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

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 against 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 and efficiency analysis of GC/OTP evaluation. We implemented two FPGA-based prototypes: a system-on-a-programmable-chip with access to hardware crypto 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 GC/OTP as a leakage-resistance approach.

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

* This is a short version of the paper. The full version is available [7].
** Supported by EU FP7 project CACE.
*** Supported by EU FP7 projects CACE and UNIQUE, and ECRYPT II.

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 side-channels such as power consumption or emitted radiation. Hence, from the hardware perspective, security has been viewed as more than the algorithmic soundness in the black-box execution model (see, e.g., [28]).

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 a physical assumption on the underlying device and the adversary. Proving security against such an adversary implies security in the real-world with the real device, subject to corresponding assumption (see [17] for a survey on this strand of research). We note that many of the results of this new line of research (i.e., leakage assumptions and leakage-resilient constructions), although clearly stated, have not yet been evaluated by practitioners and side-channel community.[4]

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.

As advocated in numerous prior works [13, 10, 18, 8], *Garbled Circuit* (GC) [29] is often the most efficient (and thus viable) SFE technique in the two-party setting. As we argue in the full version [7], the emerging fully homomorphic encryption schemes [3] 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 GC evaluation process. The inputs-related stage of GC can also be made non-interactive with appropriate hardware such as Trusted Platform Modules (TPM) [6]. Goldwasser et al. [4] observed that very simple hardware is sufficient, one that, hopefully, can be manufactured tamper-resistant at low cost.

[4] Indeed, ongoing work of [21] investigates the practical applicability and usability of theoretical leakage models and the constructions proven secure therein.

They propose to use One-Time Programs (OTP), a combination of GC and above hardware, for leakage-resilient computation. Indeed, one of our goals is to evaluate today's performance of OTP in hardware.

Our objectives. Practical efficiency of SFE and leakage-resilient computing is critical. 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* state-of-the-art GC evaluation, and optimizing it for embedded systems. Implementing SFE (at least partially) in hardware promises to significantly improve computation speed and reduce power consumption. We evaluate costs, benefits and trade-offs of hardware support for GC evaluation.

Second, we use our GC hardware-accelerator to implement OTP and evaluate the efficiency of this provably leakage-resilient protection. The envisioned applications for OTPs mentioned in [4] are complex functionalities such as one-time proofs, E-cash, or extreme software protection. We make a first step towards estimating the costs of such complex OTP applications by implementing OTP evaluation of the AES function. We chose AES as it is relatively complex and allows comparison with existing (heuristic) leakage protection.

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. 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 performance of GC/OTP evaluation of AES on our two FPGA-based implementations in §4.3: a system-on-a-programmable-chip with access to SHA-256 hardware accelerator (representative for smartcards and future smartphones) and a stand-alone hardware implementation. With optimization, GC evaluation of AES on our implementations requires about 1.3 s and 0.15 s, respectively. This shows that *provable leakage-resilience* via GC/OTP comes at a relatively high cost (but its use might still be justified in high-security applications): an *unprotected* implementation of AES in hardware runs in 0.15 μs, and requires 2.6 times smaller 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.

1.2 Related Work

Efficient implementations of Garbled Circuits (GC). We believe that our results are the first hardware implementation of garbled circuits (GC) and

one-time programs (OTP) evaluation. While several implementations and measurements of GC exist in software, e.g., [13, 18], the hardware setting presents different challenges. Our work allows to compare the approaches and estimate the resulting performance gain (our hardware implementation is faster than the software implementation of [18] by a factor of 10-17). Hardware implementation of GC *generation* in a cost-effective tamper-proof token with constant memory was shown in [8]. Our work is complementary, and our hardware accelerator for GC evaluation can be combined with the token of [8], or software frameworks.

One-Time Programs (OTP). The combination of GC with non-interactive oblivious transfer in the semi-honest setting was proposed in [6]. For malicious evaluator, OTP were introduced in [4] using minimal hardware assumptions. Subsequently, [5] showed how to build non-interactive secure computation unconditionally secure against malicious parties using tamper-proof hardware tokens. We extend and implement OTPs in hardware. Our extension is in the computational model with Random Oracles (RO), secure against malicious evaluator, and more efficient than the constructions of [4, 5].

Protecting AES against side-channel attacks. We summarize current techniques for leakage-protecting AES. We stress that our implementation is provably leakage-free, but 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., to protect against power analysis attacks [9], the power consumption needs to be made independent of the underlying secrets by either randomizing the power consumption or making it constant [14]. 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., [1]. For constant power consumption one can use gates whose power consumption is independent of input values, e.g., with dynamic differential (dual-rail) logic (see, e.g., [25]). Countermeasures against power analysis have significant area overheads ranging from factor 1.5 to 5 [23]. 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 [20]. 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 [2] and higher-order differential power analysis [15].

2 Preliminaries

Garbled Circuits (GC). Yao's GC approach [29] allows sender $\mathcal{S}$ with private input y and receiver $\mathcal{R}$ with private input x, to securely evaluate a boolean circuit C on (x, y) without revealing any information other than the result $z = C(x, y)$ of the evaluation. We summarize the idea of Yao's GC protocol next.

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., [16]), 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 $\mathcal{S}$.

Yao's garbled circuit protocol is provably secure ([12]) 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.

Improved Garbled Circuits. We use the improved GC construction of [18], 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 [10]. 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 [18]. 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 the full version [7].

Non-Interactive Garbled Circuits and One-Time Programs. The round complexity of Yao's GC protocol is exactly that of the underlying OT protocol. In [6] the authors suggested to extend the Trusted Platform Module (TPM) [26] for implementing non-interactive OT, resulting in a non-interactive version of Yao's protocol. Subsequently, One-Time Programs (OTP) were introduced in [4]. 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 $\widetilde{x}_i^0, \widetilde{x}_i^1$ ([4] suggests using a tamper-proof one-time-settable bit b_i which is set as soon as the OTM is queried). 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.[5] Finally, GC (and hence also OTP)

[5] If needed, the function can be fully hidden by evaluating a universal circuit [27, 11, 19] which simulates the function.

is inherently a one-time execution object (generalizable to k-time execution by repetition).

A subtle issue in this context, noted and addressed in [4], 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 adaptively chosen inputs results in possible real attacks by a malicious $\mathcal{R}$ in the non-interactive setting. The solution of [4] 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 $z'_j = z_j \oplus m_j$. The real-world adversary does not learn the output of the function before he had queried all OTMs with his inputs.

3 Extending and Using One-Time Programs

In §3.1 we present an extension of the OTP construction of [4], which results in improved performance for 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 [4] 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 [4], which is beneficial for OTP evaluation of functions with many output bits. 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 function to be evaluated has more outputs than the size of 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 the full version [7].

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 malicious $\mathcal{R}$ 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. 1(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. The GC is constructed as usual (e.g., as described in §2), 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.

We note that a full formal construction is readily obtained from the above description. Also note that a malicious $\mathcal{R}$ 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 the full version [7]. □

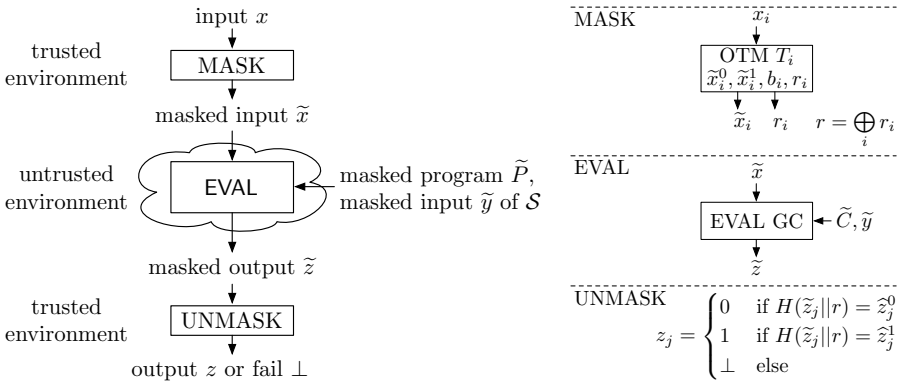
3.2 Using One-Time Programs for Leakage Protection

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 [1]). 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. 1(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.



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

Fig. 1. Evaluating a Functionality Without Leakage

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 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 [4]: For small functionalities, we favor the very cheap One-Time Memory (OTM), as this seems to carry the weakest assumptions (cf. §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.
- TPM [6]: Trusted Platform Modules (TPM) are low-cost tamper-proof cryptographic chips embedded in many of today's PCs [26]. TPM masking based on the non-interactive Oblivious Transfer (OT) protocol of [6] 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. An interactive protocol allows re-initialization for future non-interactive OTs 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 [8]. 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 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 semi-honest 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 [4]; 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 [8]. 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. 1(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.

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 [18]), we take SFE of the AES function as our example for timings and other measurements. 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.

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). 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. 2 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.

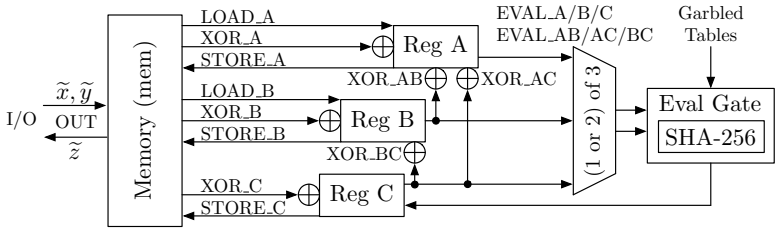


Fig. 2. Architecture for GC Evaluation (EVAL) on Memory-Constrained Devices

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 [10] as follows. XOR_A `addr` computes $A \leftarrow A \oplus \text{mem[addr]}$. Similarly, the other one-operand XOR operations (XOR1) XOR_B 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 [18] are evaluated with the Eval Gate block that contains a hardware accelerator for SHA-256 (cf. §2 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 are always stored at the first $|x| + |y|$ memory addresses. The garbled outputs are obtained from memory with OUT instructions.

The full version [7] shows the sequence of instructions for an example circuit.

4.2 Compile-time Optimizations for Memory-Constrained Devices

In this section, we summarize 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, we use AES as representative functionality for the optimizations and performance measurements, but our techniques are generic.

Baseline [18]) Our baseline is the AES circuit/code of [18], 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 ≈ 529 kByte. Without considering any caching strategies, this results in $113,054$ instructions, hence the program size is $113,054 \cdot 32$ bit ≈ 442 kByte, and the total amount of memory needed for EVAL is $34,136 \cdot 128$ bit ≈ 533 kByte.

We summarize our best optimization next and refer for a detailed description and intermediate optimization steps to the full version [7].

Optimized) First, we replace XNOR gates with an XOR gates and propagate the inverted output into the successor gates. For AES, this optimization results in the elimination of $4,086$ XNOR gates and reduces the size of AES GC to ≈ 338 kByte (improvement of 36%). Additionally, we re-use values already in registers to reduce the number of LOADs. Values in registers are saved to memory only if needed later. Finally, we randomly consider several orders of evaluation, and select the most efficient one for EVAL.

Result. Using our optimizations we were able to substantially decrease the memory footprint of EVAL. As shown in Table 1, our optimized circuit strongly improves over the circuit of [18] as follows. The size of the AES program P is only $73,583 \cdot 32$ bit ≈ 287 kByte (improvement of 34.9%). The amount of intermediate memory is $17,315 \cdot 128$ bit ≈ 271 kByte (improvement of 49.3%) and the number of memory accesses (read and write) is reduced by $\approx 35\%$.

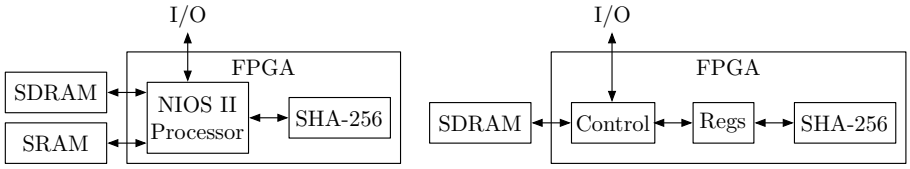
Table 1. Optimized AES Circuits (Sizes in kB)

	Garbled Circuit $\widetilde{C}$				Program P		Memory for GC Evaluation			
Circuit	non-XOR	1-input	XOR	Size	Instr.	Size	Read	Write	Entries	Size
Baseline [18]	11,286	0	22,594	529	113,054	442	67,760	33,880	34,136	533
Optimized	7,200	40	26,680	338	73,583	287	42,853	22,650	17,315	271

4.3 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 (reflecting 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 running at 50 MHz (cf. full version [7] for details on our prototype environment) 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. The interfaces (I/Os in Fig. 3) allow the host to write to and read from the SDRAM. In the beginning, the host writes the inputs to the SDRAM and, in the end, the outputs are written into specific addresses from which the host retrieves them.



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

Fig. 3. Architectures for Hardware-Assisted GC Evaluation

System-on-a-Programmable-Chip (SOPC). Our first implementation is a system-on-a-programmable-chip (SOPC) consisting of a processor with access to a hardware accelerator for SHA-256, which speeds up the most 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 [22].

The architecture of our implementation is shown in Fig. 3(a). It 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 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 is stored in SRAM whereas OTP related data is stored in SDRAM.

Stand-Alone Unit. Our 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. The architecture is depicted in Fig. 3(b).

When the host has written the inputs to the SDRAM, the stand-alone unit 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).

Area. The area requirements of our implementations are shown in Table 2. Both fit into the low-cost Cyclone II FPGA with 18,754 logic cells (LC), each containing a 4-to-1-bit look-up table (LUT) and a flip-flop (FF), and 52 4096-bit embedded memory blocks (M4K). SHA-256 is the largest and most significant block in both prototypes. Table 2 also shows the area for an iterative implementation of AES-128 with no countermeasures against side-channel attacks on the same FPGA. Compared to an unprotected implementation, countermeasures against power analysis have area overheads ranging from factor of 1.5 to 5 [23] as discussed in §1.2; therefore, the area overheads of OTP evaluation are comparable with other side-channel countermeasures.

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

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 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 is faster than EVAL2 because it accesses 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 evaluate the optimized GCs 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. The SHA-256 computations take an equal amount of time for both implementations as 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 optimized AES circuit. Hence, accelerating memory access, e.g., with burst reads and writes, is the key for further speedups.

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)	
Circuit	SHA	Total	SHA	Total	SHA	Total	SHA	Total
Baseline [18]	744,876	94,675,402	14.898	1,893.508	744,876	11,235,118	14.898	224,702
Optimized	477,840	62,629,261	9.557	1,252.585	477,840	7,201,150	9.557	144.023

Performance Comparison. A software implementation that evaluates the GC/OTP of the unoptimized AES functionality (Baseline [18]) required 2 seconds on an Intel Core 2 Duo 3.0 GHz with 4GB of RAM [18]. Our optimized circuit 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 with countermeasures against differential power analysis was factor of 3.88 [24].

Acknowledgements. We thank anonymous reviewers of CHES'10 for their helpful comments and co-authors of [18] for the initial AES circuit.

References

1. M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *CHES'01*, volume 2162 of *LNCS*, pages 309–318. Springer, 2001.
2. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In *CHES'02*, volume 2523 of *LNCS*, pages 13–28. Springer, 2003.
3. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC'09*, pages 169–178. ACM, 2009.
4. S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In *CRYPTO'08*, volume 5157 of *LNCS*, pages 39–56. Springer, 2008.
5. V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC'10*, volume 5978 of *LNCS*, pages 308–326. Springer, 2010.
6. V. Gunupudi and S. Tate. Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In *FC'08*, volume 5143 of *LNCS*, pages 98–112. Springer, 2008.
7. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. Cryptology ePrint Archive, Report 2010/276. `http://eprint.iacr.org`.
8. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Embedded SFE: Offloading server and network using hardware tokens. In *FC'10*, volume 6052 of *LNCS*, pages 207–221. Springer, 2010.
9. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.

10. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP'08*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.
11. V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC'08*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008.
12. Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.
13. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security'04*. USENIX Association, 2004.
14. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
15. T. S. Messerges. Using second-order power analysis to attack DPA resistant software. In *CHES'00*, volume 1965 of *LNCS*, pages 238–251. Springer, 2000.
16. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA'01*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.
17. K. Pietrzak. Provable security for physical cryptography. In *WEWORC'09*, 2009. `http://homepages.cwi.nl/~pietrzak/publications/Pie09b.pdf`.
18. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT'09*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
19. A.-R. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *ICISC'08*, volume 5461 of *LNCS*, pages 336–353. Springer, 2008.
20. 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.
21. 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/`.
22. 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`.
23. K. Tiri. Side-channel attack pitfalls. In *DAC'07*, pages 15–20. ACM, 2007.
24. 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 *CHES '05*, volume 3659 of *LNCS*, pages 354–365. Springer, 2005.
25. K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *DATE'04*, volume 1, pages 246–251. IEEE, 2004.
26. Trusted Computing Group (TCG). TPM main specification. Technical report, TCG, May 2009. `http://www.trustedcomputinggroup.org`.
27. L. G. Valiant. Universal circuits (preliminary report). In *STOC'76*, pages 196–203. ACM, 1976.
28. S. H. Weingart. Physical security devices for computer subsystems: A survey of attacks and defences. In *CHES'00*, volume 1965 of *LNCS*, pages 302–317. Springer, 2000.
29. A. C. Yao. How to generate and exchange secrets. In *FOCS'86*, pages 162–167. IEEE, 1986.