Verified Proofs of Higher-Order Masking
Overview of an article for the Research Seminar in Cryptography

Kristjan Krips
University of Tartu, Estonia

1 Introduction

This report gives a summary of an article named Verified Proofs of Higher-Order Masking \cite{BBDDFGS15} by Barthe, Belaïdi, Dupressoir, Fouque, Grégoire, and Strub. The paper describes the problem of how to automatically verify the security of higher-order masked programs. Before going to the automatic verification they explain how the formal security of masking can be defined and which models can be used for that. Then they introduce their security model which is going to be used throughout the paper. However, before going into the details, the concept of masking should be explained.

1.1 Motivation

In order to secure a system all parts of the system have to be secured. This means that both the software and hardware have to maintain the level of security. For example the implementation of a block cipher can be formally secure but it is also important that the hardware on which the implementation runs does not compromise the security. The implementation could be attacked via side-channel attacks. Both the computation time and the power used in the computation can leak information about the computation. Thus, the attacker could directly measure the power consumption of a processor while the secret key is being used. This is called Simple Power Analysis (SPA) but fortunately it is easy to prevent by modifying the code, e.g., one way to make the implementation more resistant to SPA is to prevent the secret key from being used in conditional branching operations. In addition to SPA the attacker could compare the power measurements of different program executions. The statistical comparison of such measurements is named Differential Power Analysis (DPA) and it can reveal the variable values including the value of the secret key. E.g., if the secret key is used in an XOR operation then the value of the operation depends on the secret key and thus the power consumption also depends on the value of the secret key. DPA has been used to break the several commercial embedded devices \cite{BGV12,MBKP11}. For more information about SPA and DPA, see \cite{KJJ99}. In addition to SPA and DPA the attacker could get physical access to the circuit by placing metal needles on the wires of the circuit and thus reading the values of the corresponding wires. This attack is called probing attack and it is described in \cite{ISW03}. 
Preventing DPA and other side channel attacks. There are several techniques for making the DPA attack more difficult. One way to prevent DPA is to physically shield the hardware but this could be a suitable solution only in a few rare cases. A better solution would modify the source code of the program such that the DPA attack or probing attack would be impossible or infeasible. Thus, the goal is to modify the implementation such that power consumption would leak less information and would increase the complexity of the attack and thus make it impractical. Chari, et al. proposed to hide bits of the computation by using secret sharing [CJRR99]. One idea for secret sharing a bit is to use uniformly random values that are XOR-ed with the initial bit. The XOR operation is denoted with the symbol $\oplus$. E.g., there are k shares $b \oplus r_1, r_2, \ldots, r_{k-1}, r_1 \oplus r_2 \ldots r_{k-1}$, where the last share $r_1 \oplus r_2 \ldots r_{k-1}$ is the XOR of the used random variables. Now, when all shares are XOR-ed then the result is the initial bit. The initial idea can also be generalised to hide values of bytes. It is important to note that after secret sharing the operations are done on the shares and these operations should not leak information. The idea behind using secret sharing to hide the secret value is to remove the statistical dependency between the secret value and the used power. When only one random value is used to mask the secret value then it is called first-order masking. In the case of masking the secret value with d random values, the masking scheme is called d-th order masking. A masking scheme that uses more than one random value is called a higher-order masking scheme. If the attacker would be able to analyse d+1 intermediate variables of a d-th order masking scheme then it would be possible to find the secret value. However, this kind of an attack becomes impractical when the masking order increases. Chari et al. did the first formal security analysis of higher-order masking schemes in a noisy leakage model and found that attacking a higher order masking scheme requires an exponential number of queries in the masking order [CJRR99]. Still, there have been attacks against higher-order masking schemes and some of the implementations have been proven to be insecure [OM06].

1.2 Leakage Models

Noisy leakage model. The notion of a noisy leakage model was used by Chari et al. in [CJRR99]. In the noisy leakage model the attacker has access to the noisy leakage channels. Thus, the attacker gets access to the values sampled according to Gaussian distribution such that the values are centred around the real values. However, in this model the adversary is only allowed to make one-bit observations without taking the computation into account. The noisy leakage model is close to the reality as the attacker does not gain access to the actual values but instead gets access to a noisy Hamming weight of a variable or the Hamming distance between two values of a variable.

T-threshold probing model. The t-threshold probing model was proposed in [ISW03]. In this model the adversary has access to the exact values of at most t internal variables. This model is not so realistic as in real world the leakages are noisy and it is not possible to get access to the exact values. This model also
differs from the noisily leakage model by the number of observations the attacker is allowed to make. In the case of t-th order masking the adversary should not learn anything about the secret value from any combination of t internal variable values.

Noisy leakage model by Prouff and Rivain. The initial noisily leakage model had several restrictions which made the model unrealistic as the attacker is able to observe several bits at a time and also observe the computations. Thus, Prouff and Rivain extended the initial model by allowing leakage distributions that differ from the Gaussian distribution [PR13]. In addition, they allow the attacker to observe any intermediate variable and also the computation. In their model each computation step gives a leakage function \( f(x) \) to the attacker and this function depends only on part of the state \( x \). Prouff and Rivain model the noise function by bounding the bias in the distribution of part of the state \( x \) given \( f(x) \). Thus, the statistical distance between distributions \( \Pr[x] \) and \( \Pr[x|f(x)] \) is bounded by a security parameter \( \omega \). Although their work gave the first security proof for a block cipher in a low-level security model, it can not be used in standard reductionist security proofs as their notion of security is information theoretic. In order to overcome this problem the security in noisy leakage model of Prouff and Rivain was reduced to be security in the t-threshold probing model [DDF14].

Transition-based Model. Both the noisy leakage model and the t-threshold probing model only consider the values of intermediate variables but this does not always capture the real-world scenarios where leakages can be caused by transitions at the gate level (glitches) [BGG+14]. Thus, it might be possible that the adversary gains access both to the old value of a variable and a new value of the same variable. Balash et al. describe that these kind of leaks can lower the security by the order of two.

T-threshold probing model with t-non interference The authors of the paper [BBD+15] modify the security of the t-threshold probing model by using the t-non interference notion that is commonly used in language based security. The non-interference property means that the attacker should not be able to distinguish two computations from the outputs if the computations vary only by the secret inputs. But before giving a more detailed description of the t-non interference the notion of program equivalence has to be defined. The authors of [BBD+15] define program equivalence by saying that two probabilistic programs are \((\mathcal{I}, \mathcal{O})\)-equivalent, denoted \( p_1 \sim_{\mathcal{O}} p_2 \), where \( \mathcal{I} \) denotes the assumptions on input variables and \( \mathcal{O} \) denotes the observed variables, whenever the conditional probability distributions on \( \mathcal{O} \) defined by \( p_1 \) and \( p_2 \) are equal. This equivalence notion is used by the authors as it subsumes both the functional equivalence and the t-non-interference. More precisely, two programs \( p \) and \( \bar{p} \) are functionally equivalent when they are \((\mathcal{I}, \mathcal{Z})\)-equivalent, where \( \mathcal{Z} \) denotes output variables and \( \mathcal{I} \) denotes input variables. Similarly the t-non-interference can be described
by the given equivalence. The authors of [BBD+15] describe that a program $\bar{p}$
is $t$-non-interfering with respect to a set of secret input variables and a set of observable variables $O$ when $\bar{p}(s_0, \cdot)$ and $\bar{p}(s_1, \cdot)$ are $(I, O)$-equivalent for any value of the secret input variables $s_0$ and $s_1$, where the set $I$ denotes the non-secret input variables.

Now there are enough tools to give the modified definition of the $t$-threshold probing model that is based on indistinguishability. The authors of [BBD+15] propose a model, where the challenger randomly chooses two secret values $s_0$ and $s_1$ and a bit $b$ which determines the type of the leakage. The computation always uses the value $s_0$ but the adversary is given leaks that are produced by $s_0$. The adversary is allowed to adaptively query an oracle with public arguments and a set of at most $t$ intermediate variables. These queries by the adversary reveal the outputs and the values of intermediate variables that were chosen by the adversary. The task of an adversary is to guess if the leaks were produced with $s_0$ or $s_1$, i.e., the adversary has guess the bit $b$.

**Theorem 1.** Let $p$ and $\bar{p}$ be two programs. If $p$ and $\bar{p}$ are functionally equivalent and $\bar{p}$ is $t$-non-interfering, then for every adversary $A$ against $\bar{p}$ in the $t$-threshold probing model, there exists an adversary $S$ against $p$ in the black box model, such that $\Delta(S \overset{bb}{\Rightarrow} p, A \overset{thr}{\Rightarrow} \bar{p}) = 0$, where $\Delta(\cdot; \cdot)$ denotes the statistical distance.

**Proof.** The previous theorem says that $p$ and $\bar{p}$ are functionally equivalent and thus $\Delta(S \overset{bb}{\Rightarrow} p, S \overset{thr}{\Rightarrow} \bar{p}) = 0$. Therefore, one has to show that $\Delta(S \overset{bb}{\Rightarrow} \bar{p}, A \overset{thr}{\Rightarrow} \bar{p}) = 0$ holds. When a simulator is created then it only gets the public variables of $\bar{p}$ and the output of $\bar{p}$. It is important to note that the simulator does not get the $t$ intermediate values that correspond to the observation set $O$. We know that the program $\bar{p}$ is $t$-non-interfering, i.e., the observations do not depend on the secret variables that are used for the execution of $\bar{p}$. Therefore, the simulator can choose arbitrary secret variables and run the program $\bar{p}$ with these values and the given public input variables while returning the outputs of the observations found during the execution of $\bar{p}$.

One of the main results of the paper [BBD+15] is the proposal of algorithms that prove functional equivalence and $t$-non interference properties of probabilistic programs as this reduces the security of masked implementations in the $t$-threshold probing model to the black-box security of the algorithms they implement. These algorithms were implemented using EasyCrypt [BDG+14,BGHB11a] and were able to successfully analyse both the first-order masked implementations of AES and 2 rounds of second-order masked implementations of AES.

The next section describes how it would be possible to verify the security of higher-order masked implementations. This is not a trivial task as one has to consider every combination of $t$ out of $n$ internal variables which makes the task exponential.
2 Language-based techniques for threshold security in
the probing model

This section describes techniques that can be used to verify the assumptions
of Theorem 1. Before verifying the assumptions of Theorem 1, the assumptions
have to be formalised and this can be done by using information flow checking
and equivalence checking.

2.1 Problem statement

The hypotheses of Theorem 1 state that two programs are functionally equiv-
alent and one of these programs is t-non-interfering. These hypotheses can be
viewed as variants of equivalence checking and information flow checking. In
the standard setting equivalence checking works on deterministic programs but
in this case equivalence checking has to be applied on probabilistic programs.
Information flow checking usually checks information flows from secret inputs
to public outputs but in the context of Theorem 1 the information flow check-
ing should check flows from secret inputs to intermediate values. Both of these
problems can be seen as instances of relational verification problems. In order to
simplify the description of the verification problems they consider straight-line
code programs which consist of random assignments and deterministic assign-
ments and have distinguished sets of input variables and output variables. For a
given program \( p \) the set of input variables are denoted by \( \text{IVar}(p) \), the set out-
put variables are denoted by \( \text{OVar}(p) \) and the set of intermediate variables are
denoted by \( \text{PVar}(p) \). The set of intermediate variables \( \text{PVar}(p) \) is divided into a
set of deterministic intermediate variables \( \text{DVar}(p) \) and into a set of probabilis-
tic intermediate variables \( \text{RVar}(p) \). In order to further simplify the description
they assume that programs are in the single static assignment (SSA) form, i.e.,
that the variables are evaluated only once. The SSA form is realistic as one can
transform any straight-line program into SSA form. Let \( \mathcal{V} \) denote the set of pro-
gram values, i.e., the values of variables used in the program. The authors of
\cite{BBD+15} state then each program \( p \) can be interpreted as a function:

\[
[p] : \mathcal{D}(\mathcal{V}^\kappa) \rightarrow \mathcal{D}(\mathcal{V}^{\ell + \ell'})
\]

where \( \mathcal{D}(T) \) denotes the set of discrete distributions over a set \( T \), and \( \kappa \) and \( \ell \)
and \( \ell' \) respectively denote the sizes of sets \( \text{IVar}(p) \), \( \text{PVar}(p) \) and \( \text{OVar}(p) \). The described
function takes as input a joint distribution on input variables and returns a joint
distribution on all program variables. They also define a function for every subset \( \mathcal{O} \) of \( \text{PVar}(p) \) of size \( m \) (i.e., for every subset of the set of intermediate variables in the program):

\[
[p]_\mathcal{O} : \mathcal{D}(\mathcal{V}^\kappa) \rightarrow \mathcal{D}(\mathcal{V}^m)
\]

that computes for every element \( v \) of \( \mathcal{V}^\kappa \) the marginal distribution of \( [p](v) \) with
respect to \( \mathcal{O} \).
Now there is enough background information to formally state the information flow checking problem. The authors state this in the following way: a program $p$ is non-interfering with respect to a partial equivalence relation $\phi \subseteq \mathcal{D}(\mathcal{V}^\phi) \rightarrow \mathcal{D}(\mathcal{V}^\phi)$, and a set $\mathcal{O} \subseteq \text{PVar}(p)$, or more precisely a program is $(\phi, \mathcal{O})$-non-interfering, iff $\llbracket p \rrbracket_{\mathcal{O}}(\mu_1) = \llbracket p \rrbracket_{\mathcal{O}}(\mu_2)$ for every $\mu_1, \mu_2 \in \mathcal{D}(\mathcal{V}^\phi)$ such that $\phi \mu_1 \sim \phi \mu_2$. This can be denoted as $\text{NI}_{\phi,\mathcal{O}}(p)$. In addition, let $\emptyset$ be a set of subsets of the intermediate variables of the program. Then one can say that program $p$ is $(\phi, \emptyset)$-non-interfering, if it is $(\phi, \mathcal{O})$-non-interfering for every subset $\mathcal{O} \in \emptyset$.

The lengthy formalisation of the information flow checking problem might create questions about the meaning of $\phi$. In the standard deterministic setting $\phi$ denotes low equivalence, which means that two tuples of values $v_1$ and $v_2$ coincide on public variables. It would be useful to transform the notion of low equivalence into the probabilistic setting as the Theorem 1 deals with probabilistic programs. In the probabilistic setting two distributions $\mu_1, \mu_2$ are low equivalent iff their marginal distributions with respect to public variables are equal.

Now the t-threshold probing security can be stated using the non-interference property. The authors state it in the following way: a program $p$ is $(\phi, t)$-non-interfering if it is $(\phi, \mathcal{O})$-non-interfering for all subsets $\mathcal{O}$ of $\text{PVar}(p)$ with size smaller than $t$. If the previous condition holds then a program $p$ is secure in the t-threshold probing model iff it is $(\phi, t)$-non-interfering. Using this property, it is possible to state the t-threshold security also in the transition-based model. In order to do that the authors use a partial function named $\text{next}$ that maps program variables to their successors according to the way it is done in the SSA form. Using the $\text{next}$ function it is possible to state that a program $p$ is $(\phi, t)$-non-interfering in the transition-based model. This can be written as $\text{NI}_{\phi, t, \text{succ}}(p)$ iff program $p$ is $(\phi, \mathcal{O} \cup \text{next}(\mathcal{O}))$-non-interfering for every subset of the set of intermediate variables of size smaller than $t$. Then it is possible to say that a program $p$ is secure in the transition-based t-threshold probing model iff it is $(\phi, t)$-non-interfering in the transition-based model.

The next step shows how it is possible to state program equivalence using the previously described program interpretation as a function $\llbracket p \rrbracket$. Let two programs $p_1$, $p_2$ have the same input and output set. In this case the output set is denoted with $\mathcal{W}$ and let $\llbracket p \rrbracket_{\mathcal{W}}$ denote a function that computes the marginal distribution of $\llbracket p \rrbracket(\mu)$ for every input distribution with respect to $\mathcal{W}$. Then the program equivalence can be stated in the following way: programs $p_1$, $p_2$ are equivalent with respect to $\phi \subseteq \mathcal{D}(\mathcal{V}^\phi) \rightarrow \mathcal{D}(\mathcal{V}^\phi)$ iff $\llbracket p_1 \rrbracket_{\mathcal{W}}(\mu) = \llbracket p_2 \rrbracket_{\mathcal{W}}(\mu)$ for every distribution $\mu$ such that $\phi \mu \sim \mu$. This kind of a program equivalence between $p_1$ and $p_2$ can be denoted by $p_1 \sim p_2$. In addition, the given notion can be described with the $(\phi, \mathcal{O})$-equivalence. The authors state that two programs $p_1$, $p_2$ are $(\phi, \mathcal{O})$-equivalent, i.e., $p_1 \sim_{\phi}^\mathcal{O} p_2$, iff $\llbracket p_1 \rrbracket_{\mathcal{O}}(\mu_1) = \llbracket p_2 \rrbracket_{\mathcal{O}}(\mu_2)$ for all distributions $\mu_1, \mu_2$, such that $\phi \mu_1 \sim \phi \mu_2$. This means that both the equivalence checking and information flow checking can be implemented with the help of an algorithm that verifies $p_1 \sim_{\phi}^\mathcal{O} p_2$. 
2.2 Approaches to proving the validity of masking countermeasures

Certified compilation vs certifying compilation. The authors of [BBD+15] claim that there are two approaches to proving the validity of masking countermeasures. The first idea comes from certified compilation [Ler06] and this approach would take an initial program and transform it into a program that is secure in a given masking order. Such compiler would get as an input the program code and would output a program that is secure in the given masking order. In addition, the compiler would output a proof to show that the transformed program is secure in the given masking order. The compiler could also output transformations that transforms the initial program into $t$ different masking orders. Such compiler could prove that the initial program and the transformed program are observationally equivalent and that the transformed program is $t$-non-interfering if the masking order is $t$. However, in order for the compiler to output a formal proof the compiler would have to use proof assistants. This solution would mean that a security proof would have to be created only once for a given program.

The second approach for proving the validity of masking countermeasures is described in [NL98] and is named certifying compilation. In this case the task is to automatically prove that the initial program and the transformed program are equivalent and that the transformed program is non-interfering. This means that the automatic prover should get as an input both the initial program and the transformed program that is secure in the given masking order. It is important to note that the transformed program is not generated by the automatic prover. Certifying compilation or translation validation can take extra inputs that simplify the verification process, e.g., loop invariants could be given as an additional input. The advantage of certifying compilation is that it does not set assumption to the generation of the transformed program. In addition, the authors claim that this approach is more flexible and easier to put in practice and therefore they chose to use this approach.

Type-based approaches. Non-interference could also be enforced by using information flow type systems. These systems track dependencies between program variables and report when illegal flows are found. In the case of deterministic programs each program variable is given a security level such that public variables and secret variables have different levels. The same idea can be generalised to probabilistic programs and in this case the information flow type systems distinguish uniformly distributed values from public values and secret values. Information flow systems were applied to masking by Moss et al. in [MOPT12], in which their solution transformed the initial program into a masked program that is secure against first-order DPA. The problem with their solution is that the information carried by the types is attached to individual values and this makes it difficult to design information flow type systems for higher order maskings. The authors propose a workaround that would allow to use information type systems with higher order DPA but they also state that such systems would grow exponentially as the resistance order grows.
Model counting. Model counting computes the number of valid assignments to a logical formula in a first order theory. Model counting could be applied to prove the equivalence of probabilistic program as it is done in [BGHB11b]. There have been attempts to use model counting in order to verify the security of implementations and to create masked implementations but these solutions only work in the case of low masking orders.

Relational verification. Another method for proving non-interference and program equivalence is to use program verification. The program verification would have to consider two programs or two executions of the same program and therefore relational program logic would be useful in this setting. It would be useful if the relational program logic would work with probabilistic programs. In [BGZB09] probabilistic Relational Hoare Logic (pRHL) is introduced. The \((\phi, O)\)-non-interference of a program could be stated with a pRHL judgement in a following way:

\[
\{\phi\} p \sim p\{\bigwedge_{y \in O} y(1) = y(2)\}
\]

This pRHL judgement states that the values of variables y are the same in any two executions if the executions start from the initial memories that are related by \(\phi\). Although the authors show that it is possible to use pRHL in order to check non-interference they also claim that this approach is not practical for checking the security in the t-threshold probing model. Therefore, the authors present a set of algorithms that can verify the security of programs in the t-threshold probing model. A brief overview of these algorithms is given in the next section.

3 Algorithms proposed in the paper

In order to prove the probabilistic non-interference the authors proposed a new logic that can be used to show that a vector or probabilistic expressions is independent of a secret variable. In addition, they describe algorithms that construct derivations in the new logic. As the proof of non-interference can be reduced to the proving the relational judgements, they show that instead of proving these judgements it is possible to construct derivations that reach to the same result. The technical details of the proposed logic and algorithms that the authors created to derive judgements can be read from chapter 3 in the original paper [BBD+15].

It is a non-trivial task to prove that a program is t-non-interfering. The authors of the paper proposed divide-and-conquer algorithms that decrease the number of checks required for verifying the t-non-interference. The problem is that the proving of t-non-interference requires the proving of non-interference for all observation sets. As the problem of proving the non-interference for all observation sets is an exponential problem, it means that the proposed algorithms have scalability limitations. The proposed algorithms are efficient and
worked well during the testing phase but this does not fully solve the scalability problem. One way to resolve the scalability problem is to develop compositional techniques like it is done in the context of masking compilers. Detailed information about the proposed algorithms can be found in chapter 4 of the original paper \cite{BBBD+15}.

4 Proving Functional Correctness

As a last step they show how to prove the functional equivalence of masked programs with the corresponding unmasked versions. It is easier to prove the functional correctness as it is a compositional property and thus the correctness of small components can be proven first and then used as building blocks for creating masked functions. The correctness of the larger masked functions can be proven with the help of EasyCrypt. EasyCrypt can replace the calls to the smaller procedures by the corresponding code by inlining the procedures. When all of the procedures calls are replaced with their functional specifications it should be possible to get a functional correctness theorem for the complex operation. It has been proposed in \cite{DDF14} that a compiler could be created that would transform any cryptographic scheme secure in the "black-box" model, i.e., in a model where the internal structure is not known, into a program secure in the noisy leakage model. This kind of a compiler could be extended to include a code generator that creates proofs for masked programs and thus the masked programs would not have to be certified.

5 Conclusion

The article studies the problem of verifying the security of higher order masking implementations. As the complexity of verifying the security of a t-th order masking grows exponentially when the order increases, it has been mostly neglected in the previous works. This paper shows that it is feasible to verify the security of masked implementations against t-th order DPA attacks. The authors propose as future work to prove compositional properties in order to be able to verify the security of masked implementations at higher orders.

References


Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Emmanuel Prouff and Patrick Schaumont, editors, Cryptographic Hardware and Embedded Systems CHES 2012, volume

