Garbled Circuits for Leakage-Resilience: Hardware Implementation and Evaluation of One-Time Programs

(Full Version)*

Kimmo Järvinen[1], Vladimir Kolesnikov[2],
Ahmad-Reza Sadeghi[3], and Thomas Schneider[3]

[1] Dep. of Information and Comp. Science, Aalto University, Finland
kimmo.jarvinen@tkk.fi**
[2] Alcatel-Lucent Bell Laboratories, Murray Hill, NJ 07974, USA
kolesnikov@research.bell-labs.com
[3] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{ahmad.sadeghi,thomas.schneider}@trust.rub.de***

Abstract. The power of side-channel leakage attacks on cryptographic implementations is evident. Today's practical defenses are typically attack-specific countermeasures, such as changes to the underlying hardware or to compilers that generate code resilient to certain classes of side-channel attacks. The demand for a more general solution has given rise to the recent theoretical research that aims to build provably leakage-resilient cryptography. This direction is, however, very new and still largely lacks practitioners' evaluation with regard to both efficiency and practical security. A recent approach, One-Time Programs (OTPs), proposes using Yao's Garbled Circuit (GC) and very simple tamper-proof hardware to securely implement oblivious transfer, to guarantee leakage resilience.
Our main contributions are (i) a generic architecture for using GC/OTP modularly, and (ii) hardware implementation of GC/OTP evaluation and its efficiency analysis. We implemented two FPGA-based prototypes: a system-on-a-programmable-chip with access to hardware accelerator (suitable for smartcards and future smartphones), and a stand-alone hardware implementation (suitable for ASIC design). We chose AES as a representative complex function for implementation and measurements. As a result of this work, we are able to understand, evaluate and improve the practicality of employing OTPs as a leakage-resistance approach. Last, but not least, we believe that our work contributes to bringing together the results of both theoretical and practical communities.

Keywords: Garbled Circuit, Hardware Implementation, Leakage-Resilience, One-Time Programs, Secure Function Evaluation

* A short version of this paper appears in CHES 2010 [19].
** Supported by EU FP7 project CACE.
*** Supported by EU FP7 projects CACE and UNIQUE, and ECRYPT II.

1 Introduction

Side-channels and protection. For over a decade, we saw the power and elegance of side-channel attacks on a variety of cryptographic implementations and devices. These attacks refute the assumption of "black-box" execution of cryptographic algorithms, allow the adversary to obtain (unintended) internal state information, such as secret keys, and consequently cause catastrophic failures of the systems. Often the attacks are on the device in attacker's possession, and exploit physical channels such as observing power consumption [21], emitted radiation [8, 41, 1], and even the memory cache [20, 36, 34]. Moreover, even when no computation is performed, stored secrets may be leaked [45] or read out from RAM, which is typically not erased at power-off, allowing, e.g., cold-boot attacks [15]. Hence, from the hardware perspective, security has been viewed as more than the algorithmic soundness in the black-box execution model (see, e.g., [27, 58, 56, 47]).

Today's practical countermeasures typically address known vulnerabilities, and thus target not *all*, but specific classes of side-channel attacks. The desire for a complete solution motivated the recent burst of theoretical research in *leakage-resilient cryptography*, the area that aims to define security models and frameworks that capture leakage aspects of computation or/and memory. Information leakage is typically modeled by allowing the adversary learn (partial) memory or execution states. The exact information given to the adversary is specified by the (adversarily chosen) leakage function. Then, the assumption on the function (today, usually the bound on the output length) directly translates into physical assumption on the underlying device and the adversary. Proving security against such an adversary implies security in the real-world with the real hardware, subject to corresponding assumption (see [38] for a survey on this strand of research). We wish to add that some leakage assumptions and leakage-resilient constructions, although clearly stated, have not yet been evaluated by practitioners and side-channel community.[4] Further, efficiency is a major concern with today's solutions, since, e.g., embedded systems on an integrated circuit (IC) have very little cost tolerance.[5]

Secure Function Evaluation in hardware and leakage-resilience. Efficient Secure Function Evaluation (SFE) in an untrusted environment is a long-standing objective of modern cryptography. Informally, the goal of two-party SFE is to let two mutually mistrusting (polynomially-bounded) parties compute an *arbitrary* function on their private inputs without revealing any information about the inputs, beyond the output of the function. SFE has a variety of applications, particularly in settings with strong security and privacy demands. Deployment of SFE was very limited and believed expensive until recent improvements in algorithms, code generation, computing platforms and networks.

[4] Indeed, ongoing work of [48] investigates the practical applicability and usability of theoretical leakage models and the constructions proven secure therein.

[5] At the same time, e.g., the size of private circuits in [17] grows quadratically with the number of wire probes by the adversary.

As advocated in numerous prior works [30, 29, 25, 39, 24, 18, 23], *Garbled Circuit* (GC) [59] is often the most efficient (and thus viable) SFE technique in the two-party setting. As we argue in §3.2, the emerging fully homomorphic encryption schemes [10, 6, 46] are unlikely to approach the efficiency of GC.

Because of the execution flow of the GC solution (one party can non-interactively evaluate the function once the inputs have been fixed), the security guarantees of SFE are well-suited to prevent *all* side-channel leakage. Indeed, even GC evaluation in the open reveals no information other than the output. Clearly, it is safe to let the adversary see (as it turns out, even to modify) the entire execution process. The inputs-related stage of GC can also be made non-interactive with appropriate hardware such as Trusted Platform Modules (TPM) [14]. Goldwasser et al. [12] observed that very simple hardware is sufficient, one that, hopefully, can be manufactured tamper-resistant at low cost. They propose to use One-Time Programs (OTP), the (strengthened, see §3.1) combination of GC and above hardware, as a leakage-resilient computing device. The combination of non-interactive computation, count-limited execution, and leakage resistance holds great promise, e.g., for outsourcing computation and software business model. As as we explain below one of our goals in this paper is to evaluate today's performance of OTP in hardware.

Our objectives. We stress that practical efficiency of SFE and leakage-resilient computing is of utmost importance. Indeed, in most settings, the technology can only be adopted if its cost impact is acceptably low. In this work, we pursue the following two objectives.

First, we aim to mark this (practical efficiency) boundary by considering *hardware-accelerated* GC evaluation. In our implementation, we use state-of-the-art GC techniques, and optimize the code for embedded systems such as Systems on a Chip (SoC) based on FPGAs. Hash functions form the most significant computational burden in GC evaluation and throughout this paper we use SHA-256 as hash function. A cost-effective, straightforward and useful accelerator architecture is likely to implement SHA-256 functions in hardware, and thus will have similar cost structure to what we consider: low-cost SHA-256 evaluation and high-cost memory access. Implementing (at least some of) the SFE functionality in hardware promises to significantly improve computation speed and reduce power consumption. Our optimized hardware design and implementation allows us to evaluate costs, benefits and trade-offs of hardware support for SFE.

Second, we use our GC hardware-accelerator to implement OTP and evaluate its efficiency as a leakage-protection technique. As discussed in [12], OTPs have applications in one-time proofs, E-cash, or extreme software protection (with features such as limited number of executions or temporary transfer of cryptographic abilities). However, the exact circuit sizes of these functions, and hence the OTP practicability for these applications, are not yet clear. As a step in estimating these costs, we implemented OTP-based evaluation of the AES function. We chose AES as it is relatively complex and allows easy comparison with existing (heuristic) leakage-resilient protection mechanisms. We stress that OTP, and our implementation, are resilient against arbitrary side-channel attacks, based

on the OTP (relatively weak) hardware assumption. Further, as an application on its own, OTP evaluation of AES can be used for sending a small number of messages securely over a completely untrusted platform (e.g., a computer in an internet café) using a simple tamper-proof hardware token (e.g., a USB token) and the same key for encrypting/authenticating multiple messages.

1.1 Our Contributions and Outline

In line with our objectives stated above, we implement a variant of OTP with state-of-the-art GC optimizations discussed in §2.1. As an algorithmic contribution, we propose an efficiency improvement for OTPs with multiple outputs in §3.1. Further, we describe a generic architecture for using GC/OTP in a modular way to protect against arbitrary side-channel attacks in §3.2.

In our implementation, we present a hardware architecture (§4.1) and optimizations (§4.2) for efficient evaluation of GC/OTP on memory-constrained embedded systems. We measure the performance of GC/OTP evaluation of AES, a representative complex functionality, on our two FPGA-based prototype implementations in §4.3: a system-on-a-programmable-chip with access to a hardware accelerator for SHA-256 (representative for smartcards and future smartphones) and a stand-alone hardware implementation. Using our optimizations, secure evaluation of AES on our prototypes requires < 1.3 s and < 0.15 s, respectively. This shows that provable leakage-resilience via GC/OTP comes at a relatively high cost: an *unprotected* implementation of AES in hardware runs in 0.15 μs, and requires 2.6 times smaller chip area than OTP-based solution. (We note that the chip area for hardware-accelerated GC/OTP evaluation is independent of the evaluated function.) As AES is a representative complex function, we believe that our results, in particular our performance measurements, will serve as reference point for estimating GC/OTP runtimes of a variety of other functions (e.g., public key schemes).

1.2 Related Work

In this section we only briefly consider garbled circuits and one-time programs, and give more detailed explanations of them in §2.

Efficient implementations of Garbled Circuits (GC). We believe this is the first hardware implementation of garbled circuits (GC) and one-time programs (OTP) evaluation. While several implementations and measurements of GC exist in software already, e.g., [30, 29, 39], the hardware setting presents different challenges. Our work allows to compare the approaches and estimate the resulting performance gain (factor of 10-17). Hardware implementation of *generation* of GC in a cost-effective tamper-proof token with constant memory was shown in [18]. Our work is complementary, and our hardware accelerator can interoperate with the token of [18], as well as with software frameworks.

One-Time Programs (OTP). The combination of GC with non-interactive oblivious transfer in the semi-honest setting was proposed in [14]. In the malicious setting, OTP were introduced in [12] using minimal hardware assumptions.

Subsequently, [13] showed how to build non-interactive unconditionally secure computation from the hardware primitives proposed by [12]. We extend and implement OTPs in hardware. Our extension is in the computational model with Random Oracles (RO), and is more efficient than the constructions of [12, 13].

Protecting AES against side-channel attacks. We consider AES as reference implementation and summarize current techniques for protecting AES implementations. We stress that our implementation is provably leakage-free, assuming the security of OTP hardware blocks. However, this comes at a computational cost which we evaluate in this work.

A large amount of research has been done on countermeasures against side-channel attacks, e.g., power analysis attacks [21] requiring that power consumption does not depend on the underlying secrets. This can be achieved either by randomizing the power consumption or by making it constant [31]. Randomizing is done with masking, i.e., by adding random values. A variety of masking schemes for both algorithmic and circuit level have been proposed for AES, e.g., [2, 7]. Power consumption can be made constant by using gates whose power consumption is independent of input values, e.g., with dynamic differential (dual-rail) logic (see, e.g., [53, 16, 40, 43]). Countermeasures against power analysis have significant area overheads ranging from factor 1.5 to 5 [51]. Protecting implementations against other side-channel attacks or even fault attacks needs additional overhead. For instance, fault attack countermeasures require error detection techniques such as proposed in [44]. None of these countermeasures provides complete security. Indeed, countermeasures providing protection against simpler attacks have been shown to be useless against more powerful attacks, such as, template attacks [5, 35, 3] and higher-order differential power analysis [32, 11].

2 Preliminaries

In this section we describe the components and preliminaries underlying our constructions – garbled circuits (§2.1) and one-time programs (§2.2).

2.1 Garbled Circuits (GC)

Yao's Garbled Circuit (GC) approach [59] allows two parties, the sender $\mathcal{S}$ with private input y and the receiver $\mathcal{R}$ with private input x, to securely evaluate a boolean circuit C on their respective private inputs without revealing any other information than the result $z = C(x, y)$ of the evaluation, i.e., no intermediate values are revealed. We summarize the idea of Yao's GC protocol in the following(see Fig. 1 for an overview).

The circuit *constructor* $\mathcal{S}$ creates a *garbled circuit* $\widetilde{C}$ from the circuit C: for each wire W_i of C, he randomly chooses two garblings $\widetilde{w}_i^0, \widetilde{w}_i^1$, where $\widetilde{w}_i^j$ is the *garbled value* of W_i's value j. (Note: $\widetilde{w}_i^j$ does not reveal j.) Further, for each gate G_i, $\mathcal{S}$ creates a *garbled table* $\widetilde{T}_i$ with the following property: given a set of garbled values of G_i's inputs, $\widetilde{T}_i$ allows to recover the garbled value of the

corresponding G_i's output, but nothing else. $\mathcal{S}$ sends these garbled tables, called *garbled circuit* $\widetilde{C}$, to *evaluator* (receiver $\mathcal{R}$).

Additionally, $\mathcal{R}$ (obliviously) obtains the *garbled inputs* $\widetilde{w}_i$ corresponding to the inputs of both parties: the garbled inputs $\widetilde{y}$ corresponding to the inputs y of $\mathcal{S}$ are sent directly: $\widetilde{y}_i = \widetilde{y}_i^{y_i}$. For each of $\mathcal{R}$'s inputs x_i, both parties run a 1-out-of-2 Oblivious Transfer (OT) protocol (e.g., [33]), where $\mathcal{S}$ inputs $\widetilde{x}_i^0, \widetilde{x}_i^1$ and $\mathcal{R}$ inputs x_i. The OT protocol ensures that $\mathcal{R}$ receives only the garbled value corresponding to his input bit, i.e., $\widetilde{x}_i = \widetilde{x}_i^{x_i}$ while $\mathcal{S}$ learns nothing about x_i.

Now, $\mathcal{R}$ evaluates the garbled circuit $\widetilde{C}$ on the garbled inputs to obtain the *garbled output* $\widetilde{z}$ by evaluating $\widetilde{C}$ gate by gate, using the garbled tables $\widetilde{T}_i$. Finally, $\mathcal{R}$ determines the plain value z corresponding to the obtained garbled output value using an output translation table sent by $\mathcal{S}$.

Correctness of GC follows from the way garbled tables $\widetilde{T}_i$ are constructed. Yao's garbled circuit protocol is provably secure ([28]) when both parties are semi-honest (i.e., follow the protocol but may try to infer information about the other party's inputs from the messages seen). We stress that each GC can be evaluated only once, i.e. a new GC $\widetilde{C}$ must be used for each invocation.

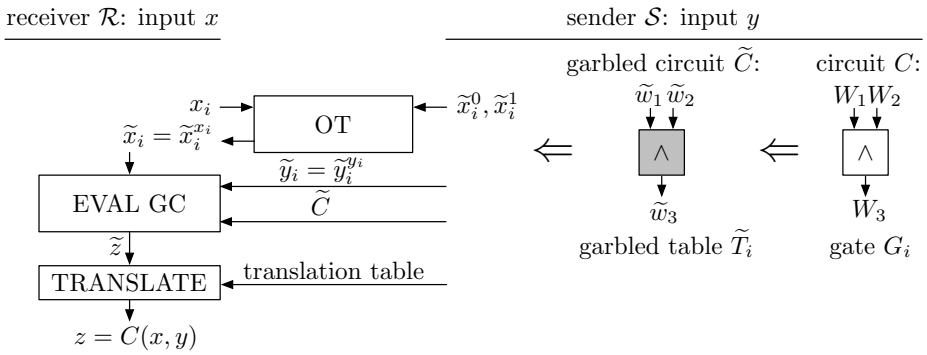


Fig. 1. Overview of Yao's Garbled Circuit Protocol (AND gate as example circuit)

Improved Garbled Circuits. We use the improved GC construction of [39], summarized next. Each garbled value $\widetilde{w}_i = \langle k_i, \pi_i \rangle$ consists of a t-bit key k_i and a permutation bit π_i, where t denotes the symmetric security parameter. XOR gates are evaluated "for free", i.e., no garbled table and negligible computation, by computing the bitwise XOR of their garbled values [25]. For each non-XOR gate with d inputs the garbled table $\widetilde{T}_i$ consists of $2^d - 1$ entries of size $t + 1$ bits each; the evaluation of a garbled non-XOR gate requires one invocation of SHA-256 [39]. At the high level, the keys k_i of the non-XOR gate's garbled inputs are used to obtain the corresponding garbled output value by decrypting the garbled table entry which is indexed by the input permutation bits π_i. We present the detailed description of the construction in Appendix §A.

2.2 Non-Interactive Garbled Circuits and One-Time Programs

The GC construction, although traditionally considered in the interactive setting, relies on interaction only as much as do the underlying OT executions. Consequently, (e.g., noted in [22]) the round complexity and (non-)interactivity features of the GC protocol are exactly those as the underlying OT.

Traditionally, for computational and storage efficiency, and because considered client-server applications permitted it, OT was considered in the interactive setting. In [14] the authors suggested to extend the Trusted Platform Module (TPM) [54] and use it as the hardware basis for non-interactive OT, resulting in a non-interactive version of Yao's protocol. Subsequently, One-Time Programs (OTP) were introduced in [12]. This approach considers malicious receivers and can be viewed simply as Yao's Garbled Circuit (GC), where the oblivious transfer (OT) function calls are implemented with One-Time Memory (OTM) tokens. An OTM token T_i is a simple tamper-proof hardware, which allows a single query of one of the two stored garbled values $\widetilde{x}_i^0, \widetilde{x}_i^1$ ([12] suggests using a tamper-proof one-time-settable bit b_i which is set, e.g. by burning a one-time fuse, as soon as the OTM is queried).[6] Further, OTM-based GC execution can be non-interactive, in the sense that the sender can send the GC and corresponding OTMs to the receiver, who will be able to execute one instance of SFE on any input of his choice.[7] Finally, GC (and hence also OTP) is inherently a one-time execution object (generalizable to k-time execution by repetition).

A subtle issue in this context, noted and addressed in [12], is the following. Previous GC-based solutions were either in the semi-honest model, or used interaction during protocol execution, which precluded the receiver $\mathcal{R}$ from choosing his input adaptively, based on given (and even partially evaluated) garbled circuit. This possibility of adaptive choice of inputs results in possible real attacks by $\mathcal{R}$ in the non-interactive setting.[8] The solution of [12] is to mask (each) output bit z_j of the function with a random bit m_j, equal to XOR of (additional) random bits $m_{i,j}$ contributed by *each* of the input OTMs T_i, i.e., $m_j = m_{1,j} \oplus m_{2,j} \oplus \ldots$ and $z'_j = z_j \oplus m_j$. This way, the real-world adversary does not learn the output of the function before he had queried all OTMs corresponding to his inputs, which

[6] Indeed, this is one of the simplest functionalities possible, and one that is hopefully easier to protect against leakage and tampering (we refer the reader to [12] for extended discussion on such protection).

[7] Further, as also noted in [13], the computed function can be fully hidden by evaluating a universal function instead. In practice, one would evaluate a garbled *Universal Circuit* that is programmed to compute the intended function. For a k-gate function, the universal circuit constructions of [55, 26, 42] result in an overhead of $\mathcal{O}(k \log k)$, $\mathcal{O}(k \log^2 k)$ and $\mathcal{O}(k^2)$ gates respectively with decreasing constant factors.

[8] From the mathematical perspective, the standard proof of security of GC now does not go through, since the simulator Sim would have to output to $\mathcal{R}$ the simulated garbled circuit (i.e., its garbled tables and output wire decoding) before knowing the inputs of the malicious receiver.

precludes him from adaptively selecting the input.[9] In §3 we present an efficiency improvement, and a generic architecture for leakage-resilient and tamper-proof computation derived from OTP.

3 Extending and Using One-Time Programs

In this section, we present in §3.1 a practical extension of the OTP construction of [12], which results in improved performance in case of multiple outputs. Additionally we make several observations about uses, security guarantees and applicability of OTP, and present a generic architecture for using OTPs for leakage-resilient computation in §3.2.

3.1 Extending One-Time Programs

As mentioned in the previous section, the solution in [12] seems to require each OTM token to additionally store a string of the size of the output. We propose a practical performance improvement to the technique proposed in [12], which is beneficial to SFE of functions with multi-bit output. In our solution each OTM token (additionally) stores a random string r_i of length of the security parameter t. Consequently, our construction results in smaller OTMs when the number of outputs is larger than the security parameter t. As a trade off, our security proof utilizes Random Oracles (RO), as we do not immediately see how to avoid their use and have OTM size independent of the number of outputs. (We discuss RO, its uses and security guarantees in Appendix §C).

Our main idea is to insert a "hold off" gate into each output wire W_j which can only be evaluated once *all* input OTMs had been queried, thus preventing $\mathcal{R}$ from choosing his input adaptively. It can be implemented by requiring a call to a hash function H (modeled as a Random Oracle) with inputs which include data from all OTMs. To implement this, we secret-share a random value $r \in_R \{0,1\}^t$ over all OTMs for the inputs. That is, OTM T_i additionally stores a share r_i (released to $\mathcal{R}$ with $\widetilde{x}_i$ upon the query), where $r = \bigoplus_i r_i$. Receiver $\mathcal{R}$ is able to recover r if and only if he queried all OTMs.

Fig. 2(b) depicts this contruction: Our version of OTM T_i, in addition to the two OT secrets $\widetilde{x}_i^0, \widetilde{x}_i^1$ and the tamper-proof bit b_i, contains a random share $r_i \in_R \{0,1\}^t$ which is released together with $\widetilde{x}_i^{x_i}$ once T_i is queried with input bit x_i. After querying all OTMs, the receiver $\mathcal{R}$ can recover $r = \bigoplus_i r_i$. The GC is constructed as usual (e.g., as described in §2.1), with the following exception. On each output wire W_j with garbled outputs $\widetilde{z}_j^0, \widetilde{z}_j^1$, we append a one-input, one-output OT-commit gate G_j, with no garbled table. We set the output wire secrets of G_j to $\widehat{z}_j^0 = H(\widetilde{z}_j^0||r), \widehat{z}_j^1 = H(\widetilde{z}_j^1||r)$. To enable $\mathcal{R}$ to compute the wire output non-interactively, GC also specifies that $\widehat{z}_j^b$ corresponds to b.

[9] In the proof, the new *Sim* is able to produce an indistinguishable simulation, since he only commits to the output values of the simulated GC when the last OTM is queried, the point at which *Sim* knows the inputs of the malicious receiver.

We note that a full formal construction is readily obtained from the above description. Also note that a malicious receiver is unable to complete the evaluation of any wire of GC until all the OTMs have been queried, and his input has been specified in full. Further, he is not able to lie about the result of the computation, since he can only compute one of the two values $\widetilde{z}_j^0, \widetilde{z}_j^1$. Demonstration of knowledge of $\widetilde{z}_j^i$ serves as a proof for the corresponding output value.

Theorem 1. *The above protocol is secure against a semi-honest sender $\mathcal{S}$, who generates the OTM tokens and the garbled circuit, and malicious receiver $\mathcal{R}$, in the Random Oracle model.*

Proof. The proof of Theorem 1 is given in Appendix §B. □

3.2 Using One-Time Programs for Leakage Protection

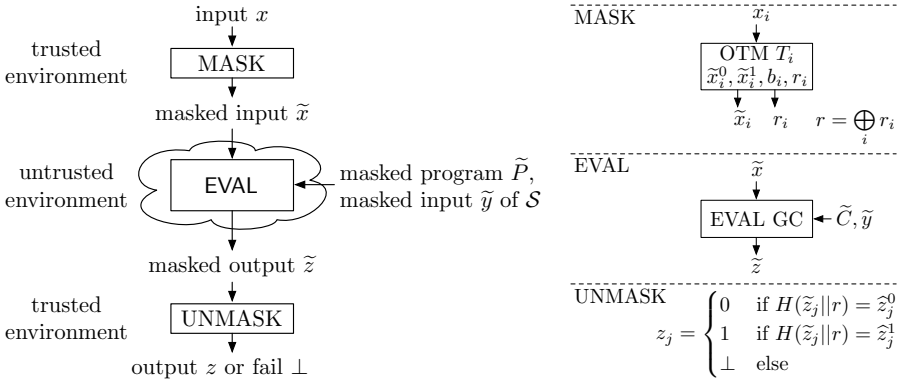
Most of today's countermeasures to side-channel attacks are specific to *known* attacks. Protecting hardware implementations (e.g., of AES) usually proceeds as follows (e.g., see [2]). First, the inputs are hidden, typically by applying a random mask (this requires trusted operation, and often the corresponding assumption is introduced). Afterwards, the computation is performed on the masked data. To allow this, the functionality needs to be adapted (e.g., using amended AES S-boxes). Finally, the mask is taken off to reveal the output of the computation.

We use a similar approach with similar assumptions (cf. Fig. 2(a)) to provably protect *arbitrary* functionalities against *all* attacks, both known and unknown:

1. The private data x provided by $\mathcal{R}$ is masked in a trusted environment MASK. The masked data $\widetilde{x}$ does not reveal any information about the private data, but still allows to compute on it.
2. The computation on the masked data is performed in an untrusted environment where the adversary is able to arbitrarily interfere (passively and actively) with the computation. To compute on the masked data, the evaluation algorithm EVAL needs a specially masked version of the program $\widetilde{P}$. Additional masked inputs $\widetilde{y}$ of $\mathcal{S}$ that are independent of $\mathcal{R}$'s inputs can be provided as well. The result of EVAL is the masked output $\widetilde{z}$.
3. Finally, $\widetilde{z}$ is unmasked into the plain output z. The procedure UNMASK allows to verify that $\widetilde{z}$ was computed correctly, i.e., no tampering happened in the EVAL phase in which case UNMASK outputs the failure symbol $\perp$. For correctness of this verification, UNMASK is executed in a trusted environment where the adversary can observe but not modify the computation.

More specifically, the masked program $\widetilde{P}$ is a garbled circuit $\widetilde{C}$, masked values $\widetilde{x}, \widetilde{y}, \widetilde{z}$ are garbled values and the algorithms MASK, EVAL and UNMASK can be implemented as described next.

MASK: Masking the input data x of receiver $\mathcal{R}$ is performed by mapping each bit x_i to its corresponding garbled value $\widetilde{x}_i$, i.e., to one of two garblings $\widetilde{x}_i^0, \widetilde{x}_i^1$. This can be provided externally (e.g., by interaction with a party on the



(a) Generic Architecture (b) Using One-Time Memory

Fig. 2. Evaluating a Functionality Without Leakage

network). We concentrate on on-board *non-interactive* masking which requires certain hardware assumptions (see below). The following can be directly used as a (non-interactive) MASK procedure:

- OTMs [12]: For small functionalities, we favor the very cheap One-Time Memory (OTM), as this seems to carry the weakest assumptions (cf. §2.2). However, as OTMs can be used only once, a fresh OTM must be provided for each input bit of the evaluated functionality. For practical applications, OTMs (together with their garbled circuits) could be implemented for example on tamper-proof USB tokens for easy distribution.
- Modified TPM [14]: Trusted Platform Modules (TPM) are low-cost tamper-proof cryptographic chips embedded in many of today's PCs [54]. TPM masking based on the non-interactive Oblivious Transfer (OT) protocol of [14] requires the (slightly extended) TPM to perform asymmetric cryptographic operations in form of a count-limited private key whose number of usages is restricted by the TPM chip. The TPM chip is extended to allow re-initialization for future non-interactive OTs with an interactive protocol instead of shipping new hardware.
- Smartcard: In our preferred solution for larger functionalities, masking could be performed by a tamper-proof smartcard. The smartcard would keep a secure monotonic counter to ensure a single query per input bit. Another advantage of this approach is that the same smartcard can be used to generate GC as well, thus eliminating GC transfer over the network as done in [18]. Further, the smartcard can be naturally used for multiple OTP evaluations.

For non-interactive masking, the hardware that masks the inputs must be trusted and the entire input must be specified before anything about the output z is revealed to prevent adaptive input selection as discussed in §2.2 and §3.1.

EVAL: The main technical contribution of this paper, the implementation of EVAL (of the masked program $\widetilde{P}$ on masked inputs $\widetilde{x}$ and $\widetilde{y}$) in embedded systems is presented in detail in §4. Here we note that $\widetilde{P}$ and $\widetilde{y}$ (masked input of $\mathcal{S}$) can be generated offline by the sender $\mathcal{S}$ and provided to EVAL by convenient means (e.g., via a data network or a storage medium). This is the scenario advocated in [12]; one of its advantages is that generation of $\widetilde{P}$ does not leak to EVAL. Alternatively, $\widetilde{P}$ and $\widetilde{y}$ could be generated "on-the-fly" using a cheap simple constant-memory stateless and tamper-proof token as shown in [18]. We reiterate that the masked program $\widetilde{P}$ can be evaluated exactly once.

UNMASK: Finally, the masked output $\widetilde{z}$ is checked for correctness and non-interactively decoded by $\mathcal{R}$ into the plain output z as follows (cf. §3.1 and Fig. 2(b)). For each output wire, the masked program $\widetilde{P}$ specifies the correspondence $\widehat{z}_j \rightarrow z_j$ in form of the two valid hash values $\widehat{z}_j^0$ and $\widehat{z}_j^1$. Even if EVAL is executed in a completely untrusted environment (e.g., processed on untrusted HW), its correct execution can be verified efficiently: when $H(\widetilde{z}_j||r)$ is neither $\widehat{z}_j^0$ nor $\widehat{z}_j^1$ the garbled output $\widetilde{z}_j$ is invalid and UNMASK outputs the failure symbol $\perp$. The reason for this verifiability property of GC is that a valid garbled output $\widetilde{z}_j$ can only be obtained by correctly evaluating the GC but cannot be guessed.

How far can we go with Homomorphic Encryption. At the first glance, the recently proposed Fully Homomorphic Encryption (FHE) [10, 6, 46] may seem as an attractive alternative solution for leakage-free computation. Indeed, FHE allows to compute arbitrary functions on encrypted data without the need for helper data in form of a masked program. The MASK algorithm would homomorphically encrypt the input x and the UNMASK algorithm could decrypt the result. Using verifiable computation [9], fully homomorphic encryption can also be extended to allow verification that the computation was performed correctly.

However, we argue that FHE is in fact not appropriate in our setting: Our first comment, which concerns any application of FHE, is that, in its state today, FHE is extremely computationally intensive. Although significant effort is underway in theoretical community to improve its performance, it seems unlikely that FHE would reach the efficiency of current public-key encryption schemes. Intuitively, this is because FHE must provide the same strong security guarantees, while, at the same time, possessing extra algebraic structure to allow for homomorphic operations. The extra structure weakens security, and countermeasures (costing performance) are necessary. Further, even assuming performance similar to that of RSA, this solution would be hundreds of times slower than GC-based solution, as symmetric primitives used in GC are orders of magnitude faster. Finally, from the leakage-resilience perspective, the UNMASK algorithm will be problematic, as it would need to perform complicated decryptions based on secret key. We would need to ensure nothing is leaked in these modules, which would bring us either to using much stronger assumptions or to a chicken-and-egg problem.

4 Efficient Evaluation of Garbled Circuits in Hardware

In this section we describe how GCs (and hence also OTPs) can be efficiently evaluated on embedded systems and memory-constrained devices. We first describe the HW architecture in §4.1. Then we present important compile-time optimizations and show their effectiveness in §4.2. Finally, we discuss technical details of our prototype implementation and timings in §4.3.

We stress that our designs and optimizations are generic. However, for concreteness and for meaningful comparison (e.g., with prior SW SFE of AES [39]), we take SFE of the AES function as our example for timings and other measurements. AES was chosen by [39] as a useful and representative function, with applications such as Oblivious Pseudo-Random Functions (OPRF), side-channel protection, blind MACs and encryption, and computation on encrypted data.

For AES evaluation, sender $\mathcal{S}$ provides AES key k as input y, and receiver $\mathcal{R}$ provides a plaintext block m as input x. $\mathcal{R}$ obtains the ciphertext c as output z, where $c = \text{AES}(k, m)$. Recall, during GC evaluation (EVAL), both key and message are masked (garbled) and hence cannot be leaked.

We use the [39] evaluation time of 2 seconds as a comparison baseline for our HW implementation.

4.1 Architecture for Evaluating Garbled Circuits in Hardware

We describe our architecture for efficient evaluation of GC on memory-constrained devices, i.e., having a small amount of slow memory only.

To minimize overhead, we choose key length $t = 127$; with a permutation bit, garbled values are thus 128 bits long (cf. §2.1). In the following we assume that memory cells and registers store 128 bit garbled values. This can be mapped to standard hardware architectures by using multiple elements in parallel.

Fig. 3 shows a conceptual high-level overview of our architecture described next. At the high-level, EVAL, the process of evaluating GC, on our architecture consists of the following steps (cf. §3.2). First, the garbled input values $\widetilde{x}, \widetilde{y}$ are stored in memory using the I/O interface. Then, GC gates are evaluated, using registers A, B, and C to cache the garbled inputs and outputs of a single garbled gate. Finally, garbled output value $\widetilde{z}$ is output over the I/O interface.

As memory access is expensive (cf. §4.3) we optimize code to re-use values already in registers. Our instructions are one-address, i.e., each instruction consists of an operator and up to one memory address. Each of our instructions has length 32 bits: 5 bits to encode one of 18 instructions (described next) and 27 bits to encode an address in memory.

LOAD/STORE: Registers can be loaded from memory using instructions LOAD_A and LOAD_B. Register C cannot be loaded as it will hold the output of evaluated non-XOR gates (see below). Values in registers can be stored back into memory using STORE_A, STORE_B, and STORE_C respectively.

XOR: We evaluate XOR gates [25] as follows. XOR_A `addr` computes $A \leftarrow A \oplus \text{mem[addr]}$. Similarly, the other one-operand XOR operations (XOR1) XOR_B

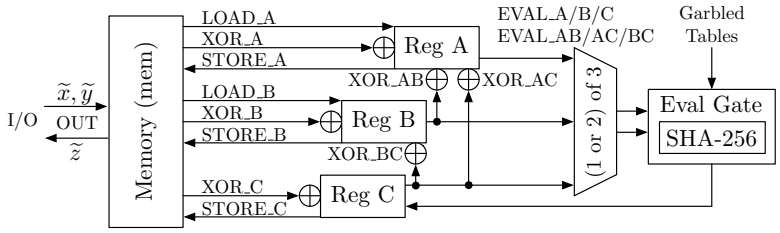


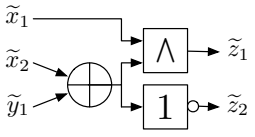
Fig. 3. Architecture for GC Evaluation (EVAL) on Memory-Constrained Devices

and XOR_C xor the value from memory with the value in the respective register. To compute XOR gates where both inputs are already in registers (XOR2), the instruction XOR_AB computes $A \leftarrow A \oplus B$. Similarly, XOR_AC computes $A \leftarrow A \oplus C$ and XOR_BC computes $B \leftarrow B \oplus C$.

EVAL: Non-XOR gates [39] are evaluated with the Eval Gate block that contains a hardware accelerator for SHA-256 (cf. §2.1 for details). The garbled inputs are in one (EVAL1) or two registers (EVAL2), and the result is stored in register C. The respective instructions for 1-input gates are EVAL_A, EVAL_B, EVAL_C and for 2-input gates EVAL_AB, EVAL_AC, EVAL_BC. The required garbled table entry is read from memory.

I/O: The garbled inputs of the circuit are always stored at the first $|x| + |y|$ memory addresses. The memory addresses in which the output values are stored are marked using final OUT instructions.

Example 1. Fig. 4 shows an example circuit and a possible sequence of instructions to evaluate it on our architecture. First, register A is loaded with $\widetilde{x}_1$ from memory address 0x0, then $\widetilde{x}_2 \oplus \widetilde{y}_1$ is computed in register B and the AND gate is evaluated to obtain output $\widetilde{z}_1$ which is stored at address 0x0 and overwrites $\widetilde{x}_1$, which is no longer needed. Then, the NOT gate is computed using register B as input and stored at address 0x1. The two outputs $\widetilde{z}_1, \widetilde{z}_2$ are at addresses 0x0 and 0x1.




```
LOAD_A   0x0   // A ← mem[0x0] = x̃1
LOAD_B   0x1   // B ← mem[0x1] = x̃2
XOR_B    0x2   // B ← B ⊕ mem[0x2] = x̃2 ⊕ ỹ1
EVAL_AB        // C ← A ∧ B
STORE_C  0x0   // mem[0x0] ← C
EVAL_B         // C ← not B
STORE_C  0x1   // mem[0x1] ← C
OUT      0x0   // z̃1 ← mem[0x0]
OUT      0x1   // z̃2 ← mem[0x1]
```

Fig. 4. Example Circuit (left) and Instruction Sequence to Evaluate its GC on our Architecture of Fig. 3 (right).

4.2 Compile-time Optimizations for Memory-Constrained Devices

In this section, we present several compile-time optimizations to improve performance of GC evaluation (EVAL) on our hardware architecture. We aim to reduce the size of GC (by minimizing the number of non-XOR gates), the size of the program (number of instructions), the number of memory accesses and memory size for storing intermediate garbled values. For concreteness, our presentation is built on the example of SFE of AES, but our techniques are generic.

Optimization a:PSSW09) Our baseline is the AES circuit/code of [39], already optimized for a small number of non-XOR gates. Their circuit consists of $11,286$ two-input non-XOR gates; thus, its GC has size $11,286 \cdot 3 \cdot 128$ bit $\approx$ 529 kByte. Without considering any caching strategies, this results in $113,054$ instructions, hence the program size is $113,054 \cdot 32$ bit $\approx$ 442 kByte, and the total amount of memory needed for EVAL is $34,136 \cdot 128$ bit $\approx$ 533 kByte.

We start with further reduction of the size of the garbled circuit.

Optimization b:NoXNOR) We reduce the GC size by replacing XNOR gates with XOR gates, and propagating the inverted output into the successor gates. Output XNOR gates are replaced with XOR and a 1-input inverter gate. The cost of this optimization is linear in the size of the circuit [37]. Overall, this optimization results in the elimination of $4,086$ XNOR gates and reduces the size of AES GC to $(7,200 \cdot 3 + 40) \cdot 128$ bit $\approx$ 338 kByte (improvement of 36%).

Remaining optimizations assume b:NoXNOR; optimizations d:MaxFanout, e:Rand use c:Cache.

Optimization c:Cache) We re-use values already in registers to reduce the number of LOADs. Values in registers are saved to memory only if needed later.

Optimization d:MaxFanout) We select a specific topologic order (traversing the circuit depth-first and following children in decreasing order of their fan-out).

Optimization e:Rand) We randomly consider several orders of evaluation, and select the most efficient one for EVAL. (This is a one-time compilation expense per function.) For present work, we considered several random topologic orders of the circuit, constructed by the traversal where the next gate is selected at random from the set of unprocessed gates with maximal fan-out. A more rigorous approach to this randomized technique can result in more substantial improvement, and is a good direction for future work.

Result. Using our optimizations we were able to substantially decrease the memory footprint of EVAL: As shown in Table 1 the smallest program was obtained with the non-deterministic optimization e:Rand which is only slightly better than our best deterministic optimization d:MaxFanout. The size of the AES program P is only $73,583 \cdot 32$ bit $\approx$ 287 kByte (improvement of 34.9% over a:PSSW09). The amount of intermediate memory is $17,315 \cdot 128$ bit $\approx$ 271 kByte (improvement of 49.3% over a:PSSW09) and the number of memory accesses (read and write) is reduced by $\approx 35\%$ compared to optimization a:PSSW09).

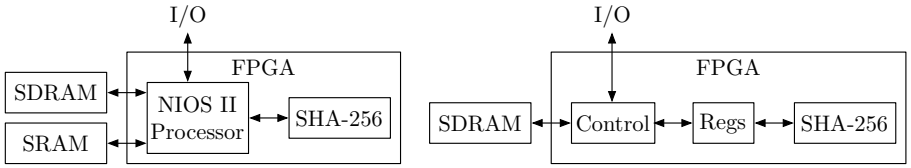
4.3 Prototype Implementation

We have designed two prototype implementations of the architecture of §4.1 – one for a System-on-a-Programmable-Chip with a hardware accelerator for SHA (re-

Table 1. Optimized AES Circuits (Sizes in kB)

	Garbled Circuit $\widetilde{C}$				Program P		Memory for GC Evaluation			
Optimization	non-XOR	1-input	XOR	Size	Instr.	Size	Read	Write	Entries	Size
a:PSSW09	11,286	0	22,594	529	113,054	442	67,760	33,880	34,136	533
b:NoXNOR	7,200	40	26,680	338	109,088	426	67,800	33,920	34,176	534
c:Cache	7,200	40	26,680	338	95,885	375	56,779	30,338	21,237	332
d:MaxFanout	7,200	40	26,680	338	74,052	289	42,469	23,767	18,676	292
e:Rand	7,200	40	26,680	338	73,583	287	42,853	22,650	17,315	271

flecting smartcard and future smartphone architectures) and another for a stand-alone unit (reflecting a custom-made GC accelerator in hardware). Both prototype implementations are evaluated on an Altera/Terasic DE1 FPGA board comprising an Altera Cyclone II EP2C20F484C7 FPGA and 512kB SRAM and 8MB SDRAM (and several other peripherals that are not relevant for this work) and are functionally equivalent: they take the same inputs (program P, garbled circuit $\widetilde{C}$, and garbled inputs $\widetilde{x}, \widetilde{y}$) and return the same garbled outputs $\widetilde{z}$; the only differences are the methods used in the implementation.



(a) System-on-a-Programmable-Chip (b) Stand-Alone Unit

Fig. 5. Architectures for Hardware-Assisted GC Evaluation

System-on-a-Programmable-Chip (SOPC). Our first prototype implementation is a system-on-a-programmable-chip (SOPC) that consists of a processor with access to a hardware accelerator for SHA-256, which speeds up the heaviest computational burden of the GC evaluation. This is a representative architecture for next generation smartphones or smartcards such as the STMicroelectronics ST33F1M smartcard which includes a 32-bit RISC processor, cryptographic peripherals, and memory comparable to our prototype system [50].

Our prototype implementation consists of a NIOS II/e 32-bit softcore RISC processor (the smallest variation of NIOS II), a custom-made SHA-256 unit, the SRAM, and the SDRAM. The entire process is run in the NIOS II processor which uses the SHA-256 unit for accelerating gate evaluations. The architecture is shown in Fig. 5(a). The SHA-256 unit is connected to the Avalon bus of the NIOS II as a peripheral component and it computes the hash for a 512-bit message in 66 clock cycles (excluding interfacing delays). The NIOS II program,

etc., are stored in the SRAM and the SDRAM is devoted solely for the data required to execute an OTP, i.e., the program for our architecture, the garbled circuit, the garbled inputs, the intermediate garbled values, etc.

Stand-Alone Unit. The second implementation is a stand-alone unit consisting of a custom-made control state machine, registers (A, B, C), a custom-made SHA-256 unit, and SDRAM. This architecture could be used to design an Application Specific IC (ASIC) as high-speed hardware accelerator for GC evaluation. Our prototype FPGA architecture is depicted in Fig. 5(b).

The interface (I/O in Fig. 5(b)) allows the host to write to and read from the SDRAM. First, the host writes the program, the garbled circuit, and the garbled inputs to SDRAM. The stand-alone unit then executes the program. The state machine parses the program and reads/writes data from/to SDRAM to/from the registers or evaluates the non-XOR gates using the SHA-256 unit according to the instructions (see §4.1 for details). The garbled outputs are written into specific addresses from which the host retrieves them using the I/O interface.

Prototyping Environment. The implementations were synthesized with Altera Quartus II, version 9.1 (2009). The custom-made units were written with VHDL and verified with simulations in ModelSim Altera-edition, version 6.5b (2009). The NIOS II processor was programmed with C using NIOS II IDE, version 9.1 (2009). All parts of both implementations run with a 50 MHz clock. The interfacing with the host was implemented with NIOS II also for the stand-alone unit. In both implementations, data was transferred to the FPGA by using the *Host File System* (HFS) of NIOS II; we point out that HFS is feasible for prototyping phase only, and the interface should be replaced with a more appropriate one (e.g., PCI-Express or Gigabit Ethernet) in a real application.

Area. The area requirements of our prototype implementations are shown in Table 2. Both prototypes easily fit into the low-cost Cyclone II FPGA which contains 18,754 logic cells (LC), each containing one 4-to-1-bit look-up table (LUT) and a flip-flop (FF), and 52 4092-bit embedded memory blocks (M4K). The values of the stand-alone unit exclude NIOS II used for the HFS in the prototype. SHA-256 is the largest and most significant block in both prototypes. The SOPC additionally contains NIOS II, on-chip memory, and SDRAM controller, and the stand-alone unit contains additional control logic, registers, and SDRAM controller (cf. Fig. 5).

Table 2 also shows the area for a straightforward iterative implementation of AES-128 on the same FPGA to ease cost evaluation of our methodology; however, this implementation does not include any countermeasures against side-channel attacks. Compared to an unprotected implementation, countermeasures against power analysis have area overheads ranging from factor of 1.5 to 5 [51] as discussed in §1.2; therefore, the area overheads of OTP evaluation are comparable with other side-channel countermeasures.

Timings. *Instructions.* The timings of instructions are summarized in Table 3. They show the average number of clock cycles required to execute an instruction excluding the latency of fetching the instruction. Gate evaluations are expensive in the SOPC implementation, although the SHA-256 computations

Table 2. Areas of the Prototypes for GC Evaluation on an Altera Cyclone II FPGA

Design	LC	FF	M4K
SOPC	7501	4364	22
NIOS II	1104	493	4
SHA-256	2918	2300	8
Stand-Alone Unit	6252	3274	8
SHA-256	3161	2300	8
AES (unprotected)	2418	431	0

Table 3. Timings for Instructions on Prototypes (clock cycles, average)

Instruction	*SOPC*	*Stand-Alone Unit*
LOAD	291.43	87.63
XOR1	395.30	87.65
XOR2	252.00	1.00
STORE	242.00	27.15
EVAL1	1,282.30	109.95
EVAL2	1,491.68	135.05
OUT	581.48	135.09

are fast, because they involve a lot of data movement (to/from the SHA-256 unit and from the SDRAM) which is expensive. The dominating role of memory reads and writes is clear in the timings of the stand-alone implementation: the only instructions that do not require memory operations (XOR2) require only a single clock cycle and EVAL1 are faster than EVAL2 because they access the memory on average every other time (no access if the permutation bit is zero) compared to three times out of four (no access if both permutation bits are zeros).

AES. The timings to run the optimized garbled circuits for the AES functionality of §4.2 on our prototype implementations are given in Table 4. These timings are for GC evaluation only; i.e, they neglect the time for transferring data to/from the system because interface timings are highly technology dependent (HFS is extremely slow, but convenient for prototyping). The SHA-256

Table 4. Timings for the FPGA-based Prototypes for GC Evaluation

	System-on-a-Programmable-Chip				*Stand-Alone Unit*			
	Clock cycles		Timings (ms)		Clock cycles		Timings (ms)	
Optimization	SHA	Total	SHA	Total	SHA	Total	SHA	Total
a:PSSW09	744,876	94,675,402	14.898	1,893.508	744,876	11,235,118	14.898	224,702
b:NoXNOR	477,840	87,433,180	9.557	1,748.664	477,840	10,604,268	9.557	212.085
c:Cache	477,840	77,991,519	9.557	1,559.830	477,840	9,208,586	9.557	184,172
d:MaxFanout	477,840	62,929,278	9.557	1,258.586	477,840	7,203,630	9.557	144.073
e:Rand	477,840	62,629,261	9.557	1,252.585	477,840	7,201,150	9.557	144.023

computations take an equal amount of time for both implementations because the SHA-256 unit is the same. The (major) difference in timings is caused by data movement, XORs, interface to the SHA-256 unit, etc. The runtimes for both implementations are dominated by writing and reading the SDRAM; e.g., 84.3% for the stand-alone unit and our smallest AES circuit (optimization e:Rand). Hence, improving the speed of the memory, e.g., by introducing burst reads and writes, is the key for further speedups.

Performance Comparison. A software implementation that evaluates the GC/OTP of the unoptimized AES functionality a:PSSW09 required 2 seconds

on an Intel Core 2 Duo 3.0 GHz with 4GB of RAM [39]. Our optimized circuit e:Rand evaluated on the stand-alone unit requires only 144 ms for the same operation and, therefore, provides a speedup by a factor of 10.4–17.4 (taking the lack of precision into account).

On the other hand, the unprotected AES implementation listed in Table 2 encrypts a message block in 10 clock cycles and runs on a maximum clock frequency of 66 MHz resulting in a timing of 0.1515 μs; hence, the GC/OTP evaluation suffers from a timing overhead factor of $\approx 950,000$. For comparison, the timing overhead of one specific implementation including power analysis countermeasures was factor of 3.88 [52].

5 Conclusion

The power of side-channels attacks on cryptographic implementations have motivated both theoreticians and practitioners to seek more general defense models and solutions. The recent strand of research on leakage-resilient cryptography is still in its early stage, and most proposals are not yet implemented and evaluated in practice. The recent observation of one-time programs (OTPs) uses established Garbled Circuit (GC) techniques, and relies on hardware with relatively weak tamper-proof assumptions, to realize provably leakage-resilient evaluation of arbitrary functions in an untrusted environment.

In this work, we studied the performance of GC/OTP evaluation in hardware. For this, we implemented two hardware prototypes for GC/OTP evaluation based on FPGA: one for a system-on-a-programmable-chip with access to a hardware accelerator for SHA-256 (representative for smartcards and future smartphones), and a stand-alone hardware implementation (reflecting a custom-made GC accelerator in hardware). We chose AES as the representative complex function, and we believe our measurements will serve as a reference point for estimating runtimes of a variety of useful functions.

Our measurements show an order of magnitude performance improvement over the previous software implementation of AES GC evaluation reported in [39]. We thus believe that GC/OTP is a viable option for hardware implementation, especially for secure computation scenarios, where hardware acceleration is desired. The use of GC/OTP for leakage resilience carries high cost, and should be reserved for truly security-critical applications. Finally, OTP can be inherently evaluated only once, i.e., each additional evaluation requires fresh OTM hardware modules and data transfer for the GC (338 kByte for AES), which prevents certain deployment scenarios.

Acknowledgements. Thanks to anonymous reviewers of CHES'10 for their helpful comments and to co-authors of [39] for the initial AES circuit.

References

1. D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems (CHES'02)*, volume 2523 of *LNCS*, pages 29–45. Springer, 2002.
2. M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 309–318. Springer, 2001.
3. C. Archambeau, E. Peeters, F.-X. Standaert, and J.-J. Quisquater. Template attacks in principal subspaces. In *Cryptographic Hardware and Embedded Systems (CHES'06)*, volume 4249 of *LNCS*, pages 1–14. Springer, 2006.
4. R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.
5. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems (CHES'02)*, volume 2523 of *LNCS*, pages 13–28. Springer, 2003.
6. M. v. Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2009/616, 2009. `http://eprint.iacr.org`. To appear at EUROCRYPT 2010.
7. W. Fischer and B. M. Gammel. Masking at gate level in the presence of glitches. In *Cryptographic Hardware and Embedded Systems (CHES'05)*, volume 3659 of *LNCS*, pages 187–200. Springer, 2005.
8. K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001.
9. R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. Cryptology ePrint Archive, Report 2009/547, 2009. `http://eprint.iacr.org`.
10. C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC'09)*, pages 169–178. ACM, 2009.
11. B. Gierlichs, L. Batina, B. Preneel, and I. Verbauwhede. Revisiting higher-order DPA attacks: Multivariate mutual information analysis. In *Cryptographers' Track at RSA Conference (CT-RSA '10)*, volume 5985 of *LNCS*, pages 221–234. Springer, 2010.
12. S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *LNCS*, pages 39–56. Springer, 2008.
13. V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In *Theory of Cryptography (TCC'10)*, volume 5978 of *LNCS*, pages 308–326. Springer, 2010.
14. V. Gunupudi and S. Tate. Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In *Financial Cryptography and Data Security (FC'08)*, volume 5143 of *LNCS*, pages 98–112. Springer, 2008.
15. J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium (Security'08)*, pages 45–60. USENIX Association, 2008.
16. D. Hwang, K. Tiri, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. AES-based security coprocessor IC in 0.18-μm CMOS with resistance to differential power analysis side-channel attacks. *IEEE Journal of Solid-State Circuits*, 41(4):781–791, 2006.

17. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.
18. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Embedded SFE: Offloading server and network using hardware tokens. In *Financial Cryptography and Data Security (FC'10)*, LNCS. Springer, 2010. To appear.
19. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *12th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'10)*, LNCS. Springer, 2010. To appear.
20. J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In *European Sumposium on Research in Computer Security (ESORICS '98)*, volume 1485 of *LNCS*, pages 97–110. Springer, 1998.
21. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *LNCS*, pages 388–397, 1999.
22. V. Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In *ASIACRYPT'05*, volume 3788 of *LNCS*, pages 136–155. Springer, 2005.
23. V. Kolesnikov. Truly efficient string oblivious transfer using resettable tamper-proof tokens. In *Theory of Cryptography (TCC'10)*, volume 5978 of *LNCS*, pages 327–342. Springer, 2010.
24. V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.
25. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.
26. V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security (FC'08)*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008.
27. K. Lemke. Embedded security: Physical protection against tampering attacks. In C. Paar K. Lemke and M. Wolf, editors, *Embedded Security in Cars*, chapter 2, pages 207–217. Springer, 2006.
28. Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009. Cryptology ePrint Archive, Report 2004/175, `http://eprint.iacr.org`.
29. Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *Security and Cryptography for Networks (SCN'08)*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.
30. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium (Security'04)*. USENIX Association, 2004.
31. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
32. T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In *Cryptographic Hardware and Embedded Systems (CHES'00)*, volume 1965 of *LNCS*, pages 238–251. Springer, 2000.
33. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

34. D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at RSA Conference (CT-RSA'06)*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.
35. E. Oswald and S. Mangard. Template attacks on masking—resistance is futile. In *Cryptographers' Track at RSA Conference (CT-RSA '07)*, volume 4377 of *LNCS*, pages 243–256. Springer, 2007.
36. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical Report CSTR-02-003, University of Bristol, 2002.
37. A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 89–106. Springer, 2009.
38. K. Pietrzak. Provable security for physical cryptography. In *Western European Workshop on Research in Cryptology (WEWORC'09)*, 2009. Survey available at `http://homepages.cwi.nl/~pietrzak/publications/Pie09b.pdf`.
39. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
40. T. Popp and S. Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing contraints. In *Cryptographic Hardware and Embedded Systems (CHES'05)*, volume 3659 of *LNCS*, pages 172–186. Springer, 2005.
41. J.-J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and countermeasures for smart cards. In *Research in Smart Cards (E-smart 2001)*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001.
42. A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *International Conference on Information Security and Cryptology (ICISC'08)*, volume 5461 of *LNCS*, pages 336–353. Springer, 2008.
43. M. Saeki, D. Suzuki, K. Shimizu, and A. Satoh. A design methodology for a DPA-resistant cryptographic LSI with RSL techniques. In *Cryptographic Hardware and Embedded Systems (CHES'09)*, volume 5747 of *LNCS*, pages 189–204. Springer, 2009.
44. A. Satoh, T. Sugawara, N. Homma, and T. Aoki. High-performance concurrent error detection scheme for AES hardware. In *Cryptographic Hardware and Embedded Systems (CHES'08)*, volume 5154 of *LNCS*, pages 100–112. Springer, 2008.
45. S. P. Skorobogatov. Data remanence in flash memory devices. In *Cryptographic Hardware and Embedded Systems (CHES'05)*, volume 3659 of *LNCS*, pages 339–353. Springer, 2005.
46. N.P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography (PKC'10)*, LNCS. Springer, 2010. Cryptology ePrint Archive, Report 2009/571, `http://eprint.iacr.org`.
47. S. W. Smith. Fairy dust, secrets, and the real world. *IEEE Security & Privacy*, 1(1):89–93, 2003.
48. F.-X. Standaert, O. Pereira, Y. Yu, J.-J. Quisquater, M. Yung, and E. Oswald. Leakage resilient cryptography in practice. Cryptology ePrint Archive, Report 2009/341, 2009. `http://eprint.iacr.org/`.
49. M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. d. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, 2009.

50. STMicroelectronics. Smartcard MCU with 32-bit ARM SecurCore SC300 CPU and 1.25 Mbytes high-density Flash memory. Data brief, October 2008. http://www.st.com/stonline/products/literature/bd/15066/st33f1m.pdf.
51. K. Tiri. Side-channel attack pitfalls. In *Design Automation Conference (DAC'07)*, pages 15–20. ACM, 2007.
52. K. Tiri, D. Hwang, A. Hodjat, B.-C. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. Prototype IC with WDDL and differential routing — DPA resistance assessment. In *Cryptographic Hardware and Embedded Systems (CHES '05)*, volume 3659 of *LNCS*, pages 354–365. Springer, 2005.
53. K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *Design, Automation and Test in Europe (DATE'04)*, volume 1, pages 246–251. IEEE, 2004.
54. Trusted Computing Group (TCG). TPM main specification. Main specification, Trusted Computing Group, May 2009. http://www.trustedcomputinggroup.org.
55. L. G. Valiant. Universal circuits (preliminary report). In *ACM Symposium on Theory of Computing (STOC'76)*, pages 196–203. ACM, 1976.
56. I. Verbauwhede and P. Schaumont. Design methods for security and trust. In *Design, Automation and Test in Europe (DATE'07)*, pages 672–677. ACM, 2007.
57. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
58. S. H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defences. In *Cryptographic Hardware and Embedded Systems (CHES'00)*, volume 1965 of *LNCS*, pages 302–317. Springer, 2000.
59. A. C. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

A Improved Garbled Circuits

Yao's original GC construction has been improved by reducing its computation and communication complexity as described next. As throughout the entire paper we denote symmetric security parameter with t.

Free XOR gates. An efficient method for creating garbled circuits which allows "free" evaluation of XOR gates was presented in [25]. More specifically, a garbled XOR gate has no garbled table (*no communication*) and its evaluation consists of XORing its garbled input values (*negligible computation*). The main observation of [25] is, that the constructor $\mathcal{S}$ randomly chooses a global key difference $\Delta \in_R \{0,1\}^t$ which remains unknown to evaluator $\mathcal{R}$ and relates the garbled values as $\widetilde{w}_i^0 = \widetilde{w}_i^1 \oplus (\Delta||1)$. The usage of such garbled values allows for *free evaluation of XOR gates* with input wires W_1, W_2 and output wire W_3 by computing $\widetilde{w}_3 = \widetilde{w}_1 \oplus \widetilde{w}_2$ (no communication and negligible computation).

Reduced non-XOR gates. Non-XOR gates, can be evaluated as in Yao's GC construction [59] with a *point-and-permute technique* (as used in [30]): The garbled values $\widetilde{w}_i = \langle k_i, \pi_i \rangle \in \{0,1\}^{t+1}$ consist of a symmetric key $k_i \in \{0,1\}^t$ and a random permutation bit $\pi_i \in \{0,1\}$. The entries of the garbled table are permuted such that the permutation bits π_i of a gate's garbled input wires can be used as index into the garbled table to directly point to the entry to be decrypted. After decrypting this entry using the garbled input wires' t-bit keys k_i, evaluator

obtains the garbled output value of the gate. The encryption is done with the symmetric encryption function $\mathsf{Enc}^s_{k_1,\ldots,k_d}(m)$, where d is the number of inputs of the gate and s is a unique identifier used once, e.g., a monotonic gate counter. Enc can be instantiated with $m \oplus \mathsf{H}(k_1||\ldots||k_d||s)$, where H is a Random Oracle (RO) which can be instantiated with a suitably chosen cryptographic hash function such as SHA-256 in practice. We note that the RO assumption can be avoided or weakened at small additional computation cost – see [39]. Additionally, garbled row reduction of [39] allows to remove the first entry from the garbled tables of non-XOR gates, i.e., the garbled table of a d-input non-XOR gate consists of $2^d - 1$ table entries of size $t + 1$ bits each.

B Proof of Theorem 1.

Proof. (sketch) Security against semi-honest $\mathcal{S}$ is trivial as $\mathcal{S}$ does not see $\mathcal{R}$'s input (we consider OTMs a separate entity from $\mathcal{S}$).

We now describe the simulator Sim which will produce a view indistinguishable from the view of $\mathcal{R}$ in real execution. Sim will query the receiver $\mathcal{R}$ as a black box and answer all of $\mathcal{R}$'s queries, including calls to (simulated) RO $\mathcal{O}$. Our proof is based on the idea that Sim will "program" $\mathcal{O}$ such that the output of the "hold off" gates will match the output given by the trusted party of the ideal game.

Without loss of generality, we assume that $\mathcal{R}$ queries RO only once for each distinct input. Upon initialization, Sim constructs GC, as would an honest $\mathcal{S}$ in the construction described above, and sends the GC to $\mathcal{R}$ together with randomly chosen commitments $\widehat{z}^0_j, \widehat{z}^1_j$ for all output wires. Additionally, Sim generates a random key r and a random secret sharing $r = \bigoplus_i r_i$ of it. For the wires corresponding to the input of $\mathcal{S}$, Sim sends secrets corresponding to 0-values. Whenever $\mathcal{R}$ queries the i-th OTM with input bit x_i, Sim responds with the corresponding garbled value $\widetilde{x}^{x_i}_i$, constructed earlier as part of GC construction, and the share r_i. Once $\mathcal{R}$ had queried the final OTM, Sim sends the input received from $\mathcal{R}$ to the trusted party, and receives the output $f(x, y)$ of the computation. Now Sim "programs" $\mathcal{O}$ to output certain values according to the received $f(x, y)$. That is, on input $(\widetilde{z}_j||r)$ (call associated with OT-commit gate G_i and the j-th bit of the output), $\mathcal{O}$ will output $\widehat{z}^{f_j(x,y)}_j$, i.e., the commitment for the wire leaving G_j that corresponds to the bit $f_j(x, y)$ of the output he received from the trusted party.

It is not hard to see that the above simulator generates view indistinguishable from the view of $\mathcal{R}$ in real execution. First, we note that the simulated GC and responses to RO and OTM queries are indistinguishable from real execution. Thus, in particular, $\mathcal{R}$ "behaves normally" during the simulation, and would not be able to, e.g., substitute inputs in a special way, etc. Further, "programming" of $\mathcal{O}$ will not be noticed by $\mathcal{R}$, since he can query programmed values only with negligible probability prior to completing all OTM calls (since r is random and unknown to $\mathcal{R}$ prior to completing all OTM calls). □

C On our use of Random-Oracle

We note that in our extension of one-time programs described in §3.1, we use a relatively strong assumption of programmable RO. In fact, it had been shown [4] that some (contrived) uses of RO cannot be securely instantiated with *any* hash function. Therefore, proofs in the RO model cannot be seen as proofs in the strictest mathematical sense. However, we believe that modeling cryptographic hash functions, such as SHA-256, as RO is well-justified in our setting.

Firstly, to date, no attacks exploiting the RO assumption are known on practical systems. This holds true even in academic context[10]. Further, even in well-understood and deployed real-life systems, the crypto core (which includes the employed hash functions) is almost never targeted for attacks, in favor of *much* easier to exploit implementation flaws. In our setting, we deal with much less understood physical leakage, and make strong assumptions on the amount and content of leakage. We believe that exploiting the structure of real hash function (required to violate the RO assumption), something that eluded cryptographers for decades, is far harder and costlier than violation of other assumptions used in design of leakage-resilient systems.

In sum, we strongly believe that making the RO assumption on the employed hash function is practically justified in ours and many other settings.

[10] Important attacks on SHA-1 [57] that exploit the structure of the functions were far impractical, and simply accelerated migration to stronger primitives, which are believed secure today. While some attacks, such as the attack on MD5 [49], are in fact practical, the use of MD5 had long been considered unsafe, and [49] broke poorly managed systems. Thus we do not consider [49] an attack on properly implemented protocols. In fact, [49], and the works that lead to it only support the historic fact that users of hash functions do receive weakness warnings years ahead of possible real breaks.