Cryptographic Password Obfuscation

Giovanni Di Crescenzo(✉), Lisa Bahler, and Brian Coan

Perspecta Labs, Basking Ridge, NJ 07920, USA
{gdicrescenzo,lbahler,bcoan}@perspectalabs.com

Abstract. We consider cryptographic program obfuscation of point functions (i.e., functions parameterized by a secret s that, on input a string x, return 1 if $x = s$ and 0 otherwise). We achieve the following notable results: (1) the first *efficient* point function obfuscator for arbitrarily large as well as very short secrets, provable *without random oracle assumptions*; (2) the first efficient and provably-secure (under the existence of one-way permutations or block ciphers that have no theoretical attack faster than exhaustive key search) real-life applications built on top of these obfuscators, such as: (a) entity authentication via password verification; (b) entity authentication via passphrase verification; and (c) password management for multi-site entity authentication.

Keywords: Program obfuscation · Password authentication

1 Introduction

Program obfuscation is the problem of modifying a computer program so to hide sensitive details of its code without changing its input/output behavior. While this problem has been known for several years in computer science, only in the last 15 years, researchers have considered the problem of provable program obfuscation; that is, the problem of program obfuscation, where sensitive code details are proved to remain hidden under a widely accepted intractability assumption (such as those often used in applied cryptography). Early results in the area implied the likely impossibility of constructing a single program capable of obfuscating an arbitrary polynomial-time program into a virtual black box [3]. Moreover, most recent results show the possibility of constructing different obfuscators for restricted families of functions, such as point functions (and extensions of them), under more or less accepted hardness assumptions (see, e.g., [2,5,10,12,16,19]). Point functions can be seen as functions that return 1 if the

Supported by the Defense Advanced Research Projects Agency (DARPA) via U.S. Army Research Office (ARO), contract number W911NF-15-C-0233. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, ARO or the U.S. Government.

© Springer Nature Switzerland AG 2018
D. Naccache et al. (Eds.): ICICS 2018, LNCS 11149, pp. 497–512, 2018.
https://doi.org/10.1007/978-3-030-01950-1_29

input value is equal to a secret value stored in the program, and 0 otherwise. Although conceptually simple, point functions come with surprisingly interesting applicability to real-life problems. As often suggested in the literature (see, e.g., [16,19]), point function obfuscation might be applicable to the password verification function in very commonly used login/password authentication methods.

In this paper we carry out an exploration of this suggestion. Our main result is a *practical* method for cryptographic password obfuscation *under standard cryptographic assumptions* (and, specifically, without using a random oracle assumption or a heuristic construction for a multilinear map), such as the existence of a one-way permutation or, of a block cipher for which there are no theoretical attacks faster than exhaustive key search. Known practical methods include the well-known password hashing method (i.e., at registration, server stores the cryptographic hash of the user's password; at authentication, server checks that hash of the provided password is equal to the stored hash). This method was analyzed in a cryptographic program obfuscation context by [16], but is however only proved secure under the random oracle assumption. (Note that this assumption, although often accepted by practitioners, has been declared as almost certainly false in its generality [9], and is especially troublesome in light of the less and more recent attacks to widely considered or used hash functions such as MD4, MD5 and SHA1). By now, there are several known obfuscators for point functions that do not make the random oracle assumption, but they all assume secrets much longer than typical passwords. The only exception and the closer result in the literature to ours is an elegant construction of a perfectly one-way function from [10], which could be used to construct a point function obfuscator under the assumption of claw-free permutations. We note that this construction is not practical as it is estimated 4–5 orders of magnitude less efficient than the one we propose.

As all point function obfuscators in the literature use secrets of length about equal to the factoring-type security parameter (e.g., 2048), to increase capability to commonly used passwords and passphrases as well as secrets of arbitrary lengths, we have designed two new methods: (a) a new hash function that transforms these point function obfuscators in the literature so that they can work with *arbitrarily longer secrets*; (b) a new (multi-bit-output) point function obfuscator, which can work with secrets *as short as the symmetric cryptography security parameter* (e.g., 128). Our obfuscators satisfy a computational notion of functional correctness (i.e., no efficient adversary can find an input on which the obfuscated program differs from the original program), and a rather strong notion of obfuscation security (i.e., the obfuscated program is efficiently simulatable). An underlying technical contribution is the construction of an *efficient second-preimage-resistant extractor* that is simultaneously a second-preimage-resistant hash function, a pairwise almost-independent hash function, and has efficient instantiations from a single efficient cryptographic primitive. Our efficiency claims on this extractor and its resulting obfuscators are substantiated by implementations and performance results on commodity computing resources. Finally, we demonstrate that program obfuscators for point functions are usable in the following real-life applications: *password verification* (obfuscating a server's

algorithm verifying if a client's input string is equal to the client's previously registered password), *passphrase verification* (as for password verification, but with the variant that the client has registered a passphrase containing only structured text); and *password manager* (obfuscating a server's verification and retrieval algorithms that verify the client's master password or passphrase, and retrieve a client's previously registered password for a specific server).

In particular, we have modified code on an open-source password manager (Pass, based on gpg2) to accommodate our (multi-bit-output) point function obfuscator instead of their current cryptographic solution (whose obfuscation properties can at best be proved using a random oracle assumption). The overall resulting runtime of a specific password retrieval on the modified application is less than 3% slower than the same operation on unmodified Pass. Solving these password and passphrase obfuscation problems without using a random oracle assumption are natural problems that have remained unsolved for decades. Details of our real-life application results are discussed in Appendix A.

2 Definitions and Preliminaries

In this section we first recall basic definitions and slightly modify the theory-oriented definition of program obfuscators into a practice-oriented definition that better fits a large class of obfuscator implementations (including ours for point functions) and where the correctness property only holds in a computational sense (i.e., even against a possibly malicious polynomial-time adversary). Finally, we discuss security notions for point function obfuscators.

Security Parameters. In our constructions and concrete security analysis, will use two types of *security parameters*, described below:

1. the *'factoring-based' security parameter*, a global parameter λ_f, currently set to 2048, that is typically used to determine key lengths in asymmetric cryptographic primitives (e.g., public-key encryption) proved secure under the hardness of number theoretic problems related to factoring and/or discrete logarithm problem; and
2. the *'symmetric-cryptography' security parameter*, a global parameter λ_s, currently set to 128, that is most typically used to determine key/output lengths in several symmetric cryptographic primitives (e.g., block ciphers).

Point Functions. We consider *families of functions* as families of maps from a domain to a range, where maps are parameterized by some values chosen according to some distribution on a parameter set. Let pF be a family of functions $f_{par} : Dom \to \{0,1\}$, where $Dom = \{0,1\}^n$, and each function is parameterized by value par from a parameter set $Par = \{0,1\}^n$, for some *length parameter* n. We say that pF is the *family of point functions* if on input $x \in Dom$, and *secret value* $y \in Par$, the point function f_y returns 1 if $x = y$ and 0 otherwise. When we assume an efficiently samplable distribution of secret values $y \in Par$, we define the (rounded) *min entropy parameter* of pF as the largest integer t

such that each $y \in Par$ is sampled with probability $\leq 2^{-t}$. The *family mbpF of multi-bit-output point functions* is defined as follows: on input $x \in Dom$, secret value $y \in Par$, and output value z, the function f_y returns z if $x = y$ and 0 otherwise.

Program Obfuscators: Formal Definitions. To define a cryptographic program obfuscator for a class of functions F, we consider a pair of efficient algorithms with the following syntax. On input function parameters $fpar$, including a description $desc(f)$ of function $f \in F$, the *obfuscation generator genO* returns generator output $gpar$. On input a description $desc(f)$ of function $f \in F$, generator output $gpar$, and evaluator input x, the *obfuscation evaluator evalO* returns evaluator output y.

Informally, we would like to define an obfuscator for the class pF of point functions as any such pair of algorithms satisfying some *functionality correctness* property (i.e., the obfuscated program computes the same function as the original program), some *efficiency* property (i.e., the obfuscated program is not much slower than the original program), and some *obfuscation security* property (i.e., the obfuscated program hides any sensitive information about the original program which is not computable by program evaluation). Here, we actually consider a slightly relaxed notion of the functionality correctness property, according to which the obfuscated program can return an output different from the original program for some of the inputs; however, these inputs are hard to find, even to an efficient algorithm that has access to the program's secret value. Furthermore, we discuss some of the security notions in the literature, and eventually formally define the strongest known notion (implicit in [3] and saying, informally speaking, that any efficient adversary's view of the obfuscated program can be efficiently simulated and thus the adversary learns nothing more than an upper bound on the program size), specialized to the class of point functions pF with secret distributions having high min-entropy. We now proceed more formally.

We say that the pair $(genO, evalO)$ is a *cryptographic program obfuscator* for the class pF of point functions if it satisfies the following:

1. (Computational correctness): For any f_s in pF, with function parameters $fpar = (s, desc(pF))$, and any efficient algorithm A, the event $f_s(x') \neq y$ holds with probability δ, for some negligible (or very small) δ, where x', y are generated by the following probabilistic steps:
   - $gpar \leftarrow genO(fpar)$,
   - $x' \leftarrow A(gpar, fpar)$,
   - $y \leftarrow evalO(gpar, x')$.
2. (Polynomial Blowup Efficiency): There exists a polynomial p such that for all f_s in pF, the running time of $evalO(gpar, \cdot)$ is $\leq p(|f_s|)$, where $|f_s|$ denotes the size of the (smallest) boolean circuit computing f_s.
3. (Adversary view simulation security): Given any high min-entropy distribution D returning an n-bit secret, there exists a polynomial-time algorithm Sim such that for any function f_s, $|s| = n$, in the class pF of point functions, with black-box access to f_s such that for all f_s in pF with parameters $fpar$, the distributions D_{view} and D_{sim} are computationally indistinguishable, where

- $D_{view} = \{s \leftarrow D; gout \leftarrow genO(s, desc(pF)) : gout\}$,
- $D_{sim} = \{s \leftarrow D; gout \leftarrow Sim(1^{|s|}, desc(pF)) : gout\}$.

Other security notions considered in the literature include *adversary output black-box simulation* (where the simulator has also access to a black-box computing the program [3] and targets simulating the adversary's output bit), *real-vs-random indistinguishability* (where no efficient adversary can distinguish the obfuscation of the function f from an obfuscation of a random function in the class F) [5], and *indistinguishability obfuscation* (where no efficient adversary can distinguish the obfuscation of any two circuits computing the same function f) [3]. We note that an obfuscator satisfying the adversary view black-box simulation security notion also satisfies these latter 3 security notions.

Known Point Function Obfuscators. The obfuscator in [16] for the family of point functions satisfies adversary view black-box simulation under the random oracle assumption. This obfuscator essentially consists of computing a cryptographic hash of the secret, similarly as typically done for passwords in real-life systems. A previous result of [7], although formulated as a oracle hashing scheme, can be restated as an obfuscator satisfying a strong variant of real-vs-random indistinguishability under the Decisional Diffie Hellman assumption. The obfuscator in [19] satisfies adversary output black-box simulation under the existence of a strong type of one-way permutations. Moreover, one of the obfuscators in [5], based on any deterministic encryption scheme, satisfies real-vs-random indistinguishability, and has several instantiations. This follows as deterministic encryption schemes can be built using the hardness of the learning with rounding problem [20] or the existence of lossy trapdoor functions [6], and the latter have been built using any one of many group-theoretic assumptions (see, e.g., [13]). Some of the resulting obfuscators have efficient implementations [12]. Finally, an obfuscator was given in [2] using the hardness of the learning with error problem.

All results mentioned so far either make the random oracle assumption or work for secret distributions not significantly different than uniform. The only obfuscator working for arbitrary secret distributions of high min-entropy can be obtained using a result from [10] on perfectly one-way functions, constructed assuming the existence of claw-free permutations. This result is far from having an efficient implementation.

Our goal in the rest of this paper is to show an obfuscator for point functions that works for arbitrary secret distributions of high min-entropy, without making the random oracle assumption, and resulting in an efficient implementation.

Families of One-Way α -Permutations. The term *efficient* is used for running time in both a practical and theoretical sense, as needed. We say that a family of functions $\{F\}$ is *efficiently samplable* if there exists an efficient algorithm randomly choosing a function F from the family, and is *efficiently computable* if there exists an efficient algorithm that evaluates any function F from the family. We say that a family of functions $\{F\}$, is a family of α *-permutations* if the probability that, for a randomly chosen x, $F(x)$ has > 1 preimages, is $< \alpha$.

Families of Pairwise-Independent Hash Functions. We say that a family of hash functions $\{H_{m,n}\}$, where $H_{m,n} : \{0,1\}^m \rightarrow \{0,1\}^n$ is *pairwise δ -independent* if for any $x_0 \neq x_1 \in \{0,1\}^m$, and any $y_0, y_1 \in \{0,1\}^n$, it holds that $\mathrm{Prob}[\, H(x_0) = y_0 \wedge H(x_1) = y_1 \,] \leq \delta + 2^{-2n}$. We say that family $\{H_{m,n}\}$ is *pairwise independent* if it is is pairwise δ-independent, for $\delta = 0$. Constructions for pairwise-independent hash functions include a random one-degree polynomial in a Galois field or a random one-degree polynomial modulo a prime [11], where by a random polynomial we denote a polynomial with coefficients randomly chosen in their domain set. All these constructions are efficiently sampleable and efficiently computable.

Families of Second-Preimage-Resistant Hash Functions. This cryptographic primitive was introduced in [17], under the name of universal one-way hash functions, and have also been called target-collision-resistant hash functions since [4] or second-preimage-resistant hash functions. We say that a family of functions $\{h \,|\, h : \{0,1\}^a \rightarrow \{0,1\}^b\}$ is *second-preimage-resistant* over $\{0,1\}^a$ if it satisfies the following three properties: (1) h is efficiently sampleable from its family; (2) every h in the family is efficiently computable; and (3) no efficient adversary can win, except with very small probability, in the following game: first, the adversary picks an input z, then a random function h is sampled from its family; finally, the adversary, given $h(z)$, wins the game if it finds an input x such that $h(x) = h(z)$. The first constructions for such families of functions were proposed in [17], based on families of one-way permutations with varying domain sizes and any family of pairwise-independent hash functions. Later, more practical constructions were proposed in [4,18], based on collision-intractable hash functions. Generally speaking, second-preimage-resistant hash functions may or may not satisfy pairwise-independence properties.

Randomness Extractors. The *statistical distance* between two distributions D_1, D_2 over the same space S is defined as $sd(D_1, D_2) = \frac{1}{2} \Sigma_{x \in S} \,|\, \mathrm{Prob}[\, x \leftarrow D_1 \,] - \mathrm{Prob}[\, x \leftarrow D_2 \,] \,|$. We say that distributions D_1, D_2 are *δ-close* if it holds that $sd(D_1, D_2) \leq \delta$. We say that a distribution D is *δ -close to uniform*, or, briefly, *δ-uniform*, if it holds that $sd(D, U) \leq \delta$, where U denotes the uniform distribution over the same space S. The *min-entropy* of a distribution D is defined as $H_\infty(D) = \min_x \{-\log_2(\mathrm{Prob}[\, x \leftarrow D \,])\}$. A function Ext: $\{0,1\}^a \times \{0,1\}^b \rightarrow \{0,1\}^c$ is called a *(k, ϵ)-extractor* if for any distribution D on $\{0,1\}^a$ with min-entropy at least k, the distribution $N(D)$ is ϵ-uniform, where $N(D) = \{x \leftarrow D; e \leftarrow \{0,1\}^b; y = \mathrm{Ext}(x, e) \,:\, (e, y)\}$. The *leftover hash lemma* [14] says that if $\{H_{m,n}\}$ is a family of pairwise-independent hash functions, value x is drawn according to a distribution D such that $H_\infty(D) \geq k$, and $n \geq k - 2\log(1/\epsilon)$, then the function $\mathrm{Ext}(x, H_{m,n})$ defined as $y = \mathrm{Ext}(x, H_{m,n}) = H_{m,n}(x)$ is a (k, ϵ)-extractor. By inspection of the proof in [15], we see that it can be directly extended to families of pairwise δ-independent hash functions, as follows.

Lemma 1. *For any $\delta > 0$, if $\{H_{m,n}\}$ is a family of pairwise δ-independent hash functions, value x is drawn according to a distribution D such that $H_\infty(D) \geq k$, and $n \leq k - 2\log(1/\epsilon)$, for some $\epsilon \leq (1/2)\log(1/\delta)$, then the function $Ext(x, H_{m,n})$ defined as $y = Ext(x, H_{m,n}) = H_{m,n}(x)$ is a (k, ϵ)-extractor.*

We say that a function Ext: $\{0,1\}^a \times \{0,1\}^b \rightarrow \{0,1\}^c$ is a *second-preimage-resistant (k, ϵ)-extractor* if it is both a second-preimage-resistant hash function over $\{0,1\}^a$ and a (k, ϵ)-extractor.

3 An Efficient Second-Preimage-Resistant Extractor

In this section we construct an efficient second-preimage-resistant extractor, or actually a family of hash functions which satisfies the following desirable combination of functionality, efficiency and security properties:

1. it achieves arbitrarily large compression, in that it maps an arbitrarily-long input string to a fixed-length output string;
2. it is an almost pairwise-independent hash function;
3. it is a one-way function with second-preimage resistance;
4. in addition to elementary operations, it only uses, as a black-box, a hash function satisfying above properties 2 and 3, and achieving small and fixed compression (specifically: it maps a fixed-length input string to a fixed-length output string, where the difference between the input string's length and the output string's length can be any small constant ≥ 1).

Properties 1 and 4 (resp., 2 and 3) are used to satisfy functionality correctness and efficiency (resp., security) requirements. The closest constructions to ours from the literature only satisfy 3 out of 4 of the listed properties, as follows: two constructions in [17] missed properties 1 or 4, and a construction from [4,18] missed property 2.

Formally, we achieve the following

Theorem 1. *Let $t_{F,sample}$ (resp., $t_{F,eval}$) denotes the running time to sample (resp., evaluate) a function F. Let $\{aF \,|\, aF : \{0,1\}^b \rightarrow \{0,1\}^b\}$ be a family of one-way α-permutations, and let $\{piH \,|\, piH : \{0,1\}^a \rightarrow \{0,1\}^b\}$ be a family of pairwise δ-independent hash functions. There exists (constructively) a family $\{sprH \,|\, sprH : \{0,1\}^{\ell(a-b)} \rightarrow \{0,1\}^b\}$ of second-preimage-resistant (k, ϵ)-extractors such that*

- *$b \leq k - 2\log(1+\epsilon)$ and $\epsilon \leq (1/2)\log(1/\delta')$, for $\delta' = \ell(\delta + \alpha)$*
- *$t_{sprH,sample} = O(\ell(t_{piH,sample}) + t_{aF,sample})$, and*
- *$t_{sprH,eval} = O(\ell(t_{aF,eval} + t_{piH,eval} + t_{piH,sample}) + t_{aF,sample})$.*

The function *sprH* obtained in the proof of Theorem 1 will be applied to obtain the following two important new results: (1) in Sect. 4, it will be used in combination with the obfuscators from [5–7,13], and [19], to design efficient obfuscators for point functions with secret length higher than the factoring-type security

parameter (e.g., 2048); (2) in Sect. 5 it will be used to design an efficient obfuscator for multi-bit-output point functions with secret length greater than or equal to the symmetric-cryptography security parameter (e.g., 128). The rest of this section is devoted to proving Theorem 1.

Informal Description of Function $sprH$: Our goal is to define a family of functions, denoted as $sprH$, that satisfies the above properties 1–4. One higher-level view of our construction looks similar to the linear hash construction in [4,18], and its lower-level component looks similar to a function from [17]. However, some technical differences with these papers actually allow us to achieve all 4 desired properties; most importantly:

1. $sprH$ processes an arbitrarily long input by repeatedly applying an inner function with the same domain and codomain sizes (instead, in [17] domain and codomain sizes vary). This approach is important to achieve properties 1 and 4.
2. in $sprH$ the inner function used at each iteration is both a second-preimage-resistant function and a pairwise almost-independent hash function (as opposed to only a collision-intractable hash function, as in [4,18]). This approach is important to achieve properties 2 and 3.

Formal Description: Let $desc(F)$ denote a conventional encoding of function F, and let a, b denote positive integers such that $a > b$ and $a - b \geq 1$ is a small constant. The construction for $sprH$ uses a pairwise δ-independent hash functions $piH : \{0,1\}^a \rightarrow \{0,1\}^b$, and a one-way α-permutations $aP : \{0,1\}^b \rightarrow \{0,1\}^b$. We define function $sprH : \{0,1\}^* \rightarrow \{0,1\}^b$, as follows.
Input to $sprH$: string $x = x_1 | \cdots | x_\ell$, where $x_i \in \{0,1\}^{a-b}$, for $i = 1, \ldots, \ell$.
Instructions for $sprH$:

1. Set $u_0 = 0^b$
2. Randomly sample a one-way α-permutation aP
3. For $i = 1, \ldots, \ell$,
   randomly sample a pairwise δ-independent hash function piH_i
   compute $v_i = aP(u_{i-1}|x_i)$ and $u_i = piH_i(v_i)$
4. Return: $(u_\ell$, desc(aP), desc$(piH_1), \ldots,$, desc$(piH_\ell))$.

The running times $t_{sprH,sample}$ and $t_{sprH,eval}$ claimed in Theorem 1 are verified by algorithm inspection, observing that $sprH$ can compress arbitrarily long inputs into b-bit outputs, and that it invokes ℓ times a single function $piH(aP(\cdot))$ compressing a-bit outputs to b-bit outputs. In what follows, we show that $sprH$ is a second-preimage-resistant (k, ϵ, δ')-extractor with the parameters in Theorem 1, by showing that it is both a second-preimage-resistant hash function over $\{0,1\}^a$ and a pairwise δ'-independent hash function.

The proof that $sprH$ is a second-preimage-resistant hash function directly follows by applying results in [4,17,18], as follows. First, we observe that the function obtained by cascading a one-way α-permutation aP with a pairwise δ-independent hash function piH, is a second-preimage-resistant hash function.

This follows directly by Lemma 2.2 in [17], which proves the exact same result when $\alpha = 0, \delta = 0$ and $a - b = 1$. We observe that no technical difficulty is encountered in extending this proof to values of α, δ that are negligible or very small and a value of $a - b$ that is a small constant (or even logarithmic in the security parameter). Because of this observation, we note that $sprH$ can be considered as the linear hash iterated application of a second-preimage-resistant hash function, as in the linear hash construction from [4,18]. In particular, we can apply Theorem 5.3 from [4] which proves our desired statement; i.e., the linear hash construction transforms a second-preimage-resistant hash function from a-bit strings to b-bit strings into a second-preimage-resistant hash function from arbitrary-length strings to b-bit strings.

The proof that $sprH$ is a pairwise δ'-independent hash function is obtained by induction over ℓ. The base case directly follows by observing that the assumptions that function piH_1 is pairwise δ-independent and that function aP is an α-permutation imply that the composed function $piH_1(aP(\cdot))$ is a pairwise δ'-independent hash function, for $\delta' = \alpha + \delta$. The inductive case follows by combining the induction hypothesis with the fact that at the ℓ-th iteration, function $sprH$ computes u_ℓ using function $piH_\ell(aP(\cdot))$ for an independently chosen pairwise δ-independent hash function piH_ℓ.

Implementation: Primitive Setting. Families of pairwise-independent hash functions piH_i can be implemented as in Sect. 2. Function aP can be instantiated in 3 ways:

1. setting $n = 2048$, and using exponentiation modulo a prime; that is, $aP_{g,p}(x) = g^x \bmod p$, where publicly available parameters p, g are as follows: p is an $(n+1)$-bit prime and g is a generator of $\mathbb{Z}_p^*$;
2. using a length-preserving collision-intractable hash function $cih_k : \{0,1\}^n \to \{0,1\}^n$ for which no theoretical attacks (faster than birthday attacks) are known, and assuming such a function is a one-way α-permutation, for a value α negligible in n or very small; that is, $aP_{cih,k}(x) = cih_k(x)$;
3. as a block cipher $bc : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ for which no theoretical attacks (faster than exhaustive search attacks) are known, to be run on a fixed, but randomly chosen, input block r, and assuming that the resulting function $bc(\cdot, r)$ is a one-way α-permutation over the set of block cipher keys, for α negligible in n or very small; that is, $aP_{bc,r}(x) = bc(x, r)$.

In our implementation, we used the 3rd option for efficiency reasons, and based on the observation that function $aP_{bc,r}(x)$, mapping the set of keys of the block cipher to the cipher's output, is indeed expected to be a one-way α-permutation. This observation is based on the fact that if function $aP_{bc,r}(x)$ were not close to a one-way α-permutation, a theoretical attack exhaustively searching for any of the colliding keys would be possible. Note that such an attack would be faster than exhaustive key-search, thus giving a theoretical break of the block cipher.

4 Obfuscators for Point Functions with Larger Secrets

In this section we show how to obtain point function obfuscators where the obfuscated secret value can have length and min entropy parameters arbitrarily greater than the factoring-type security parameter, starting from a point function obfuscator where the obfuscated secret value has fixed length and min-entropy parameter, which we already know how to build. Formally, we obtain the following

Theorem 2. *Let $\ell_a, e_a, \ell_u, \epsilon$ be integers such that $\ell_u + 2\epsilon \leq e_a \leq \ell_a$ and $\epsilon \geq \lambda_s$, let sprH be a second-preimage-resistant (ℓ_a, ϵ)-extractor, and let ($genO_u$,$evalO_u$) be a cryptographic program obfuscator for the family of point functions with ϵ-uniformly distributed ℓ_u-bit secret values. Then there exists (constructively) a cryptographic program obfuscator obfuscator ($genO_a$,$evalO_a$) for the family of point functions with respect to ℓ_a-bit secrets drawn from any distribution of min-entropy e_a.*

An important consequence of Theorem 2 is that any one of the point function obfuscators in [7], [5,6,13], or [19] can be extended to obtain a point function obfuscator that works for secret values with arbitrarily larger length and drawn from arbitrary distributions of min entropy larger than the factoring-type security parameter.

Informal and Formal Descriptions: The basic idea of the transformation underlying Theorem 2 follows a 'hash-and-obfuscate' paradigm, analogously to the much studied 'hash-and-sign' paradigm used for the design of digital signature schemes for large messages. This paradigm goes through two steps: first, the input is hashed using a second-preimage-resistant extractor, which we will implement using the construction *sprH* from Sect. 3; then, the extractor's output is processed through the obfuscator with fixed length parameter, which can be instantiated using any one of the schemes from the literature (e.g., [5,5–7,13,20], [10] or [19].) The resulting scheme satisfies computational functionality correctness, and the same adversary view simulation obfuscation notion as the used obfuscator for fixed-length secrets. We now proceed more formally. The construction for (genO$_a$,evalO$_a$) uses the family of efficiently samplable functions $sprH : \{0,1\}^* \rightarrow \{0,1\}^b$ from Sect. 3, which are simultaneously second-preimage-resistant hash functions and pairwise δ'-independent extractors, and an obfuscator (genO$_u$,evalO$_u$) for the family of point functions with length parameter ℓ_u and secret values with almost uniform distribution.
Input to $genO_a$: parameters $1^{e_u}, 1^{\ell_u}, \epsilon$, secret value $s \in \{0,1\}^{\ell_a}$
Instructions for $genO_a$:

1. Randomly sample function $sprH : \{0,1\}^{\ell_a} \rightarrow \{0,1\}^{\ell_u}$
2. Compute $v = sprH(s)$
3. Compute $out_u = genO_u(v)$
4. Return: $out_a = (desc(sprH), out_u)$.

Input to $evalO_a$: input value $x \in \{0,1\}^{\ell_a}$ and the output from $genO_a$, containing the description $desc(sprH)$ of function $sprH$ and the output out_u from $genO_u$.

Instructions for $evalO_a$:

1. Compute $v' = sprH(x)$
2. Return: $evalO_u(v')$.

Proofs are omitted due to space restrictions.

5 Obfuscators for Multi-bit-output Point Functions With Shorter Secrets

In this section we describe an obfuscator, denoted as $(genO_{mb}, evalO_{mb})$, for the family of multi-bit-output point functions, where secrets can have a shorter length parameter than in our previous implementations, which implies applicability to the obfuscation of passphrases and even passwords. More specifically, this obfuscator differs from analogue results in the literature and in previous sections, in the following properties:

1. it works for a generalized type of point functions: multi-bit-output point functions, whose output can be a long string, instead of a bit;
2. it works for a length parameter that can be arbitrarily chosen as $\geq$ the symmetric-cryptography security parameter (i.e., 128), instead of the factoring-type security parameter (i.e., 2048);
3. its obfuscation property can be based on the security of a symmetric cryptography primitive (i.e., a block cipher or a cryptographic hash function), instead of a number theory problem typically applied to construct an asymmetric cryptography primitive.

Formally, we achieve the following

Theorem 3. *Let* $\ell_o, \ell_s, k, \epsilon$ *be integers such that* $k \leq \ell_s$. *Also, let* $sprH$ *be a second-preimage-resistant* (k, ϵ)*-extractor and let (KeyGen, Enc, Dec) be a secure symmetric encryption scheme. Then there exists (constructively) a cryptographic program obfuscator* $(genO_{mb}, evalO_{mb})$ *for the family of multi-bit output point functions* $mbpF$ *with* ℓ_o*-bit outputs and* ℓ_s*-bit secrets drawn from any distribution of min-entropy* k.

We note that in the above theorem we are trading off some slightly, but not significantly, reduced confidence in the security assumptions (as indicated in item 3 of the above list), to achieve increased functionality power (as indicated in items 1 and especially item 2 of the above list). Indeed the property in item 1 can be obtained without resorting to symmetric cryptography primitives (see, e.g., [8]), but this comes with decreased obfuscator's efficiency. The (most interesting) property in item 2 was unknown and is the one that allows applications to passphrase and password obfuscation, as further detailed in Appendix A.

Formal Description: Let | denote string concatenation, and let $sprH$ denote a second-preimage-resistant (k, ϵ)-extractor (such as the one constructed in Sect. 3). Also, let (KeyGen, Enc, Dec) be a symmetric encryption scheme with the following syntax: on input a unary string 1^n denoting the symmetric encryption security parameter, KeyGen returns an n-bit random key; on input key and message m, encryption algorithm Enc returns ciphertext c; on input key and ciphertext c, decryption algorithm Dec returns message m. Our construction of $(genO_{mb}, evalO_{mb})$ combines the extractor $sprH$ with the encryption scheme (KeyGen, Enc, Dec), as follows.
Input to $genO_{mb}$: security parameters $1^n, 1^{n_0}, 1^\epsilon$, entropy parameter k, secret value $s \in \{0,1\}^{\ell_s}$, output value $w \in \{0,1\}^{\ell_o}$
Instructions for $genO_{mb}$:

1. uniformly and independently choose $r \in \{0,1\}^{n_0}$
2. compute $key = sprH(r|s)$, where $key \in \{0,1\}^n$
3. compute $v = \mathrm{Enc}(key, w|0^{n_0})$
4. set $gpar = (r, v)$ and return: $gpar$.

Input to $evalO_{mb}$: security parameters $1^n, 1^{n_0}, 1^\epsilon$, entropy parameter k, the pair (r, v) returned by $genO_{mb}$, and input value $x \in \{0,1\}^\ell$
Instructions for $evalO_{mb}$:

1. compute $key' = sprH(r|x)$, where $key' \in \{0,1\}^n$
2. compute $(w'|w'') = \mathrm{Dec}(key', v)$
3. if $w'' = 0^{n_0}$ return w' else return 0

Proofs and performance analysis are omitted due to space restrictions.

6 Conclusions

We showed for the first time how to *efficiently* obfuscate passwords, passphrases and password managers, *without a random oracle assumption*. Our obfuscator can work with passwords and passphrases of practical lengths. Even if we expect practitioners to continue using the simpler to implement construction based on cryptographic hashing of a password, our construction gives confidence that the impact of any future attacks to cryptographic hash functions can be significantly limited by a simple protocol design change.

A Applications: How to Obfuscate Password Verification, Passphrase Verification and Password Managers

In this section we show how to use the obfuscators from previous sections to obfuscate, using standard cryptographic assumptions (and specifically not using the random oracle assumption), software applications commonly used in real-life, such as password verification, passphrase verification, and password managers.

Obfuscation of Password Verification. Entity authentication based on shared secrets is one important class of system applications that seem to significantly rely on obfuscation, as we now explain. Consider the typical scenario of a client and a server who use a secret to let the server successfully register and later authenticate a client, as follows:

1. registration: the client gives an obfuscated version of the secret verification program to the server, thus not revealing the secret to any server intruder;
2. authentication: the client securely sends the secret to the server, which runs the obfuscated version of the secret verification program to verify that the received secret is the same as the one in the obfuscated verification program.

One important case is when such secret is the client's password. Indeed, *entity authentication via password verification*, has often been used as an application motivating the design of program obfuscators for point functions. Note that in this case, the verification program computes precisely a point function, where the secret value is the client's password stored during the registration phase, and the obfuscated program's input is the client's string entered during the authentication phase. As mentioned before, point function obfuscators in the literature are either proved secure under the random oracle assumption or for secret points of length about equal to the factoring-type security parameter (i.e., 2048 bits, which is much more than the length of real-life passwords). When using passwords with ASCII characters including lowercase letters, uppercase letters, numbers and special symbols (for a total of 96 characters), a password of 20 uniformly and independently chosen characters will contain just below 132 bits of entropy. Note that passwords of 20 or more characters have already widespread usage, for instance, in WiFi access to residential networks in private homes. Our solution, which works only when passwords are chosen by the user, can be defined as the hashing-based scheme from [16], with the only difference that the cryptographic hash function H, which could be implemented as SHA2 or SHA3, is actually replaced by the hash function $sprH$ from Sect. 3, where the one-way α-permutation is instantiated using a block cipher like AES, as already described there. With this instantiation, the provable properties of hash function $sprH$ only depend on the (arguably reasonable) assumption that a block cipher like AES (when parameterized by a random input block x and seen as a function $F(\cdot, x)$ mapping the key k to an output $y = F(k, x)$) is a one-way α-permutation, for some small α. Following the discussion at the end of Sect. 3, this assumption is also supported by the lack of a known theoretical attack faster than exhaustive key-search for AES.

Obfuscation of Passphrase Verification. Our solution for password verification is directly applicable to passphrase verification, where a meaningful English sentence, with lower entropy per character, is used as a passphrase. Various techniques have been proposed in the literature to estimate the average number of entropy bits per character in an English passphrase (typically, a number between 0.5 and 3), and may vary depending on the specific assumptions made on the used character sequences. After estimating the average number v of entropy

bits per character in a passphrase taken from a desired set of sequences, a system designer could augment passphrase choice requirements by requiring that a passphrase has length at least q, satisfying $qv \geq 128$.

Obfuscation of a Password Manager. Our solution for password verification is also directly applicable to password managers, another pervasive real-life authentication application, as today the number of password-based services used by the average computer user has increased dramatically. We have evaluated some password manager packages for suitability for our experiments with point function obfuscators, and chosen Pass, a well-known, open-source, password manager [1]. The cryptography currently used in Pass can be seen as a natural extension of the obfuscator from [16] for point functions, to an obfuscator for the password manager's password derivation program, under the assumption that the used collision-resistant hash function behaves like a random oracle. We have augmented this password manager with an option that obfuscates the password manager's password derivation program without making a random oracle assumption. Our option processes the user's passphrase or password using our multi-bit-output point function obfuscator from Sect. 5, which is based on the hash function from Sect. 3, and only makes the previously discussed assumption on a block cipher with unknown key. Recall that our multi-bit-output obfuscator from Sect. 5 could work in two modes: a symmetric or an asymmetric mode, depending on whether the multi-bit output string was encrypted using symmetric or asymmetric encryption. Because Pass already encrypts the website passwords using asymmetric encryption, we could use our multi-bit-output obfuscator in asymmetric mode, and were able to significantly reduce code complexity by focusing our software production on augmenting Pass with the use of our hash function from Sect. 3.

Performance Analysis. We used (here and in the rest of the paper) a Dell 2950 processor (Intel(R) Xeon(R) 8 cores: CPU E5405 @ 2.00GHz, 16 GB RAM), without parallelism. We performed two types of performance analysis. In Table 1 we compare the running time of the cryptographic hash function SHA1 implemented under Pass against the running time of our proposed replacement: i.e.,

Table 1. Performance of SHA1 against our hash function $sprH$.

Input length	Output length	SHA1 running time	$sprH$ running time
64	128	0.0000 s	0.0001 s
128	128	0.0000 s	0.0001 s
192	128	0.0001 s	0.0004 s
256	128	0.0001 s	0.0007 s
320	128	0.0001 s	0.0010 s
384	128	0.0001 s	0.0014 s
448	128	0.0001 s	0.0017 s
512	128	0.0001 s	0.0020 s

our cryptographic hash function $sprH$. In Table 2 we compare the running times of Pass procedures against the running time of our newly proposed option: i.e., Pass with our multi-bit-output point function obfuscator.

Our second set of performance results is about a metric, denoted as *time ratio*, and defined as the ratio of the running time of Pass while using SHA1 to the running time of Pass while using our hash function. Because Pass is essentially performing not much computation other than running calls to gpg2, we performed our measurements directly on gpg2 calls.

Table 2. Performance of Pass with SHA1 against Pass with our hash function $sprH$

Type of master secret	Estimated entropy	Time ratio on gpg2 -X –gen-key	Time ratio on pass insert <site>	Time ratio on pass -X show <site>
Password	132	0.8756	1	0.9783
Passphrase	100	0.9606	1	0.9363

In Table 2, we consider the three main commands that can be run with Pass: key generation, website password insertion and website password recovery. In the first row, we used a master password of 20 characters uniformly and independently chosen from a set of 96 ASCII characters. In the second row, we used a master passphrase of 56 characters from a meaningful English sentence, which was roughly estimated to have about 100 bits of entropy. As clear from the last 3 columns of the table, our modified version of Pass only slows down Pass by very small percentages, while offering a password manager obfuscation without a random oracle assumption.

References

1. https://www.passwordstore.org/
2. Bahler, L., Di Crescenzo, G., Polyakov, Y., Rohloff, K., Cousins, D.B.: Practical implementations of lattice-based program obfuscators for point functions. In: International Conference on High Performance Computing & Simulation, HPCS (2017)
3. Barak, B., et al.: On the (im)possibility of obfuscating programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 1–18. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_1
4. Bellare, M., Rogaway, P.: Collision-resistant hashing: towards making UOWHFs practical. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 470–484. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052256
5. Bellare, M., Stepanovs, I.: Point-function obfuscation: a framework and generic constructions. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 565–594. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_21

6. Boldyreva, A., Fehr, S., O'Neill, A.: On notions of security for deterministic encryption, and efficient constructions without random oracles. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 335–359. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_19
7. Canetti, R.: Towards realizing random oracles: hash functions that hide all partial information. Proc. CRYPTO 97, 455–469 (1997)
8. Canetti, R., Dakdouk, R.R.: Obfuscating point functions with multibit output. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 489–508. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_28
9. Cannetti, R., Goldreich, O., Halevi, S.: The random oracle methodology, revisited (preliminary version). In: Proceedings of 30th ACM STOC, pp. 209–218 (1998)
10. Canetti, R., Micciancio, D., Reingold, O.: Perfectly one-way probabilistic hash functions (preliminary version). In: Proceedings of 30th ACM STOC, pp. 131–140 (1998)
11. Carter, L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. **18**(2), 143–154 (1979)
12. Di Crescenzo, G., Bahler, L., Coan, B., Polyakov, Y., Rohloff, K., Cousins, D.B.: Practical implementations of program obfuscators for point functions. In: International Conference on High Performance Computing & Simulation, HPCS (2016)
13. Freeman, D.M., Goldreich, O., Kiltz, E., Rosen, A., Segev, G.: More constructions of lossy and correlation-secure trapdoor functions. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 279–295. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_17
14. Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. SIAM J. Comput. **28**(4), 1364–1396 (1999)
15. Impagliazzo, R., Zuckerman, D.: How to recycle random bits. In: 30th IEEE FOCS 1989, pp. 248–253 (1989)
16. Lynn, B., Prabhakaran, M., Sahai, A.: Positive results and techniques for obfuscation. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 20–39. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_2
17. Naor, M., Yung, M.: Universal one-way hash functions and their cryptographic applications. In: Proceedings of 21st ACM STOC 1989, pp. 33–43 (1989)
18. Shoup, V.: A composition theorem for universal one-way hash functions. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 445–452. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_32
19. Wee, H.: On obfuscating point functions. In: Proceedings of 37th ACM STOC 2005, pp. 523–532 (2005)
20. Xie, X., Xue, R., Zhang, R.: Deterministic public key encryption and identity-based encryption from lattices in the auxiliary-input setting. In: Visconti, I., De Prisco, R. (eds.) SCN 2012. LNCS, vol. 7485, pp. 1–18. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32928-9_1