a software component for the program. It is also important that the compiler be efficient, in the sense that its running time is no worse than polynomial in $\mathcal{T}$'s worst-case running time on inputs of length n.

In this work we construct a one-time compiler as above. The compiler transforms any Turing machine $\mathcal{T}$ into a one-time program $\mathcal{P}$ that satisfies the two intuitive properties outlined above:

1. **Functionality.** For any $x \in \{0,1\}^n$, when the program is run *once* on input x it outputs $\mathcal{T}(x)$; namely, $\mathcal{P}(x) = \mathcal{T}(x)$.
2. **One-Time Secrecy.** For any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ with *one-time* oracle access to the machine $\mathcal{T}$. The simulator gets the machine's output, running time and space usage on a single input of its choice, and its running time is polynomial in the machine $\mathcal{T}$'s worst-case running time (and in n). We require that for any machine $\mathcal{T}$ the following two distributions are indistinguishable:
   (a) The output of the adversary $\mathcal{A}$ when it is run with arbitrary access to the one-time program $\mathcal{P}$ (i.e. full access to the software component and black-box access to the hardware component).
   (b) The output of the simulator with one-time oracle access to $\mathcal{T}$.

Moreover, the indistinguishability holds even for a distinguisher who takes $\mathcal{T}$ as input. Note that it is crucial that the simulator $\mathcal{S}$ *only accesses its oracle once*.[4] Also note that the simulator cannot access any part of the actual one-time program, including the hardware, not even in a black-box manner.

See the full version of this work for a more rigorous treatment.

Remark. Note that in the above definition we only allow the adversary black-box access to the hardware component. Thus, we implicitly assume that the hardware devices withstand all side channel attacks. This is a very strong (and perhaps unreasonable) assumption. However, as we shall see next, the actual security assumptions we impose on the memory devices we use are much weaker, and in some sense are minimal.

3 One-Time-Memory Device (OTM)

Informally, a OTM is a memory device initialized with two keys (k_0, k_1). It takes as input a single bit $b \in \{0, 1\}$, outputs k_b and "self destructs". There are several ways to formalize the concept of self destruction. The first would be to erase both keys after outputting k_b. However, to circumvent side-channel attacks, we prefer that the device never access the key not retrieved. Instead, we choose the following formalism.

An OTM is initialized with two keys (k_0, k_1), and one additional tamper-proof bit set to 0. The OTM input is a single bit $b \in \{0, 1\}$. Upon receiving an input, the OTM verifies that the tamper-proof bit is set to 0, sets it to 1 and outputs k_b. If the tamper-proof bit is not 0, the device outputs an error symbol $\perp$. Thus, an OTM outputs one of its two keys, and the other key is irretrievably lost (and never accessed).

Our security assumptions from the OTM device are quite minimal:

1. The memory locations that are *never accessed* by the device are *never leaked* via a side channel, whereas a memory cell that *is* accessed may be immediately leaked.
2. The *single* bit b is tamper-proof (but is *readable*).

Intuitively, the above two assumptions imply that the device is as secure as a black-box. See the full version of this work for details.

OTM's are inspired by the cryptographic notion of *one out of two oblivious transfer* [Rab05,EGL85], where a sender holds two keys and lets a receiver receive one of them. The key not chosen is irrevocably lost, and the sender does not learn (is oblivious to) which key the receiver received. Whereas an oblivious transfer is an interactive protocol, an OTM is a physical device. The important requirement from an OTM is that the user using the device (analogous to the "receiver" in an oblivious transfer protocol) learns only the secret of his choice. The requirement that the key generator (analogous to the "sender") is oblivious and does not learn which secret the user received, makes little sense in our setting as the OTM is at the hands of the "receiver" and the key generator is no longer present at the time that the OTM is used.

[4] This guarantees that the one-time program cannot be duplicated and run more than once. The simulator certainly cannot duplicate, and thus an adversary who can obtain two of the program's outputs cannot be simulated.

3.1 Using OTMs vs. Other Hardware

There are many possible secure hardware devices one could conceive of using, and it is not a-priori clear whether one device is better or worse than another. This is a central question in the study of one-time programs, and secure hardware in general. In this section we compare the use of OTM devices in one-time programs with alternative solutions that use different hardware.

Task Specific Hardware. A trivial solution would be to build for each function f a special-purpose task-specific secure hardware device which computes f for one input x and then refuses to work any longer. We find this solution highly unsatisfactory, for several reasons:

- *Universality.* First, this approach calls for building a different hardware device for each different function f. This may be worthwhile for some tasks, but is too costly for most tasks and thus infeasible in practice. Instead, we advocate that the secure hardware device of choice should be *universal*. Namely, that the hardware device be task-independent, and "programmable" so it can be used to construct a one-time program for *any* functionality (one-time programs for different functions will differ in their software component). In the case of secure hardware (rather than ordinary hardware), universality is particularly important, as each type of hardware device needs to be intensely scrutinized to try to guarantee that it is not susceptible to side channel attacks. This seems impossible to do on a function by function basis.

- *Simplicity.* Second, perhaps the most central measure of reasonability for a secure hardware device is its *simplicity*, and the trivial solution suggested above is potentially *complex* as it requires producing complex hardware for complex functions. Our search for *simple* hardware devices, which are easy to build, analyze and understand, is motivated by several concerns; (i) The assumption that a hardware device is secure and/or tamper-proof is a very strong assumption, as one has to consider all possible *physical attacks*. The simpler a hardware device is, the easier it is to scrutinize and analyze its security, and the more reasonable the assumption that it is secure becomes. (ii) Continuing in this vein, *side channel attacks* have emerged as a significant threat to the integrity of cryptographic algorithms and devices (see e.g. Anderson [And01]). It seems intuitive that the less computation the hardware preforms, the less susceptible it will be to potentially devastating side-channel attacks. Indeed, this is the guiding principle behind the theoretical approach to defining physically secure devices taken by [MR04,GLM$^+$04]. In the case of task-specific hardware, ad absurdum the entire the computation can be done by the hardware. (iii) Finally, and perhaps most obviously, the simpler a hardware device is, the easier and cheaper to build it will be.

Secure General Purpose CPU. An alternate solution would be to use the physically shielded full-blown CPU, which was proposed by Best [Bes79] and Kent [Ken80] and used in the work of Goldreich and Ostrovsky on software protection [GO96]. This CPU contains a protected ROM (read only memory) unit in which a secret decryption key is written. The I/O access to the shielded CPU is through fetch instructions. In

each computation cycle, the CPU fetches an encrypted instruction, decrypts it, and executes it. The hardware security assumption here is that both the cryptographic operations (decryption, encryption etc.), as well as the general-purpose computation operations, are perfectly shielded. Each such shielded CPU was associated with a different decryption key, and the encrypted software executed on it was to be encrypted with the matching encryption key. Goldreich-Ostrovsky envisioned protecting a software program by encrypting it and packaging it with a physically shielded CPU with a matching decryption key. One-time programs can be easily built in the same manner (adding a counter to the CPU to limit the number of executions).
This solution is certainly universal, as the CPU can compute all tasks. Yet, we do not believe it is suitable for the one-time programs application:

- *Simplicity.* We consider a full blown protected CPU to be far from the goal of hardware simplicity, and so complex as to make the Goldreich-Ostrovsky approach unviable for the design of one-time programs. This is evidenced by the simple fact that although these devices were first proposed in the early 1980's, they still seem beyond the reach of current day technology in terms of cost. It seems that in particular for the application of one-time programs, using a full-blown shielded CPU for computing one task a limited number of times is an overkill.

- *Side Channel Attacks.* Secure CPUs perform complex computations (both cryptographic and otherwise), and are thus susceptible to side-channel attacks. If we assume, when modeling side channel attacks, that each computational step may leak information about bits accessed by the hardware, the [GO96] device becomes especially vulnerable: once the secret key (which is used in every step of the computation) leaks, the security of the entire construction falls apart.

Now, we can re-examine OTMs in light of the above alternative suggestions. As we show in Section 4, OTMs are*universal* task-independent devices that can be used to make any program one-time. Moreover, an OTM is also a *simple* device. Most importantly, even if we assume that side-channel adversaries can capture every bit accessed by the hardware during a computation step, the OTM construction remains secure, as long as there is a single (readable) tamper-proof bit! The OTM key that is not chosen is *never accessed*, and thus OTM constructions are secure under the (seemingly minimal, see [MR04]) assumption that untouched memory bits are not leaked.
So far, the comparison between OTM and other secure hardware was qualitative. We now present some several quantitative complexity measures for analyzing secure hardware devices and their use by a one-time program. In the next section we shall see how our solution fares in comparison to the [GO96] hardware with respect to this complexity measures.

- *Hardware Runtime.* The total combined running time of all the hardware devices used by the one-time program. This measures the amount of computation done by the secure hardware devices (e.g. number of operations done by their CPU), and *not* the amount of computation done by the one-time program's software component. Clearly, it is desirable for the hardware to do as little work as possible, both because simple

devices will be computationally much weaker than the CPU of a modern personal computer and because the more computation is done on a device the more susceptible it may become to side-channel attacks.
We also consider the *total runtime* of the one-time program, which is the combined runtime of the hardware and software components.

- *Size.* The combined sizes of all the hardware devices used by the one-time program. The size of a hardware device is the size of its (persistent) memory together with the size of its control program. The smaller the hardware device, the better, as protecting smaller memories is easier.

- *Latency.* Number of times the one-time program $\mathcal{P}$ accesses its secure hardware devices. We assume that each time hardware devices are accessed, many of them may be queried in parallel, but we want to minimize the number of (adaptive) accesses to the hardware devices, both to guarantee that the hardware is not involved in complex computations and to optimize performance (as accessing hardware is expensive).

4 A One-Time Compiler

In what follows, we present an efficient one-time compiler that uses OTMs, give an overview of the construction, compare it to other solutions from the literature, and conclude with a discussion on the implications of improvements to our results.

The Construction. Building on the ideas in Yao's Garbled-Circuit construction [Yao86], we demonstrate that a universal one-time compiler exists using OTMs. First, convert the input (Turing machine) program into a Boolean circuit on inputs of length n. Second, garble it using Yao's method. And, finally, use n OTM's to transform the garbled circuit into a one-time program. We encounter an obstacle in this last step, as the security of Yao's construction is only against honest-but-curious adversaries, whereas the one-time program needs to be secure against any malicious adversary. In the secure function evaluation setting this is resolved using interaction (e.g. via zero-knowledge proofs, see [GMW91]), or using some global setup and non-interactive zero-knowledge proofs. Our setting of one-time programs, however, is not an interactive setting, and we cannot use these solutions. Instead, we present a solution to this problem that uses the OTM devices; see below.

Informal Theorem 1: Assume that one-way functions exist. There exists an efficient one-time compiler $\mathcal{C}$ that for input length n uses n OTMs: $\mathcal{B}_1, \ldots, \mathcal{B}_n$. For any (non-uniform) Turing machine $\mathcal{T}$, with worst-case running time $t_{\mathcal{T}}(n)$ (on inputs of length n), the compiler $\mathcal{C}$, on input $1^n, 1^{t_{\mathcal{T}}(n)}$, description of $\mathcal{T}$ and security parameter $1^{\kappa(n)}$, outputs a one-time program $\mathcal{P} \triangleq \mathcal{M}^{\mathcal{B}_1(v_1),\ldots,\mathcal{B}_n(v_n)}$ such that $\mathcal{P}(x) = \mathcal{T}(x)$ for inputs of length n. Let $t_{\mathcal{T}}(x)$ denote $\mathcal{T}$'s running time on input $x \in \{0,1\}^n$. Then, $\mathcal{P}(x)$ achieves: *latency* 1; *hardware runtime* $n \cdot \kappa(n)$; *total running time* $\tilde{O}(t_{\mathcal{T}}(x) \cdot \text{poly}(\kappa, n))$; and *size* $\tilde{O}(t_{\mathcal{T}}(n) \cdot \text{poly}(\kappa))$.

Proof (Construction Overview). We begin by briefly reviewing Yao's Garbled Circuit construction. We assume (for the sake of simplicity)

that the underlying Turing machine has a boolean (1 bit) output, but the construction is easily generalized to multi-bit outputs while maintaining the performance claimed in the theorem statement. The construction proceeds by converting the Turing machine $\mathcal{T}$ into a boolean circuit of size $\tilde{O}(t_{\mathcal{T}}(n))$ and then garbling it using Yao's garbled circuit method. This gives a garbled circuit $G(\mathcal{T})$ of size $\tilde{O}(t_{\mathcal{T}}(n) \cdot \text{poly}(\kappa))$, together with n key-pairs $(k_1^0, k_1^1) \ldots (k_n^0, k_n^1)$. The construction guarantees both (i) **Correctness**: namely there is an efficient algorithm that for any input $x \in \{0,1\}^n$ takes as input $G(\mathcal{T})$ and only the n keys $k_1^{x_1}, \ldots, k_n^{x_n}$ (one from each pair of keys), and outputs $\mathcal{T}(x)$. The algorithm's running time is $\tilde{O}(t_{\mathcal{T}}(x) \cdot \text{poly}(\kappa))$. The construction also guarantees (ii) **Privacy**: an adversary cannot learn more from the garbled circuit together with one key out of each key-pair, say the x_i-th key from the i-th key pair, than the output of the machine on the input $x = x_1 \circ \ldots \circ x_n$. Formally, there exists an efficient simulator S such that for any machine $\mathcal{T}$, for any output value b and for any input x such that $C(x) = b$, the simulator's output on input $(b, x, 1^{t_{\mathcal{T}}(n)}, 1^{\kappa})$ is indistinguishable from $(x, G(\mathcal{T}), k_1^{x_1}, \ldots, k_n^{x_n})$.
We want to use Yao's garbled circuit to build one-time programs using OTMs. A deceptively straightforward idea for a one-time compiler is to use n OTMs: garble the machine $\mathcal{T}$, and put the i-th key pair in the i-th OTM. To compute $\mathcal{T}$'s output on an input x a user can retrieve the proper key from each OTM and use the correctness of the garbled circuit construction to get the machine's output. Privacy may seem to follow from the privacy of the garbled circuit construction. Surprisingly, however, the above construction does *not* seem to guarantee privacy. In fact, it hides a subtle but inherent difficulty.
The difficulty is that the privacy guarantee given by the garbled circuit construction is too weak. At a higher level this is because the standard Yao construction is in the *honest-but curious setting*, whereas we want to build a program that is one-time even against *malicious* adversaries. More concretely, the garbled circuit simulator generates a dummy garbled circuit *after the inputx is specified*, i.e. only after it knows the circuit's output $\mathcal{T}(x)$. This suffices for honest-but-curious two-party computation, but it is not sufficient for us. The (malicious) one-time program adversary may be adaptive in its choice of x: *the choice of x could depend on the garbling itself, as well as the keys revealed as the adversary accesses the OTMs.* This poses a problem, as the simulator, who wants to generate a dummy garbling and then run the adversary on it, does not know in advance on which input the adversary will choose to evaluate the garbled circuit. On closer examination, the main problem is that the simulator does not know in advance the circuit's output on the input the adversary will choose, and thus it does not know what the dummy garbling's output should be. Note that we are not in an interactive setting, and thus we cannot use standard solutions such as having the adversary commit to its input x before seeing the garbled circuit.
To overcome this difficulty, we need to change the naive construction that we got directly from the garbled circuit to give the simulator more power. Our objective is allowing the simulator to "hold off" choosing the output of the dummy garbling until the adversary has specified the

input. We do this by "hiding" a random secret bit b_i in the i-th OTM, this bit is exposed no matter which secret the adversary requests. These n bits mask (via an XOR) the circuit's output, giving the simulator the flexibility to hold off "committing" to the *unmasked* garbled circuit's output until the adversary has completely specified its input x (by accessing all n of the OTMs). The simulator outputs a garbled dummy circuit that evaluates to some random value, runs the adversary, and once the adversary has completely specified x by accessing all n OTMs, the simulator can retrieve $\mathcal{T}(x)$ (via its one-time $\mathcal{T}$-oracle), and the last masking bit it exposes to the adversary (in the last OTM the adversary accesses) always unmasks the garbled dummy circuit's output to be equal to $\mathcal{T}(x)$.

Our Scheme vs. the [GO96] Scheme. Note that in the [GO96] scheme, both the hardware runtime and the latency is the same as the *entire* running time of the program, whereas in our scheme (using OTMs) the latency is 1 and the hardware runtime is $n \cdot \kappa(n)$ (independent of the program runtime). On the other hand, one advantage of the [GO96] scheme is that the size of the entire one-time program is proportional to the size of a single cryptographic key (independent of the program runtime), whereas in our scheme the size is quasi-linear in the (worst-case) runtime of the program.

4.1 Can we Get the Best of Both Worlds?

The primary disadvantage of our construction is the *size* of the one-time program. The "garbled circuit" part of the program (the software part) is as large as the (worst-case) running time of the original program. It is natural to ask whether this is inherent. In fact, it is not, as one-time programs based on the Goldreich-Ostrovsky construction (with a counter limiting the number of executions) have size only proportional to the *size* of the original program and a cryptographic key. However, as discussed above, the Goldreich-Ostrovsky solution requires complex secure hardware that runs the *entire computation* of the one-time program.
It remains to ask, then, whether it is possible to construct one-time programs that enjoy "the best of both worlds." I.e. to build *small* one-time programs with *simple hardware that does very little work*. This is a fundamental question in the study of one-time programs. Unfortunately, we show that building a one-time program that enjoys the best of both worlds (small size and hardware running time) is beyond the reach of current knowledge in the field of cryptography. This is done by showing that such a construction would resolve a central open problem in foundational cryptography: it would give a secure-function-evaluation protocols with sub-linear (in the computation size) communication complexity.

Informal Theorem 2: Assume that for every security parameter κ, there exists a secure oblivious transfer protocol for 1-bit message pairs with communication complexity poly(κ). Fix any input length n and any (non-uniform) Turing machine $\mathcal{T}$. Suppose $\mathcal{P}$ is a one-time program corresponding to $\mathcal{T}$ (for inputs of length n). If $\mathcal{P}$ is of total size $s(n)$ and the worst-case (combined) running time of the secure hardware(s) on an n-bit input is $t(n)$, then there exists a secure function evaluation protocol where Alice has input $\mathcal{T}$, Bob has input $x \in \{0,1\}^n$, at the end of the protocol Bob learns $\mathcal{T}(x)$ but nothing else, Alice learns nothing, and the total communication complexity is $s(n) + O(t(n) \cdot \text{poly}(\kappa))$.

Let us examine the possibility of achieving the best of both worlds in light of this theorem. Suppose we had a one-time compiler that transforms any program $\mathcal{T}$ into a one-time program with simple secure hardware that does $O(n \cdot \kappa)$ work and has total size $O(|\mathcal{T}| + \kappa)$. By the above theorem, this would immediately give a secure function evaluation protocol where Alice has $\mathcal{T}$, Bob has x, Bob learns only $\mathcal{T}(x)$ and Alice learns nothing, *with linear in* $(n, |\mathcal{T}|, \kappa)$ *communication complexity!* All known protocols for this problem have communication complexity that is at least linear *in* $\mathcal{T}$*'s running time*, and constructing a protocol with better communication complexity is a central open problem in theoretical cryptography. For example, this is one of the main motivations for constructing a fully homomorphic encryption scheme. See the full version for more details.

5 One-Time Programs: Applications

One-time programs have immediate applications to software protection. They also enable new applications such as one-time proofs, outlined below. Finally, OTMs and one-time programs can be used to construct electronic cash and electronic token schemes [Cha82,Cha83]. The E-cash applications and the discussion of related work are omitted from this extended abstract for lack of space, see the full version for details.

5.1 Extreme Software Protection

By the very virtue of being one-time programs, they cannot be reverse engineered, copied, re-distributed or executed more than once.

Limiting the Number of Executions. A vendor can put an explicit restriction as to the number of times a program it sells can be used, by converting it into a one-time program which can be executed for at most k times. For example, this allows vendors to supply prospective clients with "preview" versions of software that can only be used a very limited number of times. Unlike techniques that are commonly employed in practice, here there is a *guarantee* that the software cannot be reverse-engineered, and the component that limits the number of executions cannot be removed. Moreover, our solution does not require trusting a system clock or communicating with the software over a network (as do many solutions employed in practice). This enables vendors to control (and perhaps charge for) the way in which users use their software, while completely maintaining the user's privacy (since the vendors never see users interacting with the programs). One-time programs naturally give solutions to such copy-protection and software protection problems, albeit at a price (in terms of complexity and distribution difficulty).

Temporary Transfer of Cryptographic Ability. As a natural application for one-time programs, consider the following setting, previously suggested by [GK05] in the context of program obfuscation. Alice wants to go on vacation for the month of September. While she is away, she would like to give her assistant Bob the power to decrypt and sign E-mails dated "September 2008" (and only those E-mails). Alice can now supply Bob with many one-time programs for signing and decrypting messages dated "September 2008". In October, when Alice returns, she is guaranteed that Bob will not be able to decrypt or sign any of her messages! As long as Alice knows a (reasonable) upper bound for the number of expected messages to be signed and decrypted, temporarily transferring her cryptographic ability to Bob becomes easy.

5.2 One-Time Proofs

The one-time paradigm leads to the new concept and constructions of *one-time proofs*: proof tokens that can be used to prove (or verify) an NP statement exactly once.

A one-time proof system for an NP language L consists of three entities: (i) a witness *owner* who has a witness to the membership of some element x in a language L, (ii) a prover, and (iii) a verifier, where the prover and verifier know the input x but do not know the witness to x's membership in L. A one-time proof system allows the witness owner to (efficiently) transform its NP witness into a hardware based *proof token*. The proof token can later be used by the efficient prover (who does not know a witness) to convince the verifier **exactly once** that the input x is in the language. The witness owner and the verifier are assumed to be "honest" and follow the prescribed protocols, whereas the prover may be malicious and deviate arbitrarily.[5]

In a one-time proof, the prover convinces the verifier by means of a standard interactive proof system. In particular, the verifier doesn't need physical access to the proof token (only the prover needs this access). After running the interactive proof and convincing the (honest) verifier once, the proof token becomes useless and cannot be used again. The point is that (i) the witness owner does not need to be involved in the proof, beyond supplying the token (hence the proof system is off-line), and (ii) the prover, even though it convinces the verifier, learns nothing from interacting with the hardware, and in particular cannot convince the verifier a second time. Thus, for any NP statement, one-time proofs allow a witness owner to give other parties the capability to prove the statement in a controlled manner, without revealing to them the witness. A one-time proof system gives this "one-time proof" guarantee, as well as the more standard completeness and soundness guarantees, and a zero-knowledge guarantee (see [GMR89]), stating that anything that can be learned from the proof token, can be learned without it (by a simulator). Finally, a user who wants to use the one-time proof token to convince himself that $x \in L$ can do so without any interaction by running the

[5] The case where even verifiers do not behave honestly is also interesting, see the full version of this work for a discussion.

interactive proof in his head (in this case the prover and verifier are the same entity).

Note that a prover who somehow does know a witness to x's membership can convince the verifier as many times as it wants. How then can one capture the one-time proof requirement? We do this by requiring that any prover who can use a single proof token to convince the verifier more than once, must in fact know a witness to x's membership in the language. Formally, the witness can be *extracted* from the prover in polynomial time. In particular, this means that if the prover can convince the verifier more than once using a single proof token, then the prover could also convince the verifier as many times as it wanted without ever seeing a proof token! In other words, the proof token does not help the prover prove the statement more than once.

Another natural setting where one-time proofs come up is in voting systems, where the goal is to ensure that voters can only vote once. In this setting, each voter will be given a one-time proof for possessing the right to vote (of course, one must also ensure that the proof tokens cannot be transferred from one voter to another). A similar application of one-time proofs is for electronic subway tokens. Here the subway operator wants to sell electronic subway tokens to passengers, where the tokens should be verifiable by the subway station turnstiles.[6] A passenger should only be able to use a token once to gain entrance to the subway, and after this the token becomes useless. This goal is easily realized in a natural way by one-time proofs. The subway operator generates a hard cryptographic instance, say a product $n = p \cdot q$ of two primes. Subway tokens are one-time proof tokens for proving that n is in fact a product of two large primes. The passengers play the role of the prover. The turnstiles are the verifier, and only let provers who can prove to them that n is a product of two primes into the subway station. Any passenger who can use a single token to gain entrance more than once, can also be used to find a witness, or the factorization of n, a task which we assume is impossible for efficient passengers.[7]

More generally, one can view one-time proofs as a natural generalization of count-limited access control problems. In particular, we can convert any 3-round ID scheme (or any Σ-protocol) into a one-time proof of identity. See also the application to the E-token problem presented in the next section.

One-time proofs are different from non interactive zero knowledge (NIZK) proofs (introduced by [BFM88]). In both cases, the witness owner need not be present when the proof is being verified. However, in NIZK proof systems either the proof can be verified by arbitrary verifiers an unlimited number of times, and in particular is also not deniable [Pas03] (for example, NIZK proof systems in a CRS-like model, as in [BFM88]), or the proofs have to be tailored to a specific verifier and are useless to other verifiers (for example, NIZK proof systems in the pre-processing

[6] This problem was originally suggested by Blum [Blu81]. A scheme using quantum devices (without a proof of security) was proposed by Bennett *et al.* [BBBW82]

[7] For simplicity we do not consider here issues of composability or maintaining passenger's anonymity.

model [SMP88]). One-time zero knowledge proofs, on the other hand, can only be verified once, but by *any* user, and are later deniable. They also do not need a trusted setup, public-key infrastructure, or pre-processing, but on the other hand they do use secure hardware.

In the full version of this work we define one-time proofs and show (assuming one-way permutations) that any NP statement has an efficient one-time proof using OTMs. To attain small soundness we need to overcome problems that arise in the parallel composition of zero-knowledge proof (but in a non-interactive setting). This is accomplished by using the secure hardware to allow a delicate simulation argument. While we note that the general-purpose one-time compiler from the previous section can be used to construct a one-time proof,[8] this results in considerably less efficient (and less intuitively appealing) schemes.

Informal Theorem 3: Let κ be a security parameter and k a soundness parameter. Assume that there exists a one-way permutation on κ-bit inputs. Every NP language L has a one-time zero-knowledge proof with perfect completeness and soundness 2^{-k}. The proof token uses k OTMs (each of size poly(n, k, κ), where n is the input length).

Construction Overview. We construct a one-time proof for the NP complete language Graph Hamiltonicity, from which we can derive a one-time proof for any NP language. The construction uses ideas from Blum's [Blu87] protocol for Graph Hamiltonicity. The input is a graph $G = (V, E)$, the producer has a witness w describing a hamiltonian cycle in the graph. The one-time proof uses k OTMs to get a proof with soundness 2^{-k} (and perfect completeness).

The basic idea of Blum's zero-knowledge proof is for a prover to commit to a random permutation of the graph and send this commitment to the verifier. The verifier can then ask the prover wether to send it the permutation and all the de-commitments (openings of all the commitments), or to send de-commitments to a Hamiltonian cycle in the permuted graph. The proof is zero-knowledge, with soundness $1/2$.

A natural approach for our setting is for the witness owner to generate a proof token that has the committed permuted graph as a software component. The proof token also includes an OTM whose first secret is the permutation and all the de-commitments (the answer to one possible verifier query in Blum's protocol), and whose second secret is de-commitments to a Hamiltonian cycle in the permuted graph (the answer to the second verifier query). This indeed gives a (simple) one-time proof with soundness $1/2$ via standard arguments: the only thing a prover can learn from the token is one of the two possible de-commitment sequences, and we know (from Blum's zero-knowledge simulator), that the prover could generate this on its own. On the other hand, somewhat paradoxically, this proof token does allow a prover to convince a verifier that the graph has a Hamiltonian cycle in an interactive proof with perfect completeness and soundness $1/2$.

[8] To do this, just generate a one-time program computing the prover's answers.

To amplify the soundness to 2^{-k}, a seemingly effective idea is to have the producer produce k such committed graphs and k such corresponding OTMs, each containing a pair of secrets corresponding to a new commitment to a random permutation of its graph. This idea is, however, problematic. The difficulty is that simulating the one-time proof becomes as hard as simulating parallel executions of Blum's zero-knowledge protocol. Namely, whoever gets the OTMs can choose which of the two secrets in each OTM it will retrieve as a function of *all* of the committed permuted graphs. In the standard interactive zero-knowledge proof setting this is resolved by adding interaction to the proof (see Goldreich and Kahan [GK96]). However, in our setting it is crucial to avoid adding interaction with the witness owner during the interactive proof phase, and thus known solutions do not apply. In general, reducing the soundness of Blum's protocol without adding interaction is a long-standing open problem (see e.g. Barak, Lindell and Vadhan [BLV06]). In our setting, however, we can use the secure hardware to obtain a simple solution.
To overcome the above problem, we use the secure hardware (OTMs) to "force" a user who wants to gain anything from the proof token, to access the boxes in sequential order, independently of upcoming boxes, as follows. The first committed graph C_1 is given to the user "in the clear", but subsequent committed graphs, C_i, $i \geq 2$, are not revealed until all of the prior $i-1$ OTMs (corresponding to committed graphs $C_1, ..., C_{i-1}$) have been accessed. This is achieved by, for each i: (1) splitting the description of the i-th committed graph C_i into $i-1$ random strings $m_i^1, ...m_i^{i-1}$ (or shares) such that their XOR is C_i; and (2) letting the j-th ROK output m_i^j for each $i \geq j+1$ as soon as the user accesses it (regardless of which input the user gives to the OTM). Thus by the time the user sees all shares of a committed graph, he has already accessed *all* the previous OTMs corresponding to the previous committed graphs. The user (information theoretically) does not know the i-th committed graph until he has accessed all the OTMs $1, \ldots, i-1$, and this forces "sequential" behavior that can be simulated. Of course, after accessing the boxes $1, \ldots, i-1$, the user can retrieve the committed graph and verify the proof's correctness as usual (i.e. completeness holds). See the full version for a formal theorem statement and proof.

We note that beyond being conceptually appealing, one-time proofs have obvious applications to identification and limited time (rather than revokable) credential proving applications. See the full version for formal definitions and further details.

6 Further Related Work

Using and Realizing Secure Hardware. Recently, Katz [Kat07], followed by Moran and Segev [MS08], studied the applications of secure hardware to constructing protocols that are secure in a concurrent setting. These works also model secure hardware as a "black box". Earlier work by Moran and Naor [MN05] showed how to construct cryptographic

protocols based on "tamper-evident seals", a weaker secure hardware assumption that models physical objects such as sealed envelopes, see the full version of this work for a more detailed comparison.
Works such as Ishai, Sahai and Wagner [ISW03], Gennaro *et al.* [GLM+04] and Ishai, Prabhakaran Sahai and Wagner [IPSW06], aim at achieving the notion of "black-box" access to devices using only minimal assumptions about hardware (e.g. adversaries can read some, but not all, of the hardware's wires etc.). The work of Micali and Reyzin [MR04] was also concerned with realizing ideal "black-box" access to computing devices, but they focused on obtaining model-independent reductions between specific physically secure primitives.

Alternative Approaches to Software Protection. An alternative software-based approach to software protection and obfuscation was recently suggested by Dvir, Herlihy and Shavit [DHS06]. They suggest protecting software by providing a user with an incomplete program. The user can run the program only by communicating with a server, and the server provides the "missing pieces" of the program in a protected manner. The setting of one-time programs is different, as we want to restrict even the *number of times* a user can run the program.

Count-Limiting using a TPM. In recent (independent) work, Sarmenta, van Dijk, O'Donnel, Rhodes and Devadas [SvDO+06] explore cryptographic and system security-oriented applications of real-world secure hardware, the Trusted Platform Module (TPM) chip (see [TPM07]). They show how a single TPM chip can be used to implement a large number of trusted monotonic counters and also consider applications of such counters to goals such as e-cash, DRM, and count limited cryptographic applications. These have to do with count-limiting special tasks, specific functionalities or small groups of functionalities, whereas we focus on the question of count-limiting access to general-purpose programs (and its applications). Our OTM construction indicates that TPMs can be used to count-limit *any* efficiently computable functionality. Following our work in ongoing work Sarmenta *et al.* began to consider using TPMs to count-limit general-purpose programs.

7 Acknowledgements

We thank Ran Canetti for early collaboration on this work, for his interest, suggestions and support. We are grateful to Nir Shavit for his enthusiastic and pivotal support of the one-time program concept, for suggesting applications to time-independent off-line E-tokens, and commenting on drafts of this work. Illuminating discussions on secure hardware were very helpful, for which we thank Anantha Chandrakasan, as well as Srini Devadas and Luis Sarmenta, who also explained to us their work on trusted monotonic counters using TPMs.
We would also like to thank Oded Goldreich for helpful discussions regarding software protection, Susan Hohenberger for her continuous clear and helpful answers to our questions about electronic cash, Adam Kalai for his invaluable suggestions, Moni Naor for his advice and illuminating

discussions, Adi Shamir for pointing out the importance of choosing the right order of operations on a ROM, and Salil Vadhan for his continuous help, insight and suggestions throughout various stages of this work. Finally, we are especially indebted to Silvio Micali's crucial comments following the first presentation of this work, which helped us focus on the resistance of OTMs to side channel attacks. Thank you Silvio!

References

[And01] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, January 2001.

[BBBW82] Charles H. Bennett, Gilles Brassard, Seth Breidbard, and Stephen Wiesner. Quantum cryptography, or unforgeable subway tokens. In *CRYPTO '82*, pages 267–275, 1982.

[Bes79] Robert M. Best. Us patent 4,168,396: Microprocessor for executing enciphered programs, 1979.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC '88*, pages 103–112, Chicago, Illinois, 1988.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01*, volume 2139, pages 1–18, 2001.

[Blu81] Manuel Blum, 1981. Personal communication.

[Blu87] M. Blum. How to prove a theorem so no-one else can claim it. In *Proceedings of ICML*, pages 1444–1451, 1987.

[BLV06] Boaz Barak, Yehuda Lindell, and Salil P. Vadhan. Lower bounds for non-black-box zero knowledge. *J. Comput. Syst. Sci.*, 72(2):321–391, 2006.

[Cha82] David Chaum. Blind signatures for untraceable payments. In *CRYPTO '82*, pages 199–203, 1982.

[Cha83] David Chaum. Blind signature systems. In *CRYPTO '83*, pages 153–156, 1983.

[DHS06] Ori Dvir, Maurice Herlihy, and Nir Shavit. Virtual leashing: Creating a computational foundation for software protection. *Journal of Parallel and Distributed Computing (Special Issue)*, 66(9):1233–1240, 2006.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[GK96] Oded Goldreich and Ariel Kahan. How to construct constant-round zero-knowledge proof systems for np. *J. Cryptology*, 9(3):167–190, 1996.

[GK05] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In Éva Tardos, editor, *FOCS '05*, pages 553–562. IEEE Computer Society, 2005.

[GLM+04] Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. In *TCC '04*, volume 2951, pages 258–277, 2004.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity, or all languages in np have zero-knowledge proof systems. *Journal of the ACM*, 38(1):691–729, 1991.

[GO96] Oded Goldreich and Rafi Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[IPSW06] Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and David Wagner. Private circuits ii: Keeping secrets in tamperable circuits. In *EUROCRYPT '06*, volume 4004, pages 308–327, 2006.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO '03*, volume 2729, pages 463–481, 2003.

[Kat07] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT '07*, volume 4515, pages 115–128. Springer, 2007.

[Ken80] Stephen T. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1980.

[MN05] Tal Moran and Moni Naor. Basing cryptographic protocols on tamper-evident seals. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP*, volume 3580, pages 285–297, 2005.

[MR04] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC '04*, volume 2951, pages 278–296, 2004.

[MS08] Tal Moran and Gil Segev. David and goliath commitments: Uc computation for asymmetric parties using tamper-proof hardware. In *EUROCRYPT '08*, pages 527–544, 2008.

[Pas03] Rafael Pass. On deniability in the common reference string and random oracle model. In *CRYPTO '03*, volume 2729, pages 316–337, 2003.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005.

[SMP88] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge with preprocessing. In *CRYPTO '88*, volume 403, pages 269–282, 1988.

[SvDO+06] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os (extended version). Technical Report 2006-064, MIT CSAIL Technical Report, 2006.

[TPM07] Trusted computing group trusted platform module (tpm) specifications, 2007.

[Yao86] Andrew C. Yao. How to generate and exchange secrets. In *FOCS '86*, pages 162–167, 1986.