On secure embedded token design (Long Version)

Quasi-looped Yao circuits and bounded leakage

S. Hoerder[1] and K. Järvinen[2] and D. Page[1]

[1] University of Bristol, Department of Computer Science,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.
{hoerder,page}@cs.bris.ac.uk
[2] Aalto University, Department of Information and Computer Science,
P.O. Box 15400, FI–00076 Aalto, Finland.
kimmo.jarvinen@aalto.fi

Abstract. Within a broader context of mobile and embedded computing, the design of practical, secure tokens that can store and/or process security-critical information remains an ongoing challenge. One aspect of this challenge is the threat of information leakage through side-channel attacks, which is exacerbated by any resource constraints. Although *any* countermeasure can be of value, it seems clear that approaches providing robust guarantees are most attractive. Along these lines, this paper extends previous work on use of Yao circuits via two contributions. First, we show how careful analysis can fix the maximum number of traces acquired during a DPA attack, effectively bounding leakage from a Yao-based token: for a low enough bound, the token can therefore be secured via conventional (potentially less robust) countermeasures. To achieve this we use modularised Yao circuits, which also support our second contribution: the first Yao-based implementation of a secure authentication payload, namely HMAC based on SHA-256.
Note: The primary version of this paper will be published in the proceedings for the Workshop in Information Security Theory and Practice (WISTP) 2013.

1 Introduction

A vast range of challenges and opportunities has emerged as a result of the (ongoing) proliferation of mobile and embedded computing. On one hand, an increase in computational and storage capability has enabled more complex application classes on which we now routinely and fundamentally depend. On the other hand, various supporting technologies and techniques need to keep pace with such developments. We consider a motivating example within this context, namely the establishment of a secure communication channel between some token and another party. Although we limit our remit to tokens that are more capable than a (basic) smart-card (e.g., a mobile telephone or USB token, with concrete examplars including the IBM ZTIC[3]), the secure, efficient implementation of such a device is clearly of central importance.

Although well studied cryptographic techniques can satisfy varied requirements at a high level, a diverse and expanding attack landscape means real-world security guarantees are still difficult to achieve: the field of physical security in particular, including both active fault injection and passive side-channel attacks, represents a central concern. Following an optional calibration phase, a typical side-channel attack consists of an acquisition phase wherein monitoring by the attacker yields profiles of execution, then an analysis phase that recovers a security-critical target value (e.g., cryptographic key material) from the profiles. Although practical bounds on the number of profiles collected or processed may exist (e.g., a limit on the duration of physical access), attacker capability in this respect scales over time (e.g., with Moore's law); ideally this should be catered for by any countermeasure.

Our focus is the threat of side-channel attacks such as Simple (SPA) and Differential Power Analysis (DPA) [11], including variations thereof; attacks based on electro-magnetic emission (stemming from power consumption) fall into this category. We cater for timing side-channels, but explicitly deem (semi-)invasive or active attacks as outside the scope of this paper. In our scenario,

[3] http://www.zurich.ibm.com/ztic/

profiles acquired take the form of traces that describe power consumption by the token. Development of approaches to prevent attacks of this sort is an active field of research. One broad class aims to produce an implementation of some primitive, then secure it by applying (typically attack-specific) countermeasures at an algorithmic, software and hardware level (or combinations thereof). Selected examples include schemes for hiding [18, Chapter 7] and masking [18, Chapter 10] input and/or intermediate values. An alternative class aims to formulate then implement a primitive which is secure-by-design. Following a philosophy that says security should be treated as a first-class goal [10, 26], alongside efficiency for example, this is an attractive direction in that (more) robust guarantees can be provided. However, as the emerging field of leakage-resilient primitives (see [28] for example) illustrates, difficulties remain. For instance, any such guarantees hinge on accurate modelling of the underlying platform (i.e., the token). Most importantly still, leakage-resilient cryptography has focused on assuring security provided leakage entropy remains below a certain bound; unfortunately, no practical means of (provably) satisfying such a bound is currently available.

This paper extends work on Yao circuits [32, 33, 17, 1, 6, 7, 16], especially their implementation on tokens [9, 8] as a means of performing leakage-resilient computation within the motivating scenario above. In our work the focus is practicality: to reduce manufacturing and verification cost, we aim to keep the entire architecture as simple as possible. Careful analysis of said architecture places a bound τ on the number of useful traces (resp. amount of leakage) an attacker can acquire. This forces an attacker to focus on development of more effective attacks rather than simply using more traces to cope with the signal-to-noise ratio; when combined with conventional countermeasures, it potentially renders such attacks moot. One can view this process as playing a similar role to key refresh [21, 20], but without the need for synchronization. In addition, we add the first secure authentication payload, HMAC [23], to the list of applications implemented as Yao circuits. Both contributions hinge on the use of modular circuit templates, held by the token, to form fresh Yao circuits (meaning circuit generation can cater for run-time parameters such as message length). The overall result is leakage-*bounded* implementations of both AES-128 and HMAC.

2 Background

An important aspect of formalising the ability of a side-channel attack(er) is the selection of a model that describes the form of leakage from a token (and thus exploitable by the attacker). The model proposed by Standaert et al. [27] can be directly applied to our scenario with just one minor modification. In said model, adversarial success depends, among other factors, on the number of oracle queries allowed per primitive independent of updates to secret values (e.g., use of key refresh schemes). We therefore replace the number of oracle queries with the number of observable leakage occurrences per secret value.

As an aside, we stress that all known leakage models implicitly focus on cryptographic primitives and subsume the system dimension into the device leakage function. While the step from a leakage-resilient primitive to a leakage-resilient system is trivial in a model, it represents a significant practical obstacle. If other (hardware *or* software) system components are badly, designed the system leaks despite the leakage-resilience of a particular cryptographic primitive. As such, we emphasise that all practically relevant countermeasures, have to be composable: it must be possible to compose a given countermeasure with other countermeasures for different leakage types *and* countermeasures for different components; we will show that our countermeasure composes easily with memory protection countermeasures.

2.1 Yao circuits at a high level

Consider some functionality f. A concrete implementation of f as a standard Boolean circuit, say B^f, can be evaluated using an input x to give an output $r = B^f(x)$. One can specialise the implementation wrt. some fixed input; we use B^f_x to denote such an implementation, and retain a similar form for evaluation $r = B^f_x(\cdot)$. At a high level, Yao circuits can be described in a similar manner: they simply represent a non-standard implementation of f, say Y^f, which allows the associated evaluation to be secure.

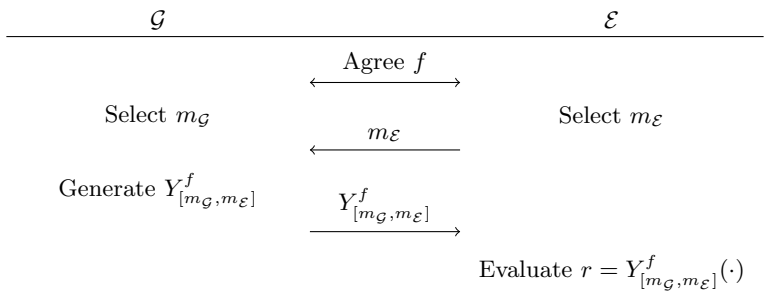


Fig. 1. A high-level, generic description of Yao circuit generation and evaluation. Note that all inputs and outputs are implicitly defined wrt. f, and that depending on the scenario a) one or more of f, $m_{\mathcal{G}}$ and $m_{\mathcal{E}}$ could alternatively be provided as input to the protocol, and b) subsequent use of r could be included.

Use of a given Yao circuit can be described formally as a secure two-party computation protocol. The parties involved are a circuit generator $\mathcal{G}$ who (given f and x) produces Y^f_x, and a circuit evaluator $\mathcal{E}$ who (given Y^f_x) computes $r = Y^f_x(\cdot)$; both are illustrated in Figure 1. We use Yao circuits to shift the side-channel attack surface away from an individual cryptographic primitive f, and onto the task of generating a Yao circuit $Y^f_{[m_{\mathcal{G}},m_{\mathcal{E}}]}$ (where we can give strong bounds on leakage). This differs from the original usage as a two-party computation protocol. For example, we do not need oblivious transfer to communicate the circuit inputs $m_{\mathcal{E}}, m_{\mathcal{G}}$, and the token $\mathcal{G}$ is trusted so may learn the inputs from $\mathcal{E}$. Note that in theory $\mathcal{G}$ could evaluate $Y^f_{[m_{\mathcal{G}},m_{\mathcal{E}}]}$ as well as generating it, but we deem it more economic in most cases to let $\mathcal{E}$ do the evaluation.

Imagine g_k refers to some k-th truth table wlog. describing a 2-input, 1-output Boolean function (e.g., AND, OR) or gate instance within B^f. Both the inputs to and outputs from said gate are provided on wires indexed using a unique wire identifier (or wire ID): we write m_i for the value carried by the i-th such wire, with the wire ID therefore being i. Note that the output wire ID can act to identify a given gate instance (i.e., acts as a gate ID). Figure 2a is a trivial starting point outlining how such a gate can be evaluated.

Yao circuits are constructed by taking each Boolean gate instance in B^f, then forming a corresponding "garbled" Yao gate instance in Y^f. Both inputs to and outputs from the Yao gate are altered to mask their underlying value: this means each underlying value on m_i is replaced by c_i, an encryption of the former. Given $\textsc{Enc}_x(y)$ denotes the encryption of y under the key x using some symmetric cipher (with a κ-bit key and β-bit block size), Figure 2b illustrates a Yao gate corresponding to the above. Note that

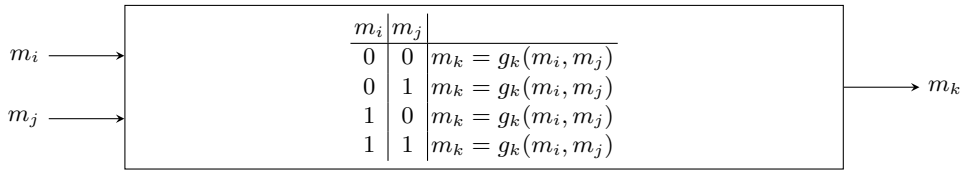
- the public c_i and c_j inputs (whose secret underlying values are m_i and m_j) are provided on wires with public indices i and j,
- the public c_k output (whose secret underlying value is m_k) is provided on a wire with public index k,
- the standard gate functionality is g_k, and
- π_i, π_j and π_k act as secret permutation bits on the rows of the truth table.

During evaluation of the gate, $\mathcal{E}$ gets c_i and c_j meaning it cannot recover the underlying values of m_i and m_j since it does not know π_i and π_j. However, c_i and c_j index *one* entry of the truth table and allow *only* this entry to be decrypted (since they determine the associated cipher key) and yield c_k. The central idea is that a Yao gate reveal nothing about a) the gate functionality nor b) underlying values, iff. it is evaluated at most once.

2.2 Abstract realisation of Yao circuits

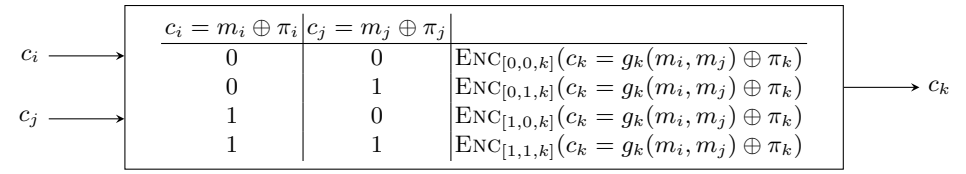
Wire labels The illustrative example above has a major shortcoming: the effective cipher key size is just two bits (since all wire IDs are public), meaning the key is inherently susceptible to exhaustive search. To combat this, given a security parameter λ one replaces the Boolean value communicated on each wire with a randomised λ-bit wire label. Let

$$w_i^{c_i} = \rho_i \parallel (m_i \oplus \pi_i) = \rho_i \parallel c_i \tag{1}$$



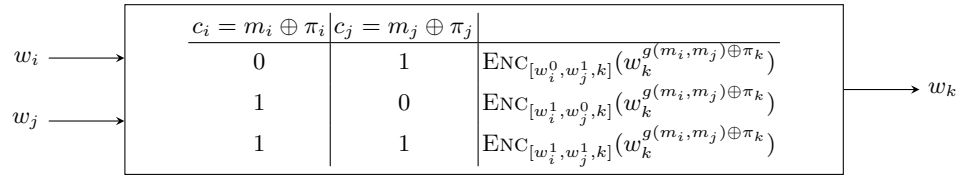
m_i	m_j	
0	0	$m_k = g_k(m_i, m_j)$
0	1	$m_k = g_k(m_i, m_j)$
1	0	$m_k = g_k(m_i, m_j)$
1	1	$m_k = g_k(m_i, m_j)$

(a) An example Boolean gate.



$c_i = m_i \oplus \pi_i$	$c_j = m_j \oplus \pi_j$	
0	0	$\textsc{Enc}_{[0,0,k]}(c_k = g_k(m_i, m_j) \oplus \pi_k)$
0	1	$\textsc{Enc}_{[0,1,k]}(c_k = g_k(m_i, m_j) \oplus \pi_k)$
1	0	$\textsc{Enc}_{[1,0,k]}(c_k = g_k(m_i, m_j) \oplus \pi_k)$
1	1	$\textsc{Enc}_{[1,1,k]}(c_k = g_k(m_i, m_j) \oplus \pi_k)$

(b) An illustrative example of the corresponding Yao gate.



$c_i = m_i \oplus \pi_i$	$c_j = m_j \oplus \pi_j$	
0	1	$\textsc{Enc}_{[w_i^0, w_j^1, k]}(w_k^{g(m_i, m_j) \oplus \pi_k})$
1	0	$\textsc{Enc}_{[w_i^1, w_j^0, k]}(w_k^{g(m_i, m_j) \oplus \pi_k})$
1	1	$\textsc{Enc}_{[w_i^1, w_j^1, k]}(w_k^{g(m_i, m_j) \oplus \pi_k})$

(c) The corresponding Yao gate with wire labelling and Garbled Row Reduction (GRR) [22] applied (noting that each c_i forms part of the associated w_i per Equation 1).

Fig. 2. A step-by-step comparison of a 2-input, 1-output Boolean gate (whose function is described by g_k), and the associated Yao gate construction before and after optimisation.

denote the i-th such wire label in the general case where $m_i \oplus \pi_i = c_i$ for $c_i \in \{0, 1\}$, $\rho_i \xleftarrow{\$} \mathbb{Z}_2^{\lambda-1}$ and $\pi_i \xleftarrow{\$} \mathbb{Z}_2$. Note that the combination of ρ_i and π_i could be viewed as a λ-bit ephemeral key, implying $\kappa = 2\lambda + \epsilon$ where ϵ is the number of bits used to encode wire IDs, and that Equation 1 caters for two optimisations outlined below.

The Garbled Row Reduction (GRR) optimisation The GRR trick was introduced by Naor et al. [22]. In short, by carefully selecting

$$w_k^{c_k} = w_k^{g(m_i, m_j) \oplus \pi_k} = \textsc{Enc}_{[w_i^0, w_j^0, k]}(\gamma) \tag{2}$$

for $c_i = c_j = 0$ and a suitable public constant γ, the first truth table entry can be eliminated (as illustrated by Figure 2c). This is attractive since it permits up to a 25% reduction in communication of the Yao circuit from $\mathcal{G}$ to $\mathcal{E}$, plus reduces the amount of storage required.

The "free XOR" optimisation There is no need for explicit inclusion of NOT gates in a Yao circuit, since their influence can simply be folded into any Yao gate which would normally use the associated output. Similarly, Kolesnikov and Schneider [12] describe a further optimisation which realises XOR gates (almost) for free. Given a global (i.e., one for each instance of Y^f), secret constant $\Delta \in \mathbb{Z}_2^{\lambda-1}$ they select w_i^1 as in Equation 1, then define

$$w_i^0 = w_i^1 \oplus (\Delta \parallel 1) = (\rho_i \oplus \Delta) \parallel (\pi_i \oplus (m_i \oplus 1)) \tag{3}$$

to allow computation of XOR gates via the relationships

$$\begin{aligned} w_i^0 \oplus w_i^1 &= w_i^1 \oplus (\Delta \parallel 1) \oplus w_j^1 &&= w_i^1 \oplus w_j^0 = w_k^1 \\ w_i^0 \oplus w_i^0 &= w_i^1 \oplus (\Delta \parallel 1) \oplus w_j^1 \oplus (\Delta \parallel 1) &&= w_i^1 \oplus w_j^1 = w_k^0 = w_k^1 \oplus (\Delta \parallel 1) \end{aligned}$$

2.3 Token implementations of Yao circuits

As far as cryptographic primitives are concerned, previous work (with the exception of [13]) focus on use of AES-128 as the functionality f. Other functionalities considered relate to higher level applications, e.g., database search [6, 16] or bioinformatics [7], rather than tasks required of a cryptographic token. Pinkas et al. [24] provide the first feasibility results (using software) of Yao-based constructions; since they relate more directly to the chosen scenario, we detail work by Järvinen et al. [9, 8] below which both implement Yao circuits on tokens but do not use modularisation.

Secure computation via One-Time Programs (OTPs) In this work, Järvinen et al. [9] consider a scenario wherein $\mathcal{E}$ is a hardware token, and $\mathcal{G}$ is a trusted party during a setup stage. The idea is that $\mathcal{G}$ as token issuer stores One-Time Memories (OTMs) for a fixed number x of OTPs represented by Y^f on the token. At run-time, the token owner uses one set of OTMs to form the input labels corresponding to his data, and the token evaluates the Yao circuit before finally releasing the result.

The advantage of this scenario is that very little protection against side-channel attacks is required. However, the major disadvantage is the limited number of Yao circuits: a real-life scenario where the token is a credit card rendered unusable after some number of uses make this problematic. In addition, a generic framework without this disadvantage is given at the cost of loosing the leakage-resilient circuit generation. Our work can be seen as a leakage-resilient, more flexible version of the framework.

Secure computation via out-sourcing In this work, Järvinen et al. [8] consider a scenario wherein $\mathcal{E}$ is a cloud computing provider; the role of $\mathcal{G}$ is split between a secure hardware token $\mathcal{G}_S$ and some other party $\mathcal{G}_U$ (e.g., a desktop workstation). The idea is for $\mathcal{G}_U$ to generate B^f, which is passed to $\mathcal{G}_S$ and translated (securely) into Y^f. The Y^f can then be evaluated by $\mathcal{E}$, with the overall effect of securely out-sourcing computation from $\mathcal{G}_U$ to $\mathcal{E}$.

This scenario is advantageous in the sense it allows a flexible choice of f (wrt. the token) and is very speed- and memory-efficient. However, it has a relatively high hardware requirement: in addition to the SHA-256 core it requires at least one AES-128 core, [8] uses two, which all have to be free from leakage.

3 Supporting alterations to traditional Yao circuits

To support our design in Section 4, we first outline two supporting concepts: detail relating to their realisation and utility is deferred until later. While neither represents a significant change to underlying theory, we posit that both significantly ease the practical task of constructing and using Yao circuits.

3.1 Circuit modularisation

Existing limitations Consider a typical iterative block cipher design, with s rounds in total (e.g, AES-128 with $s = 10$). The functionality for round i is described by f_i, which implemented as a Boolean circuit is B^{f_i}. The s different round functionalities can be the same or different as required, with the overall cipher thus described by the functionality

$$f = f_{s-1} \circ \cdots \circ f_1 \circ f_0.$$

One can implement this either combinatorially, whereby instances of each B^{f_i} are "unrolled" to form the resulting circuit, or iteratively, whereby instances of each unique B^{f_i} are "looped over" in steps under control of some auxiliary logic. Put another way, the former roughly describes the monolithic circuit whereas the latter describes iterated application with a need for only one circuit instance and some control logic. Traditional Yao circuits *must* adopt the former approach: no sequential elements (e.g., latches, clock signals) allowed because each gate can be evaluated at most once (to ensure security). One cannot reuse the resulting Yao circuit unless rerandomisable constructions [5] are considered; typically these incur a prohibitive overhead.

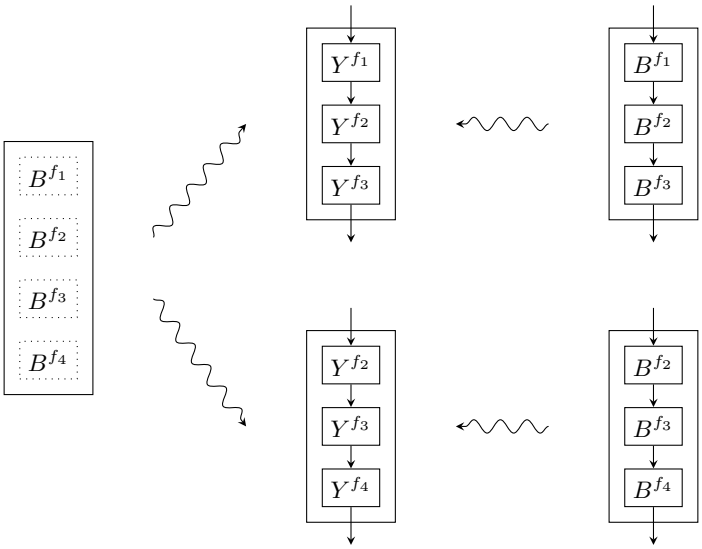


Fig. 3. A high-level illustration of Yao circuit modularisation: on the right the traditional (e.g., [17, 1, 9, 8]) monolithic scenario, on the left the modular alternative.

One impact of this restriction is that previous work almost exclusively focuses on AES-128 (as far as cryptographic primitives are concerned) which a) has a fixed s and can hence be unrolled, and b) has a fairly compact hardware implementation. We know of only one implementation of another cryptographic primitive [13], wherein an (insecure) implementation of 256-bit RSA is described. Even with such small operands, the resulting Yao circuit is ≈ 8500 times larger than their AES-128 circuit, in part as a result of the requirement to unroll the loop representing a binary, modular exponentiation.

Modularised Yao circuits To address this issue, we make use of modular Yao circuits. Similar concepts have been used recently[4] in [7, 16] to achieve efficiency gains for large circuits, but only [16] mentions the possibility of using run-time parameters to control circuit assembly.

The concept is illustrated by Figure 3. Traditionally (right) each monolithic Boolean circuit (internally composed of one or more modules, i.e., each B^{f_i}) must be translated into the corresponding Yao circuit by $\mathcal{G}$ each time the latter needs to be evaluated. In our alternative (left), $\mathcal{G}$ holds a static set of Boolean circuit templates which are instantiated at run-time to form an evaluatable Yao circuit. The main advantage of doing so is that $\mathcal{G}$ holds only the description of each template B^{f_i} (and associated meta-data), instantiating them to form Y^f without holding the entirety of the corresponding (potentially large) B^f. Generation of the corresponding Y^f can be streamed in that $\mathcal{G}$ communicates one part at a time to $\mathcal{E}$.

In short, this technique permits a quasi-loop: given s at run-time (rather than being fixed), the token approximates an iterative approach in that one circuit template can be unrolled, at run-time, to form the resulting Yao circuit. The resulting reduction in resource requirement and increased flexibility allows us to implement HMAC.

3.2 Updating Δ between template instances

In previous work [24, 9, 8] where the free XOR trick [12] is used, the authors argue that for correctness Δ must remain constant within a given Yao circuit. Indeed, if an XOR gate is evaluated using different constants Δ_1 and Δ_2, the result is incorrect as

$$\begin{aligned} w_i^0 \oplus w_j^1 &= (w_i^1 \oplus (\Delta_1 \parallel 1)) \oplus w_j^1 &&\neq w_i^1 \oplus (w_j^1 \oplus (\Delta_2 \parallel 1)) = w_i^1 \oplus w_j^0 \\ w_i^0 \oplus w_j^0 &= (w_i^1 \oplus (\Delta_1 \parallel 1)) \oplus (w_j^1 \oplus (\Delta_2 \parallel 1)) \neq && w_i^1 \oplus w_j^1 \end{aligned}$$

[4] A year earlier, [6] proposed a different kind of modularity, namely mixing Yao circuits and homomorphic operations. The cost associated with additional hardware required to support homomorphic operations means we do not adopt this approach.

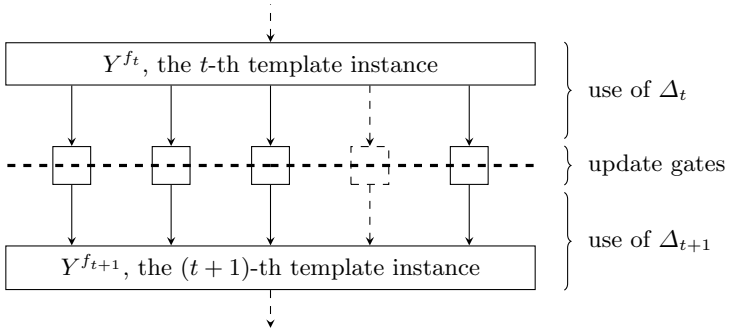


Fig. 4. An example of updating Δ between two template instances, with the heavy dashed line denoting the boundary between use of Δ_t and Δ_{t+1}.

shows. Despite this, a crucial observation is that Δ *can* be changed *if* said change is applied consistently. Although doing so has no functional benefit, we use it to directly formulate a leakage bound within Section 5.

In some modularised Yao circuit, imagine the t-th template instance uses Δ_t. The next, $(t+1)$-th instance can then use Δ_{t+1} (as illustrated by Figure 4) iff. each connecting wire is updated appropriately. The simplest approach is to consider a dedicated 1-input, 1-output update gate with the identity functionality, i.e., $g_k(x) = x$. Given

$$w_k^{c_k} = w_k^{m_i \oplus \pi_k} = \textsc{Enc}_{[w_i^0, w_i^0, k]}(\gamma)$$

by definition, and that $m_i = m_k$ since the gate updates Δ rather than change the underlying input value, Equation 2 means the associated wire labels are

$$\begin{aligned} w_i^0 &= w_i^1 \oplus (\ \Delta_t \ \ \|\ 1\ ) \\ w_k^0 &= w_k^1 \oplus (\ \Delta_{t+1} \ \|\ 1\ ) \end{aligned}$$

However, this suggests that *any* non-XOR gate can be used to perform the update without cost: the existing GRR-optimised Yao gate truth table only needs to have Δ_{t+1} folded into the label $w_k^{g_k(m_i,m_j)\oplus\pi_k}$ instead of Δ_t where appropriate. Selecting between the approaches is essentially a trade-off, governed in part by circuit structure: use of dedicated update gates incurs more overhead, but can be avoided iff. existing gates that generate the output from an instance are non-XOR.

4 Token design

We consider a scenario wherein $\mathcal{G}$ is a hardware token that needs to be secured against SPA and DPA attacks, and $\mathcal{E}$ is some other party to which computation is outsourced. The idea is to put a bound on the number of times secret values are used for any computation and leakage can be observed before the value – akin to key refresh – is updated. The known, residual leakage can then be accommodated by careful use of conventional countermeasures.

To do so, $\mathcal{G}$ holds a fixed set (limited only by memory) of circuit templates constituting B^f, which can be locally translated into a fresh Y^f subsequently evaluated by $\mathcal{E}$. This scenario is less flexible in terms of choice of f than [8], however the goal is to weaken both the token a) security and b) resource requirements, and hence produce a more practical result.

4.1 Cipher construction

To instantiate the symmetric cipher required to encrypt wire labels, we follow existing work [24, 8, 9] by using a one-time pad like construction

$$\textsc{Enc}_{[w_i^{c_i}, w_j^{c_j}, k]}(w_k^{c_k}) = \mathsf{SHA\text{-}256}(w_i^{c_i} \ \|\ w_j^{c_j} \ \|\ k) \oplus w_k^{c_k}$$

where the SHA-256 output is implicitly truncated to λ bits to match the wire label size. We reuse SHA-256 as a secure Pseudo-Random Number Generator (PRNG), splitting the SHA-256 into two 128-bit values

$$[x, y] \leftarrow \mathsf{SHA\text{-}256}(seed_{t-1})$$
$$seed_t \leftarrow seed_{t-1} \oplus y$$
$$rand \leftarrow x$$

so $x = rand$ is used as a wire label for example, while y is used exclusively to update the seed. Note SHA-256 is therefore the *only* significant cryptographic core required by the token. This construction is certainly secure if the PRNG is modelled as a random oracle which is a weaker model than the one we have for our Yao circuits. Intuitively however, some form of correlation robustness should be sufficient. Research on the correlation robustness of hash functions is still developing, see [12, 3] for example, and therefore we defer the exact security requirements to future work.

4.2 Describing circuit templates and functionalities

We use a VHDL[5] dialect and associated compiler, both of our own design, to describe structural and behavioural circuit templates. There are two major differences between our VHDL dialect and standard VHDL, namely

- we disallow concurrency since Yao circuits are strictly sequential, and
- we adapt the syntax of generate statements to allow run-time data, such as the number of message blocks, to determine the number of loop iterations.

To deploy a token, each VHDL entity in some input description is translated into a circuit template B^{f_i} and then stored in the token memory; a range of semantic checks are applied. Appendix A houses annotated examples of input, which expand on the description of functionalities using our VHDL dialect.

For each functionality composed of these templates, the token memory holds additional meta-data pointing to the corresponding top-level entity and the input data $m_{\mathcal{G}}$. The functionality itself resembles a tree with the root node represented by the top-level entity, internal nodes by structural B^{f_i} entities, and leaf nodes by behavioural B^{f_i} entities. In order to reconstruct (or unroll) the functionality, the token processes this tree in depth-first order by unrolling child nodes in the sequence specified by the VHDL input.

4.3 Operational protocol and token architecture

The communication between the token $\mathcal{G}$ and the evaluator $\mathcal{E}$ is shown in Figure 5. $\mathcal{E}$ can, for example, be a local untrusted work station or an untrusted but more powerful chip within a mobile phone; in most cases it will not be the authentication partner. We assume a physical connection between the parties, and hence focus on optimising their workload rather than the number of communication rounds.

Initially, in step 0, $\mathcal{E}$ requests a functionality f (e.g., HMAC or AES-128) and both parties need to have (or generate) corresponding inputs $m_{\mathcal{G}}$ and $m_{\mathcal{E}}$. Step 1, from a theoretic perspective, is the same as the monolithic communication in Figure 1 despite now supporting the modularised approach. Specifically, $\mathcal{G}$ generates the Yao circuit Y^f based on the circuit templates B^{f_i} and sends it step-by-step to be evaluated. Note that modularisation forces three important checks:

1. step $1d$ checks if all input values (from both $m_{\mathcal{G}}$ and $m_{\mathcal{E}}$) have been used,
2. step $1e$ checks if all gates have evaluated, and
3. step 4 checks if all output values (specified by $\mathcal{E}$) are valid, i.e., if $w_{o_j} \in \{w^0_{o_j}, w^1_{o_j}\}$ for each o_j.

When a check condition can not be satisfied the token aborts immediately, meaning in particular that it does not reach step 5 where the result $r = f(m_{\mathcal{G}}, m_{\mathcal{E}}) = Y^f_{[m_{\mathcal{G}}, m_{\mathcal{E}}]}$ is revealed, and that *seed* values of the PRNG are not accidentally reused.

[5] Previous work has used domain-specific languages (e.g. SFDL [17, 1]) to implement the payloads. While this *may* ease implementation, it reduces control over the actual circuit layout. Standard Hardware Description Languages (HDLs) are, in fact, well suited to payload description: they simply lack the protocol aspect of Yao circuits. Therefore we aim to reap benefits of familiarity, design and code portability, and control by harnessing VHDL instead.

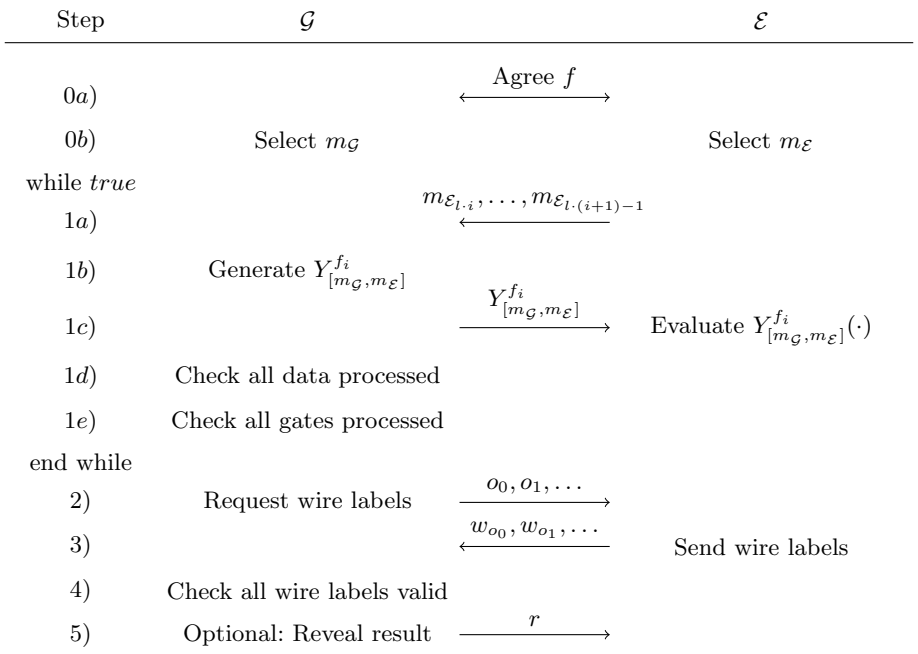


Fig. 5. The two-party computation protocol reflecting circuit modularisation. Note that $\mathcal{E}$ does not communicate $m_{\mathcal{E}}$ in one block, but rather in multiple l-bit blocks. Output wires of the Yao circuit (i.e. the wires carrying results from the functionality) have wire IDs $o_0, o_1, \ldots$.

Example 1*:* HMAC. Despite the invulnerability of SHA-256 itself to side-channel attacks (due to the absence of a security-critical target value), it is well known that MAC constructions based on hash functions are vulnerable; see e.g. [14, 19] which use DPA successfully against HMAC and [4] for traditional protections of HMAC. In this example, we use the flexibility afforded by modularisation to allow for messages of arbitrary length. The functionality for HMAC is modelled by

$$r = \mathsf{SHA\text{-}256}\Big((auth \oplus pad_2) \parallel \mathsf{SHA\text{-}256}\big((auth \oplus pad_1) \parallel m_{\mathcal{E}} \parallel padding\big)\Big)$$

We did not implement the message padding for our HMAC implementation as this can be done more efficiently via a non-Yao pre-processing step. Let `cv` denote the SHA-256 chaining variable, `m` denote an $l = 512$-bit message block, and `sha_cf` denote the SHA-256 compression function. It seems more economic to not implement the two round function iterations that process the padded authentication keys, but rather use pre-computed $\mathtt{cv}_1, \mathtt{cv}_2$ instead. Thus, we have the following correspondences with Figure 5:

$$\begin{aligned}
0b) \quad & [\mathtt{cv}_1, \mathtt{cv}_2] \leftrightarrow m_{\mathcal{G}} \\
0b) \quad & [\mathtt{m}_0, \ldots, \mathtt{m}_{s-1}] \leftrightarrow m_{\mathcal{E}} \\
1a) \quad & \mathtt{m}_i \leftrightarrow m_{\mathcal{E}_0}, \ldots, m_{\mathcal{E}_{l-1}} \\
1b) \quad & \mathtt{sha_cf} \leftrightarrow B^{\mathtt{sha_cf}} \rightsquigarrow Y^{\mathtt{sha_cf}}
\end{aligned}$$

After s rounds $m_{\mathcal{E}_i}$, the input from $\mathcal{E}$, is exhausted but the token still has to process $\mathtt{cv}_2$. This signals the beginning of the second SHA-256 function, i.e. the final instantiation, this time without evaluator input, of `sha_cf`. Only when this is finished, the token will proceed to step 2.

Example 2*:* AES-128. In this example, we use the flexibility afforded by modularisation to avoid the need for $\mathcal{G}$ to hold an entire unrolled implementation of AES-128 (cf. [24, 9, 8]), but rather to iterate over the round functions. We omitted implementation of the key schedule as we believe it to be more economic to store all round keys in the token memory. We used the Boolean formulas given by Boyar et al. [2] to implement the S-box. Let `rk` denote a round key, `m` the 128-bit message and `aes_rf` the round function consisting of `aes_ark`, `aes_sub` and `aes_mix` where the latter implements both

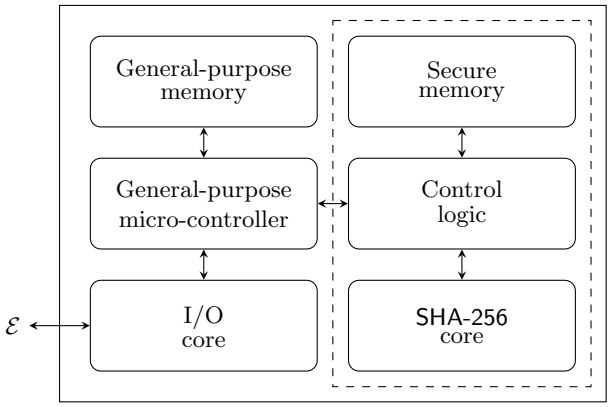


Fig. 6. A block diagram of our proposed token architecture. Only the area within the dashed box (right) has requirements for conventional side channel countermeasures (e.g., secure logic styles).

ShiftRow and MixColumn operations. For step 1 of Figure 5 we have the following correspondences:

$$\begin{aligned} 0b) \quad & [\mathtt{rk}_1, \ldots, \mathtt{rk}_{11}] \leftrightarrow m_{\mathcal{G}} \\ 0b) \quad & \mathtt{m} \leftrightarrow m_{\mathcal{E}} \\ 1a) \quad & \mathtt{m} \leftrightarrow m_{\mathcal{E}_0}, \ldots, m_{\mathcal{E}_{l-1}} \\ 1b) \quad & \mathtt{aes\_rf} \leftrightarrow B^{\mathtt{aes\_rf}} \rightsquigarrow Y^{\mathtt{aes\_rf}} \end{aligned}$$

After a specified nine iterations of `aes_rf`, the check in step $1d$ will show that $m_{\mathcal{E}}$ has been exhausted (which happened in the first iteration) but that $m_{\mathcal{G}}$ still contains the two data blocks $\mathtt{rk}_{10}, \mathtt{rk}_{11}$. Also, there are additional gates that need to be processed (step $1e$): `aes_ark`, `aes_sub` and another `aes_ark`. Once these last gates are processed, $m_{\mathcal{G}}$ is exhausted as well and the token can proceed to step 2.

To support the protocol outlined above, Figure 6 outlines a proposed token architecture. The main components and their roles are as follows:

- A general-purpose micro-controller manages the communication protocol in Figure 5, controlling assembly of Y^f from the B^{f_i} circuit templates and performing other tasks (such as message padding for HMAC). A separate input/output core is shown, but this role could also be assumed by the micro-controller.
- A general-purpose memory holds the circuit templates, micro-controller program and other public run-time variables values which only require correctness.
- Storage of and computation using secret values is limited to the Yao core, which consists of:
  - A SHA-256 core, used to encrypt wire labels and also as a PRNG.
  - Control logic used for auxiliary operations such as wire label generation and encryption.
  - A secure memory, split into two parts: a non-volatile part holds $m_{\mathcal{G}}$ and *seed*, while a volatile part holds Δ_t, Δ_{t+1} and, for each wire i, the tuple $\{w_i^0, w_i^1\}$. Note that $\Delta_{t+1} = 0$ unless the token is processing update gates, and that a crucial role of the secure memory as a whole is to prevent read-out or other leaks of values such as $m_{\mathcal{G}}$ and *seed*.

5 Analysis and results

5.1 Security analysis

This section attempts to explore security aspects of the proposal outlined in Section 4. After a statement of general assumptions, we deal specifically with potential attack vectors exploited during an SPA or DPA attack, or by a malicious adversary within the operational protocol.

Security assumptions An adversary is successful if the input $x_{\mathcal{G}}$ held by the token is recovered. Two main strategies seem obvious: one can attempt to recover $x_{\mathcal{G}}$ directly (e.g., via a DPA attack), or try to "ungarble" the Yao circuit $Y_{x_{\mathcal{G}}, x_{\mathcal{E}}^f}$ (or part thereof). In showing neither strategy is viable, we make some important assumptions:

A-1 An authentication protocol that prevents man-in-the-middle attacks against $m_{\mathcal{E}}$ in step $1d$ of the Yao protocol must be selected (if this threat is deemed relevant).

A-2 The control-flow of the token, managed by the micro-controller, is tamper-proof: countermeasures against fault injection must be implemented. This implies countermeasures against manipulation of the general-purpose memory. Although this is a strong requirement, it is no more strong than any other design.

A-3 The hash function used for encryption of wire labels (in our case SHA-256) must be circular-2-correlation robust. This assumption stems from the security proofs for semi-honest adversaries (see Choi et al. [3]).

A-4 The token cannot be reset, and no randomness reused. In case of an error (e.g., a failed check) the token must abort and generate new randomness: this prevents an adversary forcing the token to regenerate the same Yao circuit with the same randomness, then reevaluating it with different inputs.

Catering for power analysis adversaries SPA attacks attempt to recover the security-critical target value using one (or at least very few) traces; for example, this might imply examination of data-dependent control-flow. Either way, note that the potency of such an attack is largely independent from the number of traces collected. There are two possible attack vectors:

SPA-1 For each i-th input wire, the token must send either w_i^0 or w_i^1 to the evaluator depending on the underlying value of m_i. To succeed, the adversary must be able to determine whether $m_i \in x_{\mathcal{G}}$ is 0 or 1 (for all i).

SPA-2 Gates such as AND and OR are biased towards 0 or 1 in their output: if the adversary determines during computation of

$$\textsc{Enc}_{[w_i^{c_i}, w_j^{c_j}, k]}\left(w_k^{g(m_i, m_j) \oplus \pi_k}\right)$$

which truth table row contains the minority output, they can reverse the permutations (i.e., π_i and π_j) and recover the underlying values of almost all output wire labels from non-XOR gates.

Both SPA attack vectors concern choice, realised concretely using multiplexer components. For hardware implementations, the data dependency of the power consumption is usually already hidden well enough without countermeasures (e.g., [18, Appendix A.3]). Even if this is not the case, traditional countermeasures (e.g., random masking of the multiplexer select with corresponding permutation of the multiplexer inputs) are efficient.

In contrast, DPA attacks attempt to recover the security-critical target value by applying statistical distinguishers to a large set of traces; issues of signal-to-noise ratio, as well as explicit countermeasures, determine the exact number. More formally, let k be the target value, v be a variable value and r the result of some generic operation $\odot$. A common example is the XOR operation, representing addition of the state and a round key in the first round of AES-128. A DPA adversary collects traces relating to execution of $r_i \leftarrow k \odot v_i$ for many different $v_1, v_2, \ldots, v_\sigma$. The potency of a DPA attack is then judged by σ, the number of traces required to be reliably recover k. Our approach is to have a design-time constant bound $\tau \ll \sigma$ instead of allowing the adversary to control it. Put another way, we bound the leakage such an adversary can utilise in a DPA attack: if the application of conventional countermeasures can prevent attacks with said leakage level, the token is secure.

DPA-1 The token must compute

$$w_i^0 \leftarrow w_i^1 \oplus (\Delta_t \parallel 1)$$

for every wire. As such we have

$$\tau_{\textbf{DPA-1}} = \max(\delta_1, \delta_2, \ldots)$$

where δ_t denotes the number of wires using Δ_t. If the technique in Section 3.2 is used correctly, $\tau_{\textbf{DPA-1}}$ is a constant determined by the token designer (instead of the attacker).

DPA-2 For each gate, the four values of $w_{\{i,j\}}^{\{0,1\}}$ are each used twice as input to

$$\mathsf{SHA\text{-}256}(w_i^{\{0,1\}} \parallel w_j^{\{0,1\}} \parallel k).$$

Focusing on one label, wlog. w_i say, and one external value, wlog. 0 say, the attacker gets two traces for each gate where w_i^0 is used as input. Therefore, we have

$$\tau_{\mathbf{DPA\text{-}2}} = 2 \cdot \max_{\forall k}(G_k)$$

where G_k represents the fan-out of the k-th gate (and input wires are also considered as being driven by imaginary gates). Note that a similar attack vector exists when the token processes an XOR gate. Such a gate must compute w_k^0 and w_k^1, and one possible approach is to compute

$$\begin{array}{l} w_k^0 \leftarrow w_i^0 \oplus w_j^0 \\ w_k^1 \leftarrow w_i^0 \oplus w_j^1 \end{array} \tag{4}$$

in which case $\tau_{\mathbf{DPA\text{-}2}}$ conveniently covers this attack vector as well.

Concrete, non-optimised examples for these bounds are given in Section 5.2. If our design is used to protect against DPA attacks, functionalities that were inherently secure against SPA clearly inherit any SPA vulnerabilities of the underlying Yao circuit approach. We suggest that preventing SPA attacks on our design using conventional countermeasures is, broadly speaking, easier than preventing DPA attacks on the functionality in question: the cost of preventing the former is easily justified by the improvement offered wrt. the latter.

Catering for timing analysis adversaries The execution time associated with generating of a Yao circuit is inherently independent of the inputs to that Yao circuit: it depends *only* on the circuit size. As far as the architecture is concerned, we do not use a cache for the micro-processor in order to avoid cache-based timing attacks. Working without a cache is a common decision for cryptographic tokens and therefore not an exceptional burden of our design.

Catering for malicious adversaries One advantage of Yao circuits is the availability of related security proofs. For semi-honest adversaries, Figure 5 preserves security proofs already given by [12, 24, 8, 9, 3]. This is a direct result of the loop (over steps $1a$ to $1e$) being equivalent to the single generate-evaluate step from previous protocols.

However, we also need to consider malicious adversaries. Lindell et al. [15] show how a two-party computation protocol using Yao circuits can cover the case of malicious $\mathcal{G}$ (to ensure that Y^f is the required functionality) using a cut-and-choose approach. Our scenario is far less complex, since Y^f is generated by the token which is implicitly trusted: we disallow a malicious $\mathcal{G}$. Therefore we only have to consider a malicious $\mathcal{E}$, and show it cannot learn anything about $x_{\mathcal{G}}$ not implied by the result $r = f(x_{\mathcal{G}}, x_{\mathcal{E}})$.

In summary, it remains to be shown that any possible protocol deviation by $\mathcal{E}$ is harmless. $\mathcal{E}$ has two options (which we expand on below): it can either a) provide faulty input or b) perform variants on early termination.

Faulty data. The only steps where $\mathcal{E}$ can provide faulty data are $1a$ and 3. As $m_{\mathcal{E}}$ in step $1a$ is the input of the functionality, sending a faulty $m_{\mathcal{E}}$ has no impact on the security of the Yao protocol: it can only influence the output of the functionality f. In contrast, sending faulty w_{o_j} in step 3 is potentially a problem if the adversary sends labels from intermediate wires instead of the output labels. However, this is prevented by the check in step 4 which ensures that for each output wire o_j, exactly one wire label $w_{o_j}^{c_{o_j}} \in \{w_{o_j}^0, w_{o_j}^1\}$ has been sent before the result is revealed.

Early termination. Since $\mathcal{E}$ can not learn anything from one of the partial circuits (i.e., a given $Y_{[m_{\mathcal{G}}, m_{\mathcal{E}}]}^{f_i}$) until the protocol is finished (i.e., until $\mathcal{G}$ reveals the result), $\mathcal{E}$ cannot profit from straight early termination. However, if the functionality f requires s iterations of a loop to form $f = f_{s-1} \circ \cdots \circ f_1 \circ f_0$, per the description of AES-128 in Section 3.1 for instance, the adversary could

	#blocks	$\#\Delta$	#XOR	#non-XOR	#SHA-256	RAM	$\lvert B^{f_i} \rvert$	$\lvert m_{\mathcal{G},sec} \rvert$	$\lvert m_{\mathcal{G},pub} \rvert$	$\tau_{\textbf{DPA-1}}$	$\tau_{\textbf{DPA-2}}$
AES	1	1	19088	5760	24578	245.9kB	12318B	176B	–	7296	11
AES_U1	1	2	19088	5888	25091	263.4kB	13628B	176B	–	3776	11
AES_U9	1	10	19088	6912	29195	262.3kB	12845B	176B	–	960	11
HMAC	1	1	148080	129680	556866	1883.6kB	121942B	64B	32B	167824	19
	2	1	222120	194520	835170	2489.8kB	121942B	64B	32B	251608	19
	3	1	296160	260384	1113474	3069.0kB	121942B	64B	32B	335392	19
	4	1	370200	324200	1391778	3671.4kB	121942B	64B	32B	419176	19
HMAC_U	1	3	148080	130192	558916	1911.0kB	122981B	64B	32B	84040	19
	2	4	222120	195288	835170	2500.8kB	122981B	64B	32B	84040	19
	3	5	296160	260384	1117574	3113.4kB	122981B	64B	32B	84040	19
	4	6	370200	325480	1396903	3724.6kB	122981B	64B	32B	84040	19

Table 1. Efficiency metrics and leakage bounds for our token design and a range of payload implementations. The block size for AES-128 is 128 bits, for HMAC it is 512 bits (including the padding in the last block).

potentially gain information from terminating the loop early, i.e., to get $f' = f_{s'-1} \circ \cdots \circ f_1 \circ f_0$ for $0 < s' < s$: this would be analogous to a reduced-round attack. To prevent this, we require the token to check (in steps $1d$ and $1c$) whether the Yao circuit for f has been completely generated or whether some $Y^{f_i}_{[m_{\mathcal{G}}, m_{\mathcal{E}}]}$ is missing.

5.2 Experimental results and analysis

For the evaluation of our proposed design, we implemented a VHDL compiler (per Section 4.2), a token simulator $\mathcal{G}$, an evaluator $\mathcal{E}$ as well as two payloads functionalities, namely AES-128 and $\mathsf{HMAC}_{\mathsf{SHA\text{-}256}}$. As the use of a simulator suggests, our goal is to study gross, indicative metrics and trade-offs rather than focus on absolute figures that could be improved via incremental optimisation.

For each payload, we considered variants that differ in their frequency of Δ update: for AES-128 three variants are used, for $\mathsf{HMAC}_{\mathsf{SHA\text{-}256}}$ two variants. The variants are as follows:

AES	Baseline AES-128 implementation without updating of Δ.
AES_U1	AES-128 with a Δ update after the fifth iteration of the round function.
AES_U9	AES-128 with a Δ update after every iteration of the round function.
HMAC	Baseline HMAC implementation without updating of Δ.
HMAC_U	HMAC with Δ being updated after every iteration of the compression function.

Table 1 details efficiency metrics for implementation of these variants on the platform described and shows the two associated bounds $\tau_{\textbf{DPA-1}}$ and $\tau_{\textbf{DPA-2}}$. The first three columns specify the payload, the number of input blocks from $\mathcal{E}$ and the number of Δ values being used at run-time.

Efficiency The columns #XOR and #non-XOR in Table 1 give the number of gates in the resultant Yao circuit. Compared to [24, 9] we have considerably smaller AES-128 circuits, which is mainly due to omission of key scheduling and, to a less extent, use of more optimised S-box formulas of Boyar et al. [2]. The omission of key scheduling implies a small penalty of having to store all round keys $m_{\mathcal{G},sec}$ in secure ROM.

The column #SHA-256 shows the number of distinct uses of the SHA-256 core, for a one-block hash in each case: this figure is directly related to the number of non-XOR gates and the number of inputs wires to the Yao circuit. Ignoring the absolute simulation time, we feel this metric best represents the execution time of a concrete token since the SHA-256 core will most likely be the throughput bottleneck. [8] use a SHA-256 core which requires 67 cycles per 512 bit block at 66 MHz. Based on these numbers, a crude time estimation (based only on calls to the SHA-256 core) is 24ms for AES and 1418ms for HMAC_U with 4 message blocks.

A significant issue is the amount of RAM required at run-time. To assess this, we measured the simulator heap and stack usage using the Valgrind `massif` tool [31]. We note that the tool itself is

not perfect, and that the result includes overhead of up to 20% relating to performance and security counters (esp. for the per-wire counters used to determine $\tau_{\textbf{DPA-2}}$). Even so, the indicative RAM requirement is large: it remains within the capability of devices in our remit, but clearly beyond smart-cards or RFID tokens for example. The requirement stems in the most part from storing *all* wire labels $\{w_i^0, w_i^1\}$ in RAM. One possible trade-off would be to store only w_i^0 and recompute w_i^1 when needed. This would reduce the RAM usage by a factor of two, but increase the number of traces available by a factor similar to the maximum fan-out. [8] chose a keyed PRNG which allows recomputation of w_i^0 when needed, thus reducing the RAM requirements drastically. However, any keyed PRNG is vulnerable to DPA attacks with unlimited τ which negates our aim of bounding the leakage.

An interesting observation can be made about the RAM usage of `AES_U1` and `AES_U9`. Intuitively, one would expect the RAM usage to always grow in line with the number of Δ_t used: intuitively, `AES_U9` should need more RAM than `AES_U1`. However, the opposite is true. This happens because `AES_U1` applies the Δ updating within the top-level entity (which also accounts for the larger $|B^{f_i}|$), requiring more wires for which RAM is allocated during the entire run-time. `AES_U9` performs the updating at the end of the round function entity instead, and the RAM for additional wires can be deallocated as soon as each round function instance of has been completed.

The size of the templates, $|B^{f_i}|$ (stored in unsecured ROM), profits directly from modularisation. As predicted, the size of $|B^{f_i}|$ for HMAC does not depend on the number of message blocks being processed as it would have for the traditional approach.

Security Having explained $\tau_{\textbf{DPA-1}}$ and $\tau_{\textbf{DPA-2}}$ in Section 5.1, we note that our Δ updating technique limits $\tau_{\textbf{DPA-1}}$ as predicted; note esp. the `HMAC_U` payload, where updating Δ fixes previously unlimited leakage to a constant chosen by the token designer.

The result for $\tau_{\textbf{DPA-2}}$ is an absolute upper bound, i.e., for all output wires we counted how often it gets used while processing the follow-up gates. As explained in Section 5.1, if a wire is used as input to a non-XOR gate each label gets used twice; for XOR gates Equation 4 gives the numbers relevant to our implementation. For an attacker it will be very difficult to combine traces from two different operations like this but we prefer to err on the side of security by overestimating the attacker. However, as the numbers for $\tau_{\textbf{DPA-1}}$ dwarf the numbers for $\tau_{\textbf{DPA-2}}$ by several orders of magnitude, **DPA-2** is almost irrelevant as an attack vector. Having a low $\tau_{\textbf{DPA-2}}$ was an explicit aim of our work: $\tau_{\textbf{DPA-2}}$ is the only possible attack vector on the SHA-256 core, and therefore $\tau_{\textbf{DPA-2}}$ is the crucial factor to determine the level of conventional countermeasures needed to protect the SHA-256 core. Compared to the SHA-256 core, protecting the XOR from **DPA-1** to match a much higher $\tau_{\textbf{DPA-1}}$ is inexpensive.

As a reference one may look at the Power-Trust micro-processor of Tillich et al. [30], which has parts of the ALU implemented within a secure zone. For evaluation purposes they implemented the secure zone in three different logic styles (namely CMOS, iMDPL [25] and DWDDL [34]) and performed a DPA attack against an AES-128 software implementation using the secure zone. While it is difficult to directly extrapolate from a design as different from ours, this at least gives an estimate: there is no reason why secure logic styles such as iMDPL and DWDDL should fare worse for our token. For the DPA attack on the secure zone to be successful, Tillich et al. required $130,000$ traces against the (unprotected) CMOS implementation, $260,000$ traces against the iMDPL implementation and $675,000$ traces against the DWDDL implementation. With $\tau_{\textbf{DPA-1}} = 7296$ in the worst case for AES-128 and $\tau_{\textbf{DPA-1}} = 84040$ for `HMAC_U` we surmise that both iMDPL and DWDDL would have successfully thwarted the DPA attack from Tillich et al. against an implementation of our token. It is important to note, that for both bounds we did not yet try to find the absolute minimum. For example it is possible to add additional gates to achieve fan-out $= 2$ and thus $\tau_{\textbf{DPA-2}} \leq 4$ while $\tau_{\textbf{DPA-1}}$ can be easily reduced by updating Δ more often within the round resp. compression functions, not just at their end.

Potential improvements: Based on these results, we have some improvements to suggest.

1. RAM usage: The compiler should put annotations into the B^{f_i} to tell the token the earliest possible point to deallocate RAM for wires.

2. Execution time: With additional SHA-256 cores in the Yao core, parallelized entity processing is possible. Again, the compiler should put annotations into the B^{f_i} telling it which entities can be parallelized. With two SHA-256 cores we would expect the time needed to generate $Y^{\text{AES-128}}$ to be halved.
3. $\tau_{\textbf{DPA-2}}$: If the token supports more than one Δ at a time, then different Δ can be used for parallelized B^{f_i}. Thus $\tau_{\textbf{DPA-2}}$ can be reduced even further than described above.
4. Secure memory: Integration of memory masking into Yao circuits will be an important step in order to secure the token's memory and bus against leakage.

6 Conclusions

In essence, this paper has demonstrated that an embedded token can be designed which gives strong bounds on the number of useful traces a power analysis adversary can collect. Our design methodology

1. is generic in that it works for all payloads and use-cases (cf. PIN block schemes),
2. does not impose limits on the token lifetime,
3. does not require synchronization (cf. key update schemes),
4. is easily verifiable, and
5. can successfully thwart side-channel attacks in connection with conventional countermeasures.

In relation to the former point, we have already extended previous work through support for a Yao circuit for HMAC. Exploration of further primitives based on modularisation (including methods and trade-offs to further reduce the leakage bound), plus incremental optimisation of both the token design and operational protocol (especially the RAM requirement) are interesting avenues for further work. In relation to the latter point, the clear next step is to produce experimental results from a concrete implementation of the token. This would, for example, allow investigation of the concrete leakage and, given a τ, whether the implementation can definitively be secured using conventional countermeasures as expected. Special considerations should also be given to higher order attacks ([18, Chapter 10]); it is clear that the number of leakage occurrences available to an adversary will be bounded as well, but the exact τ may include implementation specific additional higher order events that can not be detected by formal analysis as presented here.

As a final note, we want to highlight the fact that our countermeasure easily combines with memory masking (see e.g. [29, 30]) or memory encryption countermeasures; instead of unmasking (resp. decrypting) a value before it is being encoded for the Yao circuit, unmasking (resp. decryption) can be easily integrated into the Yao circuit. This does not solve the issue of storing the masks (resp. keys used for memory encryption) but that issue exists independently of our technique. However, combining both countermeasures has the interesting result of a system where long term keys never have to exist unmasked (resp. unencrypted) within the token.

Acknowledgements The work described in this paper has been supported in part by EPSRC grant EP/H001689/1 and by Academy of Finland, project #138358. We would like to thank Elisabeth Oswald for her valuable comments.

References

1. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP - A Secure Multi-Party Computation System. In *CCS*, pages 257–266, 2008.
2. J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *SEA*, volume LNCS 6049, pages 178–189, 2010.
3. S.G. Choi, J. Katz, and R. Kumaresan. On the Security of the "Free-XOR" Technique. In *TCC*, volume LNCS 7194, pages 39–53, 2012.
4. J.-S. Coron and A. Tchulkine. A New Algorithm for Switching from Arithmetic to Boolean Masking. In *CHES*, volume LNCS 2779, pages 89–97, 2003.
5. C. Gentry, S. Halevi, and V. Vaikuntanathan. i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits. In *CRYPTO*, volume LNCS 6223, pages 155–172, 2010.
6. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *CCS*, pages 451–462, 2010.

7. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.
8. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Embedded SFE: Offloading Server and Network using Hardware Tokens (short version). In *Financial Cryptography*, volume LNCS 6052, pages 207–221, 2010.
9. K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled Circuits for Leakage-Resilience: Hardware Implementation and Evaluation of One-Time Programs. In *CHES*, volume LNCS 6225, pages 383–397, 2010.
10. P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi. Security as a New Dimension in Embedded System Design. In *DAC*, pages 753–760, 2004.
11. P.C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO*, volume LNCS 1666, pages 388–397, 1999.
12. V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*, volume LNCS 5126, pages 486–498, 2008.
13. B. Kreuter, A. Shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, 2012.
14. K. Lemke, K. Schramm, and C. Paar. DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In *CHES*, volume LNCS 3156, pages 205–219, 2004.
15. Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *EUROCRYPT*, volume LNCS 4515, pages 52–78, 2007.
16. L. Malka and J. Katz. VMCrypt – Modular Software Architecture for Scalable Secure Computation. In *CCS*, pages 715–724, 2011.
17. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *USENIX Security Symposium*, pages 287–302, 2004.
18. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer-Verlag, 2007.
19. R. McEvoy, M. Tunstall, C.C. Murphy, and W.P. Marnane. Differential Power Analysis of HMAC based on SHA-2, and Countermeasures. In *WISA*, volume LNCS 4867, pages 317–332, 2007.
20. M. Medwed, C. Petit, F. Regazzoni, M. Renauld, and F.-X. Standaert. Fresh re-keying II: Securing multiple parties against side-channel and fault attacks. In *CARDIS*, pages 115–132. LNCS 7079, 2011.
21. M. Medwed, F.-X. Standaert, J. Großschädl, and F. Regazzoni. Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In *AFRICACRYPT*, pages 279–296. LNCS 6055, 2010.
22. M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Electronic Commerce*, pages 129–139, 1999.
23. National Institute of Standards and Technology (NIST). The Keyed-Hash Message Authentication Code (HMAC). Federal Information Processing Standards Publication 198-1, Jul. 2008.
24. B. Pinkas, T. Schneider, N.P. Smart, and S.C. Williams. Secure Two-Party Computation is Practical. In *ASIACRYPT*, volume LNCS 5912, pages 250–267, 2009.
25. T. Popp, M. Kirschbaum, T. Zefferer, and S. Mangard. Evaluation of the Masked Logic Style MDPL on a Prototype Chip. In *CHES*, volume LNCS 4727, pages 81–94, 2007.
26. S. Ravi, A. Raghunathan, P.C. Kocher, and S. Hattangady. Security in Embedded Systems: Design Challenges. *TECS*, 3(3):461–491, 2004.
27. F.-X. Standaert, T.G. Malkin, and M. Yung. A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In *EuroCrypt*, volume LNCS 5479, pages 443–461, 2009.
28. F.-X. Standaert, O. Pereira, Y. Yu, J.-J. Quisquater, M. Yung, and E. Oswald. Leakage Resilient Cryptography in Practice. In *Towards Hardware-Intrinsic Security*, pages 99–134. 2010.
29. S. Tillich, M. Kirschbaum, and A. Szekely. SCA-Resistant Embedded Processors - The Next Generation. In *ACSAC*, pages 211–220, 2010.
30. S. Tillich, M. Kirschbaum, and A. Szekely. Implementation and Evaluation of an SCA-Resistant Embedded Processor. In *CARDIS*, volume LNCS 7079, pages 151–165, 2011.
31. Valgrind Project. Massif User Manual. `http://valgrind.org/docs/manual/ms-manual.html`.
32. A.C. Yao. Protocols for secure computations. In *Foundations of Computer Science*, pages 160–164, 1982.
33. A.C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science*, pages 162–167, 1986.
34. P. Yu and P. Schaumont. Secure FPGA circuits using controlled placement and routing. In *CODES+ISSS*, pages 45–50, 2007.

A VHDL examples

In Section A.1 the top level entity of our HMAC implementation is shown and in Section A.2 the top level entity of our AES-128 implementation is shown. Despite having only binary values, the signals are defined as `std_logic` and `std_logic_vector` to allow reuse of existing VHDL code with as little effort as possible. For both implementations, the top level entities are the most interesting ones; all other instantiated entities are pretty much straight forward implementations of their functionalities. Line 58 of the AES-128 example show cases the Δ updating.

A.1 HMAC

```
1  entity hmac is
2  port (
3      padding: in  std_logic_vector( 255 downto 0); -- provided by token
4      auth:    in  std_logic_vector( 255 downto 0); -- provided by token
5                                                    -- 256 = length of one block
6      m:       in  std_logic_vector( 511 downto 0); -- provided by evaluaor;
7                                                    -- 512 = length of one block
8      r:       out std_logic_vector( 255 downto 0) --  result
9  );
10 end hmac;
11
12 architecture structural of hmac is
13
14     component sha256_cf
15     port (
16         stateOld: in  std_logic_vector( 255 downto 0);
17         msg:      in  std_logic_vector( 511 downto 0);
18         stateNew: out std_logic_vector( 255 downto 0)
19     );
20     end component;
21
22     signal   w1:      std_logic_vector( 255 downto 0);
23
24 begin
25
26     w1 <= auth; -- Processing the first block from auth.
27
28     -- The first SHA computation:
29     msgLoop: for i in 0 to <m> generate -- The number of iterations is equal to
30                                         -- the number of message blocks.
31         compF2: sha256_cf port map (
32             stateOld => w1,
33             msg      => m, -- Each time sha_256_rf gets instantiated within the
34                            -- loop, a new block of m will be processed.
35             stateNew => w1 -- Lacking concurrency, this is possible.
36         );
37     end generate msgLoop;
38
39     -- The second SHA computation
40     compF4: sha256_cf port map (
41         stateOld => auth, -- Processing the second block from auth.
42         msg      => w1(255 downto 0) & padding(255 downto 0),
43         stateNew => r
44     );
45
46 end structural;
```

A.2 AES_U1

Note that $B^{\texttt{aes\_rf}}$ also instantiates $B^{\texttt{aes\_ark}}$, $B^{\texttt{aes\_sub}}$ and $B^{\texttt{aes\_mix}}$ illustrating our point about reuse of B^{f_i}.

```
1  entity aes_u1 is
2  port (
3      rk: in  std_logic_vector(127 downto 0); -- unrolled key 11 * 128 bit in total
4      m:  in  std_logic_vector(127 downto 0); -- msg
5      r:  out std_logic_vector(127 downto 0); -- ciphertext
6  );
7  end aes_u1;
8
9  architecture structural of aes_u1 is
10
11     component aes_ark
12     port (
13         stateOld: in  std_logic_vector(127 downto 0);
14         key:      in  std_logic_vector(127 downto 0);
15         stateNew: out std_logic_vector(127 downto 0);
16     );
17     end component;
18
19     component aes_sub
20     port (
21         stateOld: in  std_logic_vector(127 downto 0);
22         stateNew: out std_logic_vector(127 downto 0);
23     );
24     end component;
25
26     component aes_mix
27     port (
28         stateOld: in  std_logic_vector(127 downto 0);
29         stateNew: out std_logic_vector(127 downto 0);
30     );
31     end component;
32
33     component aes_rf
```

```
34      port (
35          stateOld: in  std_logic_vector(127 downto 0);
36          key:      in  std_logic_vector(127 downto 0);
37          stateNew: out std_logic_vector(127 downto 0);
38      );
39      end component;
40
41      signal stateA: Std_logic_vector(127 downto 0);
42      signal stateB: Std_logic_vector(127 downto 0);
43      signal stateC: Std_logic_vector(127 downto 0);
44      signal stateD: Std_logic_vector(127 downto 0);
45
46  begin
47
48      stateA <= m; -- The first and only block of m is being read.
49
50      RF_LOOP: for i in 1 to 5 generate
51      rf_i: aes_rf port map (
52          stateOld => stateA,
53          key      => rk, -- The first 5 blocks of rk are being processed.
54          stateNew => stateA
55      );
56      end generate RF_LOOP;
57
58      stateB  <= stateA; -- The Delta update happens!
59
60      RF_LOOP: for i in 1 to 4 generate
61      rf_i: aes_rf port map (
62          stateOld => stateB,
63          key      => rk, -- The next 4 blocks of rk are being processed.
64          stateNew => stateB
65      );
66      end generate RF_LOOP;
67
68      ark_10: aes_ark port map (
69          stateOld => stateB,
70          key      => rk, -- 10th block of rk.
71          stateNew => stateC
72      );
73      sub_10: aes_sub port map (
74          stateOld => stateC,
75          stateNew => stateD
76      );
77      ark_11: aes_ark port map (
78          stateOld => stateD(127 downto 120) & stateD( 87 downto  80) & stateD( 47 downto  40) &
79                      stateD(  7 downto   0) & stateD( 95 downto  88) & stateD( 55 downto  48) &
80                      stateD( 15 downto   8) & stateD(103 downto  96) & stateD( 63 downto  56) &
81                      stateD( 23 downto  16) & stateD(111 downto 104) & stateD( 71 downto  64) &
82                      stateD( 31 downto  24) & stateD(119 downto 112) & stateD( 79 downto  72) &
83                      stateD( 39 downto  32),
84          key      => rk, -- 11th block of rk.
85          stateNew => r
86      );
87
88  end structural;
```