On Resilient Password-Based Key Derivation Functions

Jason Hennessey
Sarah Scheffler
Mayank Varia

ABSTRACT

Password-based key derivation functions (PBKDFs) create keys that are used to protect a great wealth of private information. In the modern world where we bring our devices everywhere, there are ample opportunities for criminals or governments to use hardware dedicated to the task of brute-force "cracking" PBKDFs in order to enter these devices.

Existing PBKDFs require substantial computing power or memory with the goal of running slowly in order to raise the cost of brute-forcing. However, with the increased availability of specialized hardware, attackers can often brute-force crack passwords in a matter of seconds to hours. This method of slowing the computation is no longer viable when the attacker has hardware that can compute the PBKDF orders of magnitude faster than the defender.

Even though they are designed by both the systems and cryptography communities, the actual requirements of a PBKDF remain muddled somewhere in between. In this paper, we provide a definition of *resilient PBKDFs*. This definition combines (1) the cryptographic requirement that low-entropy passwords can produce keys that can only be broken via brute force search with (2) the systems requirement the components collectively permit an attacker to perform at best a linear speedup over the defender's execution, no matter the attacker platform.

Additionally, we construct a resilient PBKDF called Bog that meets our definition. Bog achieves resilience by iterating the hash combiner of Fischlin, Lehmann, and Pietrzak with a pluggable architecture for plugins that consume resources of different types. Bog ties these primitives together via a sponge function design introduced by Bertoni et al. Bog was developed while keeping both cryptographic techniques and systems security principles in mind. We provide a proof of security of Bog in the Random Oracle Model, as well as a proof-of-concept implementation.

1 INTRODUCTION

Laptops and smartphones today hold an extraordinary amount of sensitive, personal details about all aspects of people's lives. Full disk encryption (FDE) is our best defensive mechanism today to protect the confidentiality of this data when devices are stolen by thieves or temporarily confiscated by governments, especially when crossing the border between countries. Since FDE must withstand an attacker who possesses and can introspect the device, it follows that the decryption key cannot be stored on the device itself. Instead, the (perhaps multi-factor) authentication of a user must be cryptographically bound to the user's authorization to decrypt the data on the device.

Password-based key derivation functions (PBKDFs) provide this binding. PBKDFs use input factors like passwords, physical tokens, and biometrics (along with public data like a salt and the key length) to generate the output key required to decrypt the disk. Because passwords are notoriously low-entropy, password-based systems are particularly vulnerable to brute-force attack.

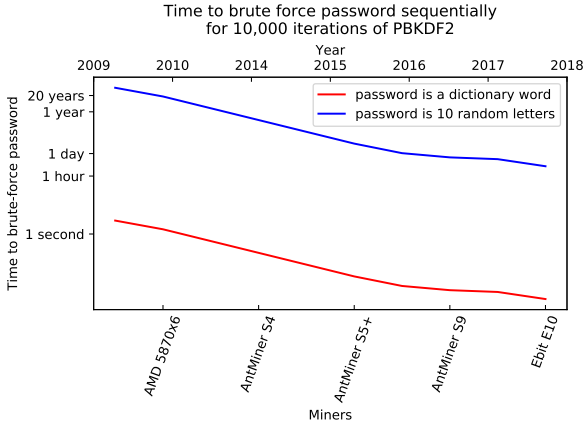

Figure 1: Time to brute-force PBKDF2 with 10,000 iterations for a password space of a single dictionary word (red) or 10 random letters (blue). Estimates for cracking hash rates were approximated with the Bitcoin hashing rate of miners on CPUs, GPUs, and commercially available ASICs [1, 19, 20]. Due to Bitcoin's use of SHA2-256, it is expected that a PBKDF2 iteration is roughly equivalent to a single hash.

PBKDFs today. Modern PBKDFs expend computing resources to produce the key *slowly*, in an attempt to thwart brute-force attacks [64]. A slowdown is beneficial because the honest user only pays this cost once whereas a brute-force attacker must pay this cost for every password attempt. The NIST-standardized algorithm PBKDF2 [64] relies on repeated iterations of a single function to achieve the desired slow computation speed.

But this projected cost to the adversary relies on an unstated assumption: that computation of the PBKDF costs the same for the adversary and the honest user. In the case of a border crossing, a full copy of the information on disk could be made within minutes to hours, and then the PBKDF can be computed on a different device that is specialized for computation of the PBKDF. As shown in Figure 1, brute-forcing a PBKDF on dedicated hardware provides many orders of magnitude of faster computation, [2, 31, 44] to the point where most FDE systems can be broken in a few hours by an adversary using dedicated hardware, even for iteration counts that make the login time prohibitively high for the honest user.

Newer PBKDFs utilize two techniques to thwart this computation asymmetry. First, newer cryptographic primitives like scrypt [50], bcrypt [52], and argon2 [18] are algorithmically designed to re-balance attacker and defender effort by leveraging features

available on defender machines (e.g., large RAM) and reducing the benefit of features that are unlikely to be on defender machines (e.g., parallelism across several cores). Second, modern disk encryption systems connect these crypto primitives with trusted hardware or operating systems-level protections.

The net result of these countermeasures is the design of a complicated full-stack PBKDF system that no single person can analyze, that lacks a clear definitional goal to achieve, and that tends to break down whenever someone penetrates its weakest link.

Need for a concrete, full-stack definition. Apple iPhones have some of the best-engineered PBKDFs to date, combining several standardized cryptographic primitives with a trusted hardware element and an operating system-enforced erasure failsafe [8]. Even with these 3 interlocking pieces, a company called Cellebrite built a system called 'GrayKey' that facilitates brute-force dictionary attacks [53]. In response, Apple's latest operating system update introduced new OS and hardware-level countermeasures that are specifically designed to thwart GrayKey [5].

This type of cat-and-mouse game is precisely what cryptography has typically avoided, thanks to rigorous security definitions that are independent of any construction and that guarantee resilience against a powerful attacker. Ergo, the Apple-Cellebrite story is emblematic of a larger issue: to date we have been treating PBKDFs as hash function constructions that happen to have additional properties. PBKDFs are different from their two parents (password-based hashes and KDFs), and we should consider them as a first-order objective worth achieving on their own, with a security definition that combines their crypto and systems security requirements.

Need for resilience. Rather than consuming one resource (computing power, or in the case of scrypt [50], memory and computing power), a PBKDF should consume many different kinds of resources. This mitigates attackers' specialized hardware advantage and provides an "approximate" localization that ensures that the advantage of attackers' specialized hardware over the honest user's device is at most linear in the number of different resources consumed. This can be thought of as paying a linear cost in return for an exponential gain in the cost ratio of the defender to the attacker.

It should also go a step further and include *resilience* - 10,000 iterations of a function was much more burdensome a decade ago before ASICs became so much cheaper due to Bitcion. Ideally, a PBKDF would have resiliency and graceful degradation, so that as each component of the function became obsolete, the system as a whole would continue functioning well.

1.1 Our Contributions

This work provides a joint crypto-systems definition of a resilient PBKDF that combines concepts from cryptography and systems security as well as a construction, called Bog, that provably achieves this definition. More concretely, we claim three contributions in this work.

Defining resilient PBKDF. First, we provide a definition and set of security guarantees for resilient password-based key derivation functions. Our definition operates in the random oracle model, and it captures a list of systems requirements that we also include.

Formalizing resources to assess defender optimality. Second, we include an abstract conceptualization of *resources* consumed when attempting to brute-force a password. These extend and formalize the FDE KDF "resource-consumption" idea of Brož et. al. [22].

The generic notion of resources helps us to capture both the guarantee that components should attempt to be as cost-optimal and localized to the defender's system as possible and also that the resilient system should achieve (approximately) the best security margins achieved by any of its underlying primitives. We demonstrate that for a well-balanced set of defender resources, adversaries can be restricted to a linear advantage over an honest defender's resources, even accounting for parallelism. This restricts the benefit of using specialized hardware that would be orders of magnitude faster than the honest user's device for PBKDF2.

Bog construction. Finally, we introduce Bog, a construction and prototype Rust implementation of a resilient PBKDF. Bog uses Bertoni et. al.'s sponge function [16] to combine several plugins that each take advantage of different resources available on the device. The resource-consuming plugins together ensure that computation is optimized and localized to the defender's machine. Additionally, Bog uses Fischlin et. al.'s hash combiner [38] to provide a password with high (pseudo)entropy even if all-but-one hash functions are later discovered to be broken, backdoored, or malicious. We prove that Bog meets our security definition in the random oracle model.

Bog has graceful degradation. The compromise of a resource-consuming plugin is not catastrophic; the resources consumed by the remaining non-bypassed plugins are unaffected. The hash combiner ensures that we have the guarantees of our strongest hash function, even if we're not sure which hash function is the strongest.

When designing Bog, we approached the problem from both a cryptography and a systems security standpoint, hoping to get the best of both worlds.

1.2 Outline

In Section 2, we provide requirements and provide a definition for a resilient PBKDF. In Section 3, we describe Bog, our construction of a resilient PBKDF. In Section 4, we demonstrate that Bog meets the definition of a resilient PBKDF. Section 5 provides a description of our proof of concept implementation of Bog. Finally, Section 6 describes several categories of related work.

2 DEFINING A RESILIENT PBKDF

Password-based key derivation functions are used in a variety of applications (cf. §6), yet there is little analysis of their desired properties from both the systems and the cryptography communities. The *KDF* nature of the PBKDF specifies stringent output randomness requirements that surpass those of normal password hashing,[1] and the *password-based* part of the PBKDF means that some crypto definitions are not necessarily useful. (In particular, a PBKDF is not a pseudorandom function because it lacks a high-entropy secret key.)

[1] As a simple counterexample: one could append a 0 bit to the end of all password hashes and no security in the password hash was lost. But appending a 0 to the end of the hash and then treating it as a key does impact the security of things that use that key in the future.

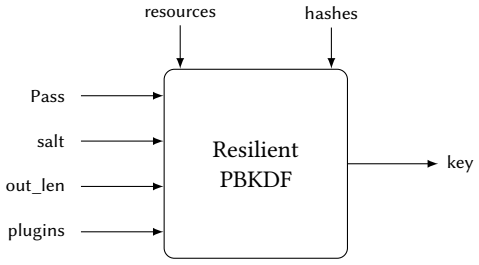

Figure 2: The input-output behavior of resilient PBKDFs like Bog and RObog. Note that the user doesn't select the resources; they are an intrinsic part of the computing environment. Also, the device's legitimate owner chooses the hash functions, which then remain constant across invocations.

According to the NIST specification of PBKDFs [64], the main purpose of the design of PBKDFs is to slow down brute-force dictionary attacks on passwords. The PBKDF2 specification uses a set number of iterations of a pseudorandom function (PRF) in order to achieve this slowness. The PBKDF takes as input a private but low-entropy password, a public high-entropy salt, and a key output length. Since it is a KDF, the output must be pseudorandom and suitable for use as a cryptographic key. This holds the output to a higher standard than a typical password-hashing function, which only requires the function to be slow and collision resistant.

In this section, we define requirements and security guarantees for a *resilient PBKDF*. It incorporates all the requirements of a PBKDF stated above (random output, resource-consuming). Novelly, our construction also addresses differences in resources between attackers and defenders; even an attacker with specialized hardware must face a high cost for brute-forcing. We also believe that a PBKDF should demonstrate *resilience*; that is, it should maintain its full security unless all of its components are broken.

We first list our systems-level requirements, then provide a definition for resilient PBKDF that complements these requirements.

2.1 Requirements

A resilient PBKDF is one where the risk of total compromise is spread out among several different components such that a failure or degradation in any of these components results in a proportional graceful degradation of the overall construct. Ergo, in addition to requiring that the system transform a low entropy password into a high (pseudo)entropy key, a resilient PBKDF must achieve the following additional properties.

(1) *Defender Optimality:* A PBKDF must resist brute-force attacks. So the defender's device must be an optimal place to compute the function - or at least, the defender's device cannot be significantly worse than the attacker's device at computing the function. In the ideal case, the defender's device is optimal for computing the function. Informally, the best way for an attacker to brute-force the PBKDF is to possess many copies of the defender's device.

One way to achieve this is to combine multiple plugins, each of which takes advantage of one defender resource, such as memory, L2 cache, or CPU instruction agility. At that point, any attacker who has the same combination of resources either has the defender's device itself, or has something that is functionally equivalent.

(2) *Resiliency*: Above all, a resilient PBKDF should continue providing security guarantees even if some of its component parts are found to be broken, backdoored, or just not good enough. Security should remain intact unless *all* components are broken. Similarly, failure in consuming one resource should not impact other resources consumed.

(3) *Locality*: Component selection can also *localize* the PBKDF function by using, e.g., a Hardware Security Module [62] will ensure that *only* the honest device can consume this resource. Though it should be difficult, with enough work, the attacker can acquire these localizing resources (say, by stealing the user's laptop), and so our security game in the upcoming section understands the difference between *local* and *nonlocal* plugins. To compute the resilient PBKDF, an adversary must first Acquire all local resources.

(4) *Crypto agility*: Hashing algorithms have been broken in the past [43, 56, 60, 61] and it is reasonable to assume that others will be broken in the future. Additionally, new designs are likely to be created to take their place. A resilient PBKDF improves cryptographic agility for two reasons: first, its graceful degradation provides time to swap out a broken algorithm as long as at least one hash function remains unbroken, and second, its pluggable architecture provides a simple upgrade path to new constructions.

(5) *Statelessness*: Algorithms should not need to preserve state between different iterations of the PBKDF. They should not even need to preserve state once they are done with their one component of the function, nor should they behave differently when given different input. This includes being timing-independent of the password itself, as identified by [22].

(6) *Isolation*: Implementations of algorithms may have exploitable flaws or be actively malicious. The construct should take steps to ensure that their failures can't affect the other components in the system - even if the function itself is "malicious" and will try to undo work done by other parts. For complex functions, this may involve containerization or isolation. For simpler ones, especially hash functions, it may mean simply reading all 100 lines of the hash function code to ensure that it is not doing any external reads or writes.

In the next section, we provide a cryptographic definition that incorporates these properties.

2.2 Definition And Security Guarantees

Definition 2.1 (resilient PBKDF). A function of the form

$$\mathcal{F}^{h_1,\ldots,h_m} : (\mathsf{Pass}, \mathsf{salt}, \mathsf{out\_len}, \mathsf{plugins}) \mapsto \mathsf{key}$$

is a *resilient PBKDF* if the following 2 game-based properties hold.

Indifferentiable: It is (q, t, ϵ)-indifferentiable from a keyed random oracle $\mathcal{R}$ with the same input and output lengths (cf. Definition 4.1). Put simply, this property states the best

that $\mathcal{A}$ can do to learn passwords is to perform a brute-force attack.

Resource-bounded: For all adversaries $\mathcal{A}$, there exists a negligible function negl such that for all security parameters $\lambda \in \mathbb{N}$, Pr[$\mathcal{A}$ wins the resilient PBKDF game] $<$ negl(λ), where the probability is taken over the randomness of $\mathcal{C}$ and $\mathcal{A}$. Hence, brute-force attacks are resource-intensive.

The resilient PBKDF resource game defined below gives the adversary to *bypass* plugins. This represents some fundamental way through which a function is broken, such as if the attacker has a backdoor. It does *not* encode different resources between the attacker and defender

We devote the rest of this section to a thorough description of the games that underlie Definition 2.1. Both the games begin with the same setup phase in which the challenger constructs, and then the adversary partially breaks, the resilient PBKDF construction. This setup process has three phases: choosing plugins (S1), choosing hash functions (S2), and permitting the adversary to bypass plugins (S3).

(S1) **Choose plugins.**
  (a) The challenger $\mathcal{C}$ picks plugins $\mathcal{P}_i$, which are pseudorandom function families. It picks ℓ of them from $\mathcal{L}$ and $(m - \ell)$ of them from plugins.
  (b) From each family $\mathcal{P}_i$, $\mathcal{C}$ picks a function p_i.
  (c) $\mathcal{C}$ gives all families $\mathcal{P}_i$ to $\mathcal{A}$.
  (d) For all nonlocal plugins ($\mathcal{P}_i \notin \mathcal{L}$), it gives $\mathcal{A}$ oracle access to the functions p_i.

(S2) **Choose hash functions.**
  (a) The adversary $\mathcal{A}$ picks how many hash functions n will be run, and chooses $c < n$ of them. It passes the code of these hash functions to $\mathcal{C}$. (The hash functions must be stateless between calls and deterministic.)
  (b) The challenger $\mathcal{C}$ picks the remaining hashes. It will provide (free) oracle access to these hashes to Adversary.

(S3) **Adversary plugin acquisition and bypassing.** $\mathcal{A}$ is given the chance to Acquire and Bypass any plugins it desires.
  (a) $\mathcal{C}$ initializes an empty set $\mathcal{B}$ of bypassed plugins.
  (b) Many times, $\mathcal{A}$ can choose to Acquire a plugin i. This grants $\mathcal{A}$ oracle access to p_i if it didn't have it already. (This step is required before $\mathcal{A}$ can query a local plugin.)
  (c) Many times, $\mathcal{A}$ can choose to Bypass a plugin i. This represents the adversary finding some algorithmic weakness in p_i so that they do not have to query it in order to get the result.

The indifferentiability game completes with a test as to whether the adversary can distinguish between an interaction with the real password-based key derivation function $\mathcal{H}$ and the real hash functions $\{h_i\}$ or with a random oracle $\mathcal{R}$ and a simulated version $\mathcal{S}^{\mathcal{R}}$ of the hash functions.

(I4) **Game Initialization**
  (a) $\mathcal{C}$ picks a bit b at random.
  (b) If $b = 0$, it will provide access to the real $\mathcal{F}$ and the real set of $\{h_i\}$s.
  (c) If $b = 1$, it will provide access to the random oracle $\mathcal{R}$ and simulators $\mathcal{S}_i^{\mathcal{R}}$ that simulate the hash functions.

(I5) **Query Phase**.
  (a) $\mathcal{A}$ makes up to q queries, with runtime bounded by t, to both the oracles it has been provided.

(I6) **Guess Phase**.
  (a) $\mathcal{A}$ outputs b'.
  (b) The adversary wins if $b' = b$.

The resource game completes with a test of whether the adversary can perform a dictionary attack at lower cost than a straightforward query of each password in the dictionary. More precisely, $\mathcal{A}$ wins if she can make fewer than one oracle query per plugin per password attempt. Later, in Section 4, we will use this minimum bound on different categories of adversary resources to relate the adversary's resource consumption to an honest user's resource consumption and show that for well-chosen plugins, the sometimes-exponential advantage granted by hardware specialization can be reduced to a linear advantage.

Let plugins be the set of all (stateless?) pseudorandom function families, and let $\mathcal{L}$ be the set of all *local* plugins. Local plugins are those that can only be accessed after the adversary does an Acquire action on them.

(R4) **State Setup Phase**
  (a) $\mathcal{C}$ chooses a salt and out_len. It gives these to $\mathcal{A}$.
  (b) $\mathcal{A}$ chooses a dictionary Dict of passwords it will attempt to brute-force. $\mathcal{A}$ also controls the size d of this dictionary. It passes this dictionary to $\mathcal{C}$.

(R5) **Query Phase**.
  (a) $\mathcal{C}$ initializes a query count for each plugin: $q_i = 0$ for all $i \in [m]$.
  (b) $\mathcal{A}$ queries $\mathcal{C}$ for different p_i oracles as many times as it wishes.
  (c) For each query $\mathcal{C}$ receives, it increments the appropriate query counter q_i by 1.

(R6) **Guess Phase**.
  (a) $\mathcal{A}$ outputs (Dict, Keys).
  (b) The adversary wins if the following two win conditions are met:
    (i) The keys are correct. That is, $\forall i \in [d]$,

$$\mathcal{F}^{h_1,\ldots,h_m}(\mathsf{Dict}[i],\mathsf{salt},\mathsf{out\_len},\mathsf{plugins}) = \mathsf{Keys}[i]$$

    (ii) The adversary did not spend sufficient resources; namely $\mathcal{A}$ made fewer than s queries to at least one nonbypassed plugin. Equivalently, $\exists q_i$ such that $q_i < d$ and $i \notin \mathcal{B}$.

2.3 Resource Cost Ratio

One possible criticism of the multi-resource approach is that it dilutes the effectiveness of some "optimal" PBKDF. If an ideal algorithm resistant to all specialized hardware were known, we would agree that this should be used. In fact, if the PBKDF is being computed on highly specialized software, then the defender is moderately safe using that one plugin, at least until other hardware catches up. However, in almost all cases, we do not know of a function that is generically resource-agnostic to being run on different

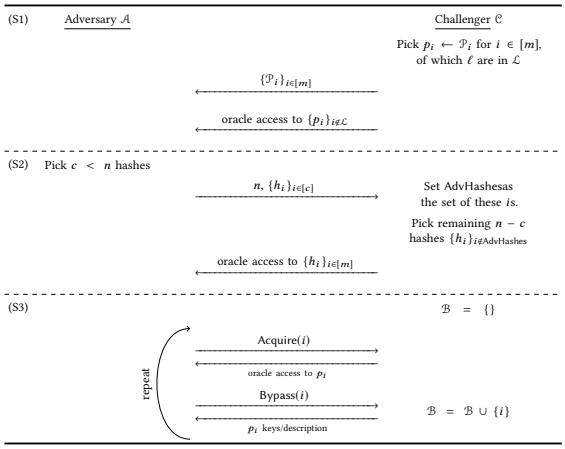

Figure 3: The setup phase for both the indifferentiability game and the resource game. The adversary selectively compromises a subset of hash functions and resource-consuming plugins, then attempts a dictionary attack using fewer resources than a brute-force attack would require.

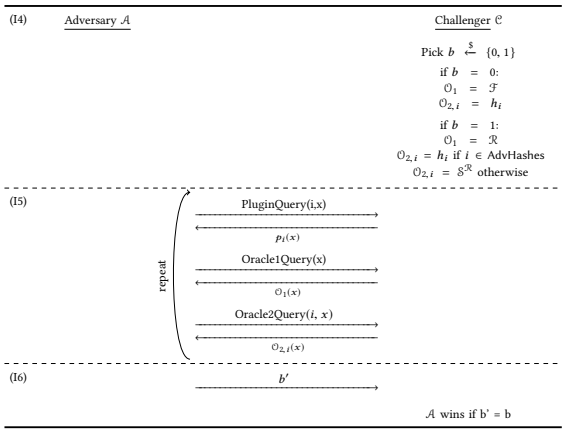

Figure 4: The indifferentiability game.

hardware. Even in the specialized hardware use case, we believe that one should not put all of one's proverbial cryptographic eggs into one basket. We argue that it is worth taking a linear hit to our advantage to buy ourselves the guarantee that attackers' advantage is not *more* than linear.

We can encode the resources a party possesses for computing these different plugin functions in a "cost vector" that represents how much effort it takes that party to compute the plugin once. A lower number means that party is better at computing that plugin. So the defender's normal laptop computer might have a cost vector of [1,1,1,1,1], but an adversary with specialized hardware for some of the functions might have a cost vector of [0.000001, 0.5, 0.000001, 1, 0.000001]. So our honest user above has a total cost of 5, whereas the specialized adversary had a cost of 1.5000003.

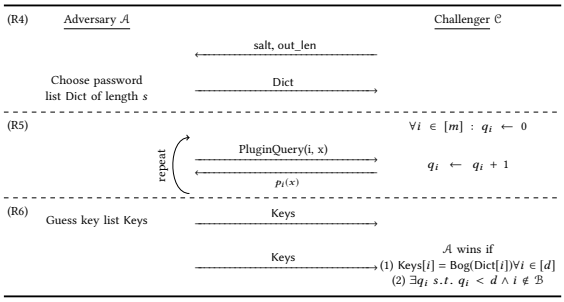

Figure 5: The resilient PBKDF security game. The adversary selectively compromises a subset of hash functions and resource-consuming plugins, then attempts a dictionary attack using fewer resources than a brute-force attack would require.

We can bound the cost ratio of the adversary to the defender. If the defender cost (1) is better than the attacker cost in a of m plugins, and (2) the defender costs are "well-balanced" (similar to each other), then:

$$\text{attacker cost} \geq \frac{a}{m}(\text{defender cost})$$

In other words, in the worst case, when the defender only has an advantage in one plugin, the attacker has at most a linear advantage over them. This is a much safer option than the many-orders-of-magnitude increase offered by specialized hardware for one plugin.

The resource vectors mentioned above also help clarify the cryptographic resource game described in the previous section. An adversary's cost for computing *localized* functions that they have not yet acquired can be represented as a prohibitively high number in the cost vector; calling Acquire on the plugin reduces that cost down to at or below the cost for the honest user. The Bypass action for an adversary is meant to encode the event where an adversary is so good at computing that function that it is effectively free; say, because they had a backdoor into the function, or because they learned a key or secret that was previously unknown to them.

The resource cost of computing a resilient PBKDF is the sum of all Bypassed costs, and for a well-balanced defender, the cost ratio between attacker and defender cost is at leas/t a/m.

3 BOG CONSTRUCTION

Bog uses a sponge function construction with two fundamental differences. First, the message is replaced with plugins that consume resources based on the output of the previous round. Second, the inner function is replaced with a hash combiner to provide resilience: the overall function acts as a random oracle as long as at least one of its constituent hash functions do.

The random oracle property of the hash combiner (see section 3.3) ensures that the plugin's input cannot be known until the previous round has completed, thus, resource consumption must be sequential. The alternation between the hash combiner and resource consuming plugins repeats until sufficiently many calls to the resource-consuming plugins have been made. A scheduler

determines the ordering of the plugins, using an auxiliary output of the hash combiner.

3.1 Resource-Consuming Plugins

There are many ways that a single system could combine resources to create a PBKDF, making any number of tradeoffs. It's difficult to pick a single best plugin for a resilient PBKDF, since specialized hardware offers significant resources bonuses in computing these, and the situation changes over time. Instead, as described by the approximate locality requirement and encoded in the game, many plugins should be chosen, to force an attacker to be specialized in *all* of them, and at that point.

In Bog, *resources* can mean anything from time to cache space to storage access to calls to a chip. Users should pick plugins that they believe they have an advantage in. For example, a user doing full disk encryption with Bog who wishes to defend themselves against adversaries with dedicated hardware like ASICs would use plugins that call a diverse set of instructions, accesses to the full (encrypted) disk, many branching computation paths, and a lot of memory. Most laptops are designed to call many instructions in many different orders, and have a lot of memory and storage in comparison to an ASIC, which is better at doing a single computation repeatedly, in a pipelined, parallel manner, without many accesses to memory or storage. No plugin is a foolproof function that proves that the calculation was done on the honest device, but each plugin partially localizes the computation to the honest device, such that the *combination* of plugins creates a function that is only efficiently computable on the honest device or on a device that is functionally equivalent to it. In the remainder of this section, we first describe the functionality and security requirements of plugins and then detail the different types of resources that they might consume.

3.1.1 Objectives. The interface to the resource-consuming plugins is quite simple – they just have an input (guaranteed to be pseudorandom by the hash combiner), and an output. A "good" plugin will be unpredictable without knowing both the input to the plugin (from the previous hash combiner output) and without spending a certain amount of resources. If a plugin is malicious or broken, then the output may be completely non-random regardless of what the input is. However, even if a plugin is malicious or broken, Bog as a whole still retains its security guarantees. The only thing that is lost due to a malicious plugin is the resource consumption requirement for that single plugin. Basically, the damage a broken plugin can do is contained to that one plugin.

The main goal of a good plugin is unpredictability - the output should be dependent on both the input and the resources spent. The output of the plugin should be unpredictable unless the input is known and the resources are spent. With that as the goal, different plugins may have vastly different designs. We list some candidate plugins in Section 3.1.2.

The plugins of Bog are meant to be highly customizable. If a device has access to keys baked into the hardware, or a Trusted Platform Module, or similar, then that is a very strong localization signal that should be incorporated into Bog. But if the honest device does not have any such resource, it should still be feasible to choose a set of plugins such that that kind of device is the only efficient type of device capable of computing the key. In this case, Bog's localization is preventing adversaries from having hardware advantages over the honest device. In short, the choice of plugins should be determined based on the desired use case and the resources possessed by the honest device. In the next section, we describe example resources and plugins that consume them.

3.1.2 Resources Utilized. We list here several resources that various machines might have, and we list concrete plugins that can be chosen to consume each of them. We use existing implementations for these plugins and simply write glue code to make them conform to Bog's interface.

Trusted Platform Module. If the honest device has a TPM, it can be used as a strong piece of evidence that the computation is occurring on that specific device. The TPM can provide input into the key generation process, ensuring that the key was either generated on the honest device or the TPM was somehow broken or bypassed.

Disk storage. The (encrypted) storage on the disk itself can be used as a way to localize the device. A Proof of Retrievability [58] over the encrypted disk would show that the party computing the password has access to the data on the device itself, reducing the likelihood that the attack is being conducted from a location other than the honest device.

Similarly, a Universal 2nd Factor (U2F) token such as a Yubikey serves the same purpose, but it is held externally by the user logging in rather than being internal to the device.

Network. Network resources may be appropriate tools to check the validity of the key generation process in some settings. In addition to validating connection to a specific network or rate-limiting login attempts, a service such as the Pythia PRF service [34] could be used to ensure that the device was connected to the network as desired.

Memory. A *memory-hard* function such as `scrypt` [6] or argon2d [18] imposes a trade-off between memory and time: the more memory available to the calculation, the faster the computation completes. In a heavily parallel computation of Bog, such as would be done on an ASIC, computing a memory-hard function means that the computation either takes longer, consuming time and CPU cycles as a resource, or must pay the additional cost of giving each core some memory it can use to perform the computation quickly.

We note that memory-hardness does not set a hard requirement. Memory-hardness imposes a trade-off between memory taken and time used, it does not set a lower bound on the amount of memory required. Still, we can consider a memory-hard function like scrypt to force the consumption of either memory or time.

The use of one memory-hard function does not mean that the entire Bog function is memory-hard, as different Bog instances can use one shared chunk of memory – one instance can use the memory while another one is computing a different plugin.

Cache. The layout and behavior of the cache can be both architecture and system specific. A function can use knowledge of the cache present on a particular system and optimize its performance on, say, an Intel x86 architecture with 2MB of L2 cache. By using the cache in an architecture-specific way, we can reduce the advantage of an ASIC over a general CPU by forcing the ASIC to either run

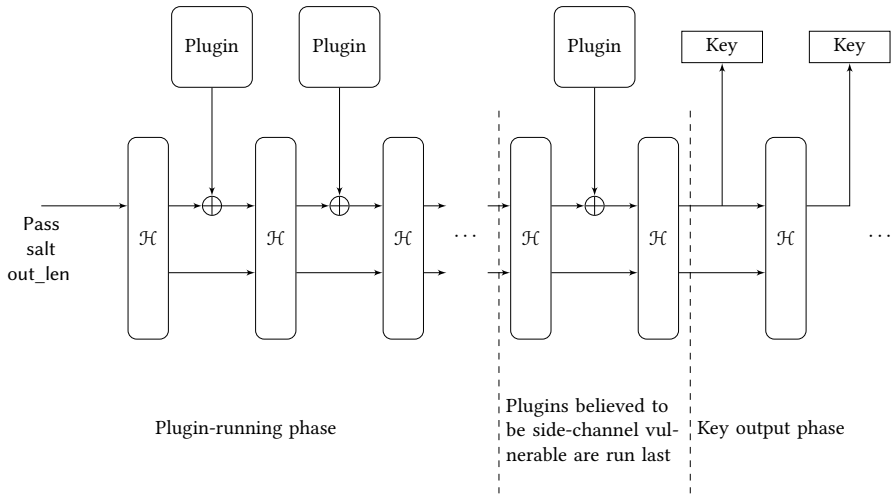

Figure 6: Bog as a sponge function.

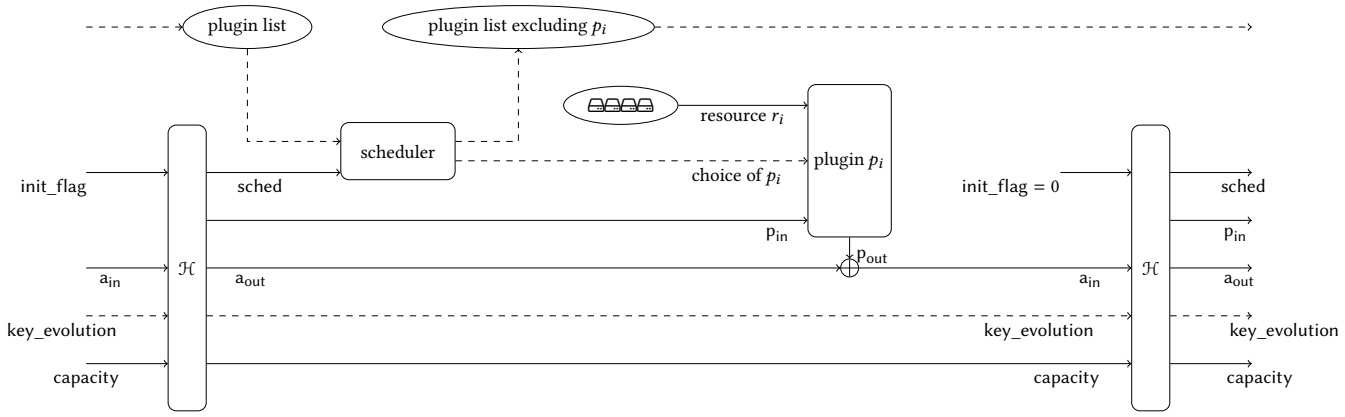

Figure 7: A round of Bog begins with the state output from the previous hash combiner. This output is split into sections of pre-determined, constant length: the sched bits are used to determine the next plugin to run, the p_{in} bits are used as input to the next plugin, the a_{out} bits are XORed with the output of the plugin (p_{out}) to produce a_{in}, and the capacity bits are left as-is. In section 3.3, we introduce the key_evolution bits, which are used to evolve the keys for the hash functions used within the hash combiner $\mathcal{H}$. Passed on as input to the next iteration of $\mathcal{H}$ is the concatenation of the init_flag (set to 0), a_{in}, key_evolution, and capacity.

much slower or to dedicate more of its transistor count to cache. Argon2d [18] is optimized for Intel x86 caching architecture.

Instruction diversity and branching paths. An advantage of general CPUs over ASICs is their ability to efficiently deal with branching paths in computation and their ability to deal with many different instructions. If the instructions actually change from a large set based on the input data, then pipelining is much less effective because the pipeline changes in a data-dependent way. The next instruction would not be known until the data were present, negating the hardware advantage of an ASIC.

Bandwidth. Related to memory-hard functions, *bandwidth-hard* functions [54] try to reduce the performance advantage for dedicated hardware by increasing RAM accesses. Energy-wise, memory accesses on ASICs are not significantly more memory-efficient than on general CPUs.

Variable	Purpose	Length
sched	Random input to scheduler to determine next plugin	Short
p_{in}	Random input to plugin	Medium-long
a_{out}	XOR with plugin output p_{out} to make a_{in}	Medium-long
key_evolution	Evolve keys in $\mathcal{H}$, see section 3.3	Medium
capacity	Provide security like the capacity bits in a sponge function	Long

Figure 8: Intermediate state in Bog

Parallelism. Plugins utilizing parallelism should optimize to the number of cores possessed by the honest device. A device possessing fewer cores should take longer to correctly compute the result, and a device with more cores should not be able to benefit from their extra cores.

Biometrics. Biometrics are a slightly different flavor of plugin, since they are not trying to localize the computation to the device, but rather identify the user logging in. Nevertheless, they can serve as an independent nature of authentication, especially on mobile devices that often include such sensors. Additionally, it is possible to extract unpredictable output from such biometrics [23, 59].

3.2 Sponge Construction

Objective. Bog 'glues' together all resource-consuming plugins via a sponge construction originally proposed by Bertoni et al. [16]. As shown in Figure 6, the main goal of the sponge construction is to ensure that the resources of *all* the plugins must be used, sequentially, in order to compute the correct password in the end.

Instantiation. In order to do this, Bog incorporates rounds of plugin outputs with calls to a hash combiner that is IRO as long as at least one hash function within it is IRO. In each round, the output of the previous hash combiner is split into parts, which are illustrated in Figure 7 and described in Figure 8. The sched bits are used as a random input to the scheduler, which determines which resource-consuming plugin will be run next. The p_{in} bits are provided to the chosen plugin as pseudorandom input. The a_{out} bits are XORed with the output of the plugin, p_{out}, and the resulting value, a_{in}, will be used as part of the input to the next round of the hash combiner $\mathcal{H}$. The remaining bits are used as capacity bits, which are also passed on to the next round of $\mathcal{H}$. The number of capacity bits determine the security parameter of the sponge function.

Also shown in Figures 7 and 8 are the key_evolution bits. These are used to change the keys for the hash functions in the next call to $\mathcal{H}$. We discuss this further in Section 3.3.

This model is meant to be highly customizable, meaning that the exact bit-lengths of each of these parts is left up to the user, but we provide test parameters used in section 5.2.

As discussed further in section 3.4, the ordering of plugins is designed to be modified based on all the previous computation done. This is an anti-pipelining defense, forcing the adversary's system to be able to run its various functions in many orders, and preventing full utilization of the attacker's system since they must now handle unpredictable scheduling of plugins.

In the specifications for the scheduler, a user can specify the number of each plugin they would like to run. This is similar to choosing the number of rounds in PBKDF2, but this does more than just setting the total runtime. In addition to approximating the total runtime of Bog, specifying values for each specific plugin allows the user to modify Bog to use resources that they believe they have an advantage in over the adversary. In our disk encryption motivating example, this means that a user wishing to protect their laptop with Bog-powered full disk encryption against adversaries with ASICs would use more plugins that utilize the disk, its memory, and its ability to perform varying instructions in any order (compared to an ASIC's highly pipelined, highly parallelized model).

Note that in our construction, we are assuming that the plugins are "costly" in some meaningful way and that the hash combiner is relatively "cheap." In particular, we claim that the amount of time taken by the hash combiner is short compared to the amount of time taken by the plugins. We present a full analysis in section 5, the takeaway of which is that though this assumption does not hold if too many hash functions are being used in the combiner, it should always hold for the numbers of hash functions and plugin runtimes that we think are reasonable.

3.3 Hash combiner

Objective. The purpose of $\mathcal{H}$ in Bog is to force the plugins to be run in sequential order. As discussed in section 3.1, the output of a good plugin will be unpredictable based on both its input and the resources consumed. Thus, there is no point to running a plugin before the input p_{in} is known. And, due to the properties of the sponge function construction, if $\mathcal{H}$ acts as a random oracle, then p_{in} cannot be known until all previous rounds have been run.

The design of a sponge function relies on this inner function being indifferentiable from random oracle. If a single keyed hash function is trusted enough to suit this purpose, then $\mathcal{H}$ could be instantiated using a single hash function.

But there is disagreement as to which hash function should be trusted enough to fill this role as different users may trust the design of these hash functions to different extents. In our implementation, we use a hash combiner that provides the desired properties of a hash function provided that at least one of the input hash functions satisfies the property, even if it is not known which hash function satisfies it.

If multiple hash functions are combined incorrectly, their security properties can actually be reduced. For example, an XOR combination of two hash functions can eliminate collision resistance, and concatenation of the function outputs can break pseudorandomness [21, 36] if one of the hashes is broken or malicious.

Instantiation. The construction F_2 we use, created by Fischlin et al. [38], provably preserves many properties including indifferentiability from a random oracle (cf. Definition 4.1) between two hashes as long as one of them has the IRO property.

In order to combine more than two hash functions with equal-length outputs, we form a binary tree of F_2 constructions. Once

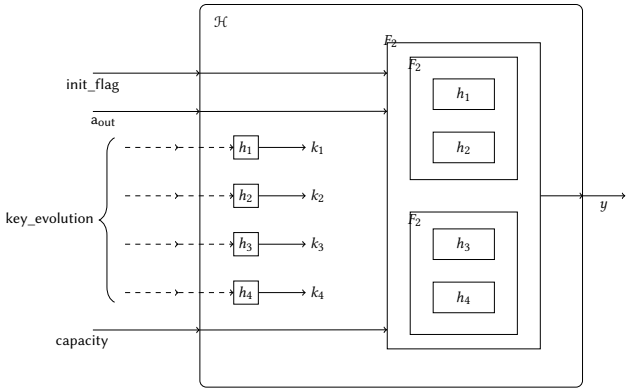

Figure 9: The inputs and functionality of $\mathcal{H}$. The key_evolution bits are split into equal-length parts and passed individually into the hash functions h_i. This creates a new key k_i to be used as the key the next time h_i is run. Separately, the init_flag, a_{out}, and the capacity bits are passed as input to the hash combiner F_2, which is called recursively in a tree structure to combine all the hash functions in a way that if one hash function h_i is IRO, then the entire F_2 construct is IRO as well.

again, the IRO guarantee composes gracefully over as many compositions of F_2 as desired. Unfortunately, the same cannot be said for performance: asymptotically, the output size grows exponentially in the number of hash functions used (which we denote by n). Nevertheless, this exponential growth rate is rather slow: approximately $(1.1)^n$. So, for a reasonable constant number of hash functions like $n = 4$ or $n = 8$, the combiner runs rather quickly. We tested this assumption on our sample implementation with 4 hash functions and learned that the overhead incurred by $\mathcal{H}$ is negligible for a couple hundred plugins, and grows to about 25% for 1000 plugins. See Section 5 for more details.

The key_evolution bits are used to rotate the keys. As part of $\mathcal{H}$, before the inputs are sent to F_2, the key_evolution bits are split into n parts, where n is the number of hash functions. Each part key_evolution_i is passed through the hash function h_i using the previous key, and the output is used to set the new key k_i for h_i.

The key evolution is conducted to prevent a malicious hash function from learning too much about the other hash functions' keys, which would violate the pre-condition required for $\mathcal{H}$ to function as a random oracle.

3.4 Scheduler

Objective. Our scheduler provides non-predictability and resistance to early confirmation. We allow the user to specify constraints on how many of each plugin should be run, and also to mark some of the plugins as vulnerable to side channels. The selection of a particular plugin for a round should leak no information as to whether this is the correct password or not. Adversaries should not be able to abort their password calculations early based on schedule choices. If the user believes that a plugin has side channels that would enable an early-confirmation attack, we allow that plugin to be run close to the end, so that the majority of the computation must still occur before the side-channel-vulnerable plugin is run.

Instantiation. We chose a simple method for our scheduler – sample randomly without replacement from the total list of plugins you'll run (that are not vulnerable to early confirmation). For example, if a user specifies that they want to run one plugin 10 times, another 5 times, and five more plugins 1 time each, then in the first round, the scheduler will pick one of those plugins randomly from a list of twenty, in the second round, it will pick one of the nineteen remaining functions, and so on, until there is only one option left.

Plugins that a user believes have side channels that make them potential avenues for early confirmation attacks are executed only in the final rounds of Bog, also in a random order according to the output of the hash combiner.

The scheduler's source of randomness is a subset of the bits returned by the hash combiner at each step. Since we are assured that these are pseudorandom, we simply use rejection sampling, calculating the index of the plugin mod the number of remaining plugins, and remove the selected plugin from the list.

4 BOG ANALYSIS

In this section, we rigorously prove that our Bog construction satisfies the definitions and requirements stated in Section 2.

First, we establish that Bog achieves a strong notion of cryptographic security called *indistinguishability from a random oracle*, or IRO, as long as one of its constituent hash functions is also IRO. We prove this assertion via the composition of two separate theorems, stated here informally.

Thm 4.2. If the inner function $\mathcal{H}$ within the sponge function is indifferentiable from a random oracle, then Bog is also IRO.

Thm 4.3. As long as at least one of the component hash functions h_i is indifferentiable from random oracle, then $\mathcal{H}$ is also IRO.

We stress that the first two theorems don't require the plugins to provide any cryptographic security. Ergo, even if all resource plugins are bypassed or backdoored, Bog still provides the same IRO guarantee as previous PBKDFs.

Second, we establish that Bog's output is unpredictable unless an attacker expends sufficient resources. In other words, we prove the following assertion, stated here informally.

Thm 4.5. If $\mathcal{H}$ is collision resistant, then an attacker must execute all unpredictable resource-consuming plugins in order to predict outputs of Bog,

We observe that this theorem provides graceful degradation of Bog against partial (but not complete) compromise in the hash functions: unpredictability may still apply even if none of the hash functions is IRO but at least one of them is collision resistant. However, if all hash functions are completely compromised (e.g., they equal the identity function), then none of our security guarantees hold.

4.1 Indifferentiability from a Random Oracle

Before we rigorously prove these cryptographic security properties about the Bog construction, we first define this target notion

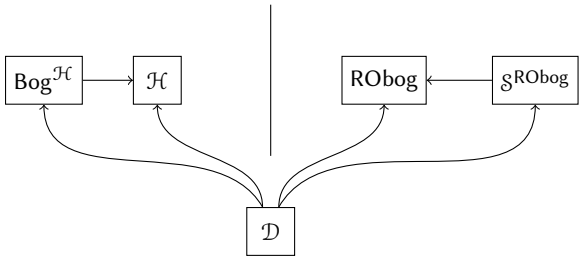

Figure 10: IRO game for Bog. Given $\mathcal{H}$, Bog$^{\mathcal{H}}$, and RObog, must produce $\mathcal{S}^{\text{RObog}}$ such that $\mathcal{D}$ cannot distinguish between the left and right sides.

of 'strength' for a password-based key derivation: the concept of *indifferentiability from a random oracle* (IRO) [46].

Intuitively, this notion states that a construction $\mathcal{F}^{\mathcal{H}}$ "properly leverages" its underlying random transformation $\mathcal{H}$ in order to produce an object that is "as good as" a fresh random oracle, even if $\mathcal{H}$ is public knowledge. Maurer et al. [46] capture this notion formally via a simulation-based definition that we state generically below and depict for the case of Bog in Figure 10.

Definition 4.1 (IRO [26, 46]). A construction $\mathcal{F}$ with oracle access to a random transformation $\mathcal{H}$ is said to be (q, t, ϵ)-*indifferentiable from a random oracle* $\mathcal{R}$ if there exists a simulator $\mathcal{S}^{\mathcal{R}}$ (with running time at most t per invocation) such that for any distinguisher $\mathcal{D}$ that makes at most q oracle queries it holds that:

$$|\Pr[\,\mathcal{D}(\,\mathcal{F}^{\mathcal{H}}, \mathcal{H}\,)\,] = 1 - \Pr[\,\mathcal{D}(\,\mathcal{R}, \mathcal{S}^{\mathcal{R}}\,)\,] = 1)| < \epsilon.$$

Note that the running time of $\mathcal{D}$ is unbounded.

Additionally, we say that $\mathcal{F}^{\mathcal{H}}$ is *indifferentiable from a random oracle* if there exist t polynomial in its input length, $q = \text{poly}(\lambda)$, and $\epsilon = \text{negl}(\lambda)$ such that the above statement holds, where λ denotes the security paramter.

We stress that the definition does *not* permit the simulator $\mathcal{S}$ to view the queries that $\mathcal{D}$ makes to $\mathcal{R}$, and yet $\mathcal{S}$ must still produce responses that are consistent with any queries that $\mathcal{D}$ could have made to $\mathcal{R}$. Additionally, we remark that all random oracles considered in this work will be keyed; however, for simplicity of notation we often omit the key parameter from the oracle's input.

Indifferentiability has desirable composition properties. Maurer et al. [46] showed that if $\mathcal{F}^{\mathcal{H}}$ is indifferentiable from a random oracle $\mathcal{R}$ of the same size, then $\mathcal{F}$ can replace $\mathcal{R}$ in any larger cryptosystem. In particular, this larger cryptosystem may even be another construction that leverages indifferentiability! Ergo: this composition property allows us to prove in stages that Bog is IRO.

4.2 Indifferentiability of Bog

The following theorem conveys the strength of the Bog sponge construction when instantiated with a keyed random oracle $\mathcal{H} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^L$.

Theorem 4.2. *If $\mathcal{H}$ acts as a random oracle, then Bog$^{\mathcal{H}}$ is (q, q^2, ϵ)-IRO where $\epsilon = \frac{q(q+1)}{2^{c+1}}$.*

The proof of this theorem uses similar concepts to Bertoni et al.'s proof of indifferentiability for the sponge construction [17]. Like the Bertoni et al. proof, we build a data structure for the simulator to discover when its queries are "connected" as part of a larger single Bog operation (i.e., that the output of one $\mathcal{H}$ is meant to be transformed by a plugin and then fed into another instantiation of $\mathcal{H}$).

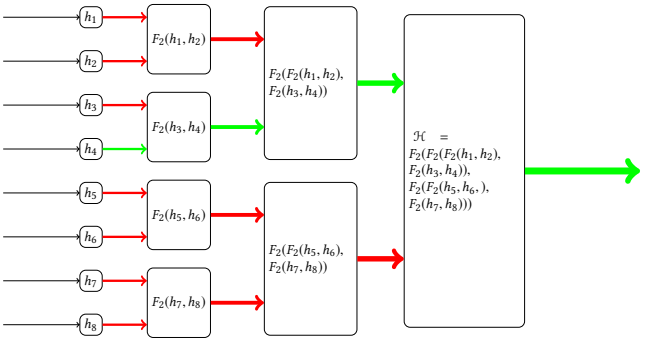

Figure 11: $\mathcal{H}$ maintains the properties of one good hash function even if all other hash functions are bad

However, some modifications are required to the Bertoni et al. argument in order to account for the design of the plugin system. The simulator in the sponge function construction has full control over the rate bits fed into the next invocation of $\mathcal{H}$ because it can choose the message to absorb. By contrast, our simulator $\mathcal{S}$ must design special plugins that can provide the right output, in the right order, to control the rate bits. The simulator needs this power in order to glean valuable information from RObog. We present a complete proof of Theorem 4.2 in Appendix A.

4.3 Resiliency from the Hash Combiner

Next, we demonstrate a theorem that combines all of the assertions from Section 3.3 about the relationship between the hash functions h_i, recursive applications of the Fischlin et al. hash combiner F_2, the key evolution process, and $\mathcal{H}$.

Theorem 4.3. *If at least one component hash function h_i acts as a random oracle, then the round construction $\mathcal{H}$ is IRO.*

Proof. Lemma 4.2 in Fischlin et al. [38] states that if the hash combiner is used to combine two hash functions, one of which acts as a keyed random oracle *where the key is unknown to the distinguisher and is chosen uniformly from the key space*, then the combiner's output is IRO.

Even though Bog begins with a low-entropy password, we meet the italicized condition within the initial round of Bog. In all future rounds, the keys were evolved by running the key_evolution bits through each of the hash functions h_i, as shown in figure 9. Assuming that one of the functions acted as a random oracle, then its output when run on its part of key_evolution (which, recall, was also indistinguishable from random) is indistinguishable from random, as shown in 11. Thus, the new key for this function was chosen in a way that was indistinguishable from random, and it is still unknown to $\mathcal{D}$. Therefore, all future rounds of Bog also meet this condition and the round construction $\mathcal{H}$ is always IRO. □

4.4 Consuming Resources

Unlike the other components of Bog, we do not rely upon the resource-consuming plugins to act as a random oracle. Instead, we view a plugin as good if its output is *unpredictable* to an adversary based upon the (partial) secrecy of its resources. For example, our plugin design is based on the principle that an adversary cannot introspect into the secrets held by a TPM, or obtain the response to a challenge sent to Pythia, or provide an appropriate biometric, etc. Or more accurately: it might be possible for an adversary to do any one of these actions, but Bog's resilient design only requires the premise that no attacker can compromise all of the plugin systems simultaneously.

We begin with a formal definition of unpredictability. Note that pseudorandom function families (as used in Definition 2.1) are unpredictable.

Definition 4.4 (Unpredictability). A keyed family of functions $\mathcal{P}$ is said to be (q, t, ϵ)-*unpredictable* if for every adversary $\mathcal{A}$ that makes at most q oracle queries and executes in time at most t,

$$\Pr[\mathcal{A}^P = (x, y) \text{ s.t. } P(x) = y \text{ and } \mathcal{A} \text{ did not query } P \text{ at point } x] < \epsilon,$$

where $P : \{0,1\}^a \to \{0,1\}^a$ is an instance of the unpredictable function family $\mathcal{P}$ chosen uniformly at random.

The following theorem demonstrates that Bog *opportunistically* leverages the unpredictability of all plugins that can provide this guarantee. Whereas the sponge construction guarantees (Theorems 4.2 and 4.3) rely on the collision resistance of the capacity bits, this theorem relies on the unpredictability of the a_{in}, a_{out}, and p_{out} wires; we let a denote their total length. Note that the following theorem only considers a combined round construction $\mathcal{H}$ that is IRO rather than its constituent pieces; we can do so because this guarantee composes with that of Theorem 4.3.

Theorem 4.5. *Let plugins be a set of m functions, of which a subset $T \subseteq$ plugins are (q, t, ϵ)-unpredictable. Additionally, let $\mathcal{A}$ be an adversary with running time t and with $q \geq m$ allowable calls to each of several oracles: a (q, t, ϵ)-collision resistant $\mathcal{H}$ along with all the plugins in T. Then, the probability that $\mathcal{A}$ can produce a password-key dictionary $(Dict, Keys = [Bog(w) : w \in Dict])$ is at most $2\epsilon|T|$ if there exists a plugin that $\mathcal{A}$ queries fewer than $|Dict|$ times.*

The proof of this theorem can be found in Appendix B. It operates in the nonprogrammable random oracle model.

5 IMPLEMENTATION

We implemented a proof of concept for Bog in Rust; it is available at [URL removed to preserve anonymity]. No intensive effort was made to optimize the code.

In this section, we present experimental results that validate two properties of our implementation. First, we demonstrate that the hash combiner's runtime is insignificant compared to that of the plugins. Second, we show that Bog performs well for parameter choices that we can reasonably expect to see in practice.

5.1 Time taken by hash combiner

We run the hash combiner a number of times equal to the number of plugins run. One consequence of this design is that if the hash combiner is computationally-intensive, it runs the risk of becoming an undesirable plugin itself. Thus, we require that the hash combiner should not take too long compared to the plugins. The plugins are meant to consume their resources in the desired login time (say, one second), and the hash combiner should not add very much time beyond that, or else hardware dedicated to the computation of Bog will once again have a resource advantage over the honest user.

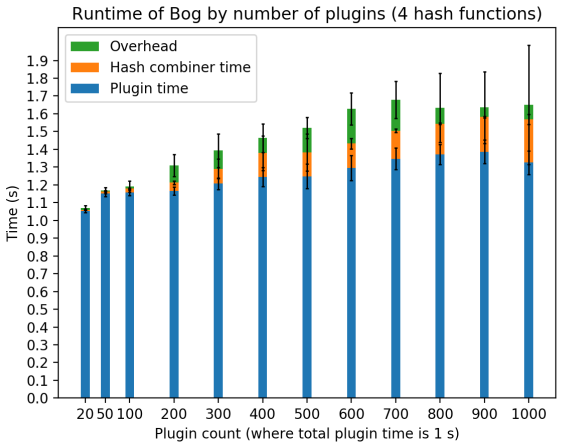

Figure 12: Runtimes for Bog as a function of the total number of plugins executed. The sum of all plugin runtimes are fixed at 1 second.

Nothing in our theoretical model prevents the hash functions from taking too long; this is a practical constraint. Each hash function call shouldn't take very much time, but the hash combiner can cause a large number of hash function calls to occur on more and more input.

Thus, as part of our analysis, we use our implementation to demonstrate that the hash combiner doesn't take too long. The results, for 4 hash functions, can be found in Figure 12. The benchmarks were run on a 2015 MacBook Pro with a 3.1 GHz i7 core and 16 GB of RAM. For this proof of concept, we did not run actual plugins; instead, several copies of a dummy plugin were created. This dummy plugin simply sleeps for a certain amount of time, to control the resource consumption for the purpose of benchmarking.

The benchmarks were run 300 times for each configuration. For fewer than 200 functions, the overhead incurred by the hash combiner is very low. Even for a thousand plugins, we still incur less than 25% overhead.

5.2 Parameter Choice

The hash combiner construction we used combines two hash functions at the cost of making its final output slightly more than double its length. To combine more hash functions than two involves creating a tree-like structure, resulting in an exponential increase in the input length. We tested our implementation using 4 and 8 hash functions. For four hash functions, each of which outputted 64 bytes initially, the intermediate state length was 274 bytes. This

n	sched	p_{in}	a_{in}	key_evolution	capacity	Total state len.
4	2 B	64 B	64 B	104 B	274 B	40 B
8	2 B	128 B	128 B	256 B	466 B	128 B

Figure 13: Parameter values used in 4- and 8-hash function implementation

was split into several parts, as discussed in section 3: 2 bytes were used for scheduler input, 64 bytes used for the plugin input and output, 104 for the "capacity bits" passed on to the next round, and 40 for key rotation of the hash functions within the hash combiner. A summary of parameters used is in Figure 13.

The number of hash functions affects the length of the intermediate state, because the more hash functions used, the more the hash combiner causes the output length to increase. The choice of number of hash functions determines the maximum size of the parameters of each portion of the state.

Much like in a sponge function, the choice of size for each component affects the security of the scheme. The capacity bits determine the security of the overall scheme. In our scheme, the plugin input/output bits function like the rate bits of a sponge function. This does not affect the security of the sponge function, but it does affect the guarantees of our plugins. We also use some bits of the state to schedule the next plugin and to rotate the keys of the hash functions in the hash combiner.

6 RELATED WORK

This work combines multiple avenues of research within applied cryptography and systems design.

PBKDFs & trusted hardware for disk encryption. The predominant use of the key derived from a PBKDF in practice today is as the encryption key for data saved in long-term storage. Support for sector-level and/or file-level disk encryption is built in natively to all modern desktop (Windows BitLocker [33], Mac OS FileVault, and Linux LUKS [39]) and smartphone (iOS [8] and Android [7]) operating systems. Many of these systems utilize a combination of software and hardware-localizing components; for instance, BitLocker leverages a TPM and iOS full disk encryption leverages a trusted Secure Enclave. There also exist several third-party software packages for full disk encryption, such as TrueCrypt [63] and VeraCrypt [65].

Hashing and key derivation. Our top-level Bog design is both inspired by and directly uses several innovations in the design of (password based) hash functions. Most directly, Bog's architecture follows the Keccak sponge function design from Bertoni et al. [16, 17], which has since been standardized by NIST as the SHA-3 [57]. We follow the lead of Keccak and many other hash functions by using Bellare and Rogaway's random oracle heuristic [15].

Additionally, there is a long history of competitions [10] and innovation in the space of password-based hashing. The first widespread protocols for password-based key derivation function were RSA Corporation's PBKDF1 and PBKDF2 [55], the latter of which has been standardized by NIST [64]. However, there exist attacks that can lower the cost advantage ratio of PBKDF2 via a partial algorithmic bypass [66] or by taking advantage of massively parallel hardware platforms like GPUs or ASICs [2, 30, 44].

In response to these concerns, researchers developed new functions like argon2, bcrypt, and scrypt (cf. §3.1.2) with features that are difficult to emulate in massively-parallel systems, such as high memory consumption. These newer functions achieve higher, though not always optimal, cost advantage ratios [31, 45, 67]. We advocate for the use of several such systems as plugins within Bog (cf. §3.1.2). One differentiator of Bog from prior PBKDFs is our ability to bring concepts from multi-factor security to hash functions: our plugin system allows the hash function to leverage who you are (e.g., [23, 59]), what you have (e.g., [8, 41]), and what other people on the network can vouch for you (e.g., [24, 34]). See section 3.1 for details.

Finally, some works have systematized PBKDF research into lists of descriptive criteria that desirable functions should provide [22, 25]. Our definition in §2 combines and extends several of their cryptographic and systems security constraints.

Hash combiners. At the level of round transformations, our work uses the theory of combiners to provide resilience against ineffective or backdoored hash functions. Initiated by Boneh and Boyen [21] and Pietrzak [51], this field was initially dominated by negative results that showed the difficulty of making collision-resistant hash functions with small output size. Mittelbach [48] and others [11, 28] extended these impossibility results to cover other desirable cryptographic properties like (second) preimage resistace. The difficulty of building cryptographic objects with small output length is a strong motivator for our decision to use hash combiners with sponge functions since, unlike the Merkle-Damgard paradigm [47], sponge functions actually require large intermediate state to achieve collision resistance and they can choose an output length independent of the intermediate state size.

Concretely, our hash function combiner draws heavily from the work of Hoch and Shamir [42] and Fischlin et al. [36, 37], whose constructions permit composition of several functions to achieve either the best security guarantee of any component function or sometimes a guarantee that is stronger than any individual constituent. Additionally, we combine these techniques with ideas from leakage resilience [32] in order to design keyed cryptosystems whose security cannot be compromised even if malicious hash functions observe some information about the secret state of good hash functions.

Subversion resilience. Finally, throughout the design process, our work is inspired by the recent research thrust into subversion-resilient cryptography. These cryptosystems are designed to withstand algorithmic flaws such as bad (pseudo)random number generators [13, 27, 29] or maliciously chosen public parameters for public key cryptosystems [3, 9, 12, 14, 40].

REFERENCES

[1] 2018. Asic Miner Value. (2018). https://www.asicminervalue.com/.

[2] Ayman Abbas, Rian Voss, Lars Wienbrandt, and Manfred Schimmler. 2014. An efficient implementation of PBKDF2 with RIPEMD-160 on multiple FPGAs. In *20th IEEE International Conference on Parallel and Distributed Systems*. IEEE Computer Society, 454–461. https://doi.org/10.1109/PADSW.2014.7097841

[3] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michal Zajac. 2017. A Subversion-Resistant SNARK. In *ASIACRYPT 2017, Part III (LNCS)*, Tsuyoshi Takagi and Thomas Peyrin (Eds.), Vol. 10626. Springer, Heidelberg, 3–33.

[4] Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.). 2008. *ICALP 2008, Part II*. LNCS, Vol. 5126. Springer, Heidelberg.

[5] Oleg Afonin. 2018. iOS 11.4 to Disable USB Port after 7 Days: What It Means For Mobile Forensics. (2018).

[6] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. 2017. Scrypt Is Maximally Memory-Hard. In *EUROCRYPT 2017, Part II (LNCS)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.), Vol. 10211. Springer, Heidelberg, 33–62.

[7] Android Open Source Project. 2018. Encryption. (2018). https://source.android.com/security/encryption.

[8] Apple, Inc. 2018. *iOS Security Guide*. Technical Report. https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[9] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. 2015. Subversion-Resilient Signatures: Definitions, Constructions and Applications. Cryptology ePrint Archive, Report 2015/517. (2015). http://eprint.iacr.org/2015/517.

[10] Jean-Philippe Aumasson, Tony Arcieri, Dmitry Chestnykh, Jeremi Gosney, Russell Graves, Matthew Green, Peter Gutmann, Pascal Junod, Poul-Henning Kamp, Stefan Lucks, Samuel Neves, Colin Percival, Alexander Peslyak, Marsh Ray, Jens Steube, Steve Thomas, Meltem Sönmez Turan, Zooko Wilcox-O'Hearn, Christian Winnerlein, and Elias Yarrkov. 2015. Password Hashing Competition. (December 2015). https://password-hashing.net/

[11] Zhenzhen Bao, Lei Wang, Jian Guo, and Dawu Gu. 2017. Functional Graph Revisited: Updates on (Second) Preimage Attacks on Hash Combiners. Cryptology ePrint Archive, Report 2017/534. (2017). http://eprint.iacr.org/2017/534.

[12] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. 2016. NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion. In *ASIACRYPT 2016, Part II (LNCS)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10032. Springer, Heidelberg, 777–804. https://doi.org/10.1007/978-3-662-53890-6_26

[13] Mihir Bellare and Viet Tung Hoang. 2015. Resisting Randomness Subversion: Fast Deterministic and Hedged Public-Key Encryption in the Standard Model. In *EUROCRYPT 2015, Part II (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, Heidelberg, 627–656. https://doi.org/10.1007/978-3-662-46803-6_21

[14] Mihir Bellare, Bertram Poettering, and Douglas Stebila. 2017. Deterring Certificate Subversion: Efficient Double-Authentication-Preventing Signatures. In *PKC 2017, Part II (LNCS)*, Serge Fehr (Ed.), Vol. 10175. Springer, Heidelberg, 121–151.

[15] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM CCS 93*, V. Ashby (Ed.). ACM Press, 62–73.

[16] Guido Bertoni, J Daemen, Michaël Peeters, and Gilles van Assche. 2007. Sponge functions. Ecrypt Hash Workshop. (2007).

[17] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. 2008. On the Indifferentiability of the Sponge Construction. In *EUROCRYPT 2008 (LNCS)*, Nigel P. Smart (Ed.), Vol. 4965. Springer, Heidelberg, 181–197.

[18] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. 2016. Argon2: new generation of memory-hard functions for password hashing and other applications. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 292–302.

[19] Bitcoin Wiki. 2014. https://www.emc.com/collateral/white-papers/h11302-pkcs5v2-1-password-based-cryptography-standard-wp.pdf. (2014). https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison.

[20] Bitcoin Wiki. 2018. Bitcoin Mining Comparison. (2018). https://en.bitcoin.it/wiki/Mining_hardware_comparison.

[21] Dan Boneh and Xavier Boyen. 2006. On the Impossibility of Efficiently Combining Collision Resistant Hash Functions. In *CRYPTO 2006 (LNCS)*, Cynthia Dwork (Ed.), Vol. 4117. Springer, Heidelberg, 570–583.

[22] Milan Brož and Vashek Matyáš. 2015. Selecting a New Key Derivation FunGction for Disk Encryption. In *Security and Trust Management*, Sara Foresti (Ed.). Springer International Publishing, Cham, 185–199.

[23] Ran Canetti, Benjamin Fuller, Omer Paneth, Leonid Reyzin, and Adam D. Smith. 2016. Reusable Fuzzy Extractors for Low-Entropy Distributions, See [35], 117–146. https://doi.org/10.1007/978-3-662-49890-3_5

[24] Nishanth Chandran, Vipul Goyal, Ryan Moriarty, and Rafail Ostrovsky. 2009. Position Based Cryptography. In *CRYPTO 2009 (LNCS)*, Shai Halevi (Ed.), Vol. 5677. Springer, Heidelberg, 391–407.

[25] Donghoon Chang, Arpan Jati, Sweta Mishra, and Somitra Kumar Sanadhya. 2015. Rig: A Simple, Secure and Flexible Design for Password Hashing. Cryptology ePrint Archive, Report 2015/009. (2015). http://eprint.iacr.org/2015/009.

[26] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. 2005. Merkle-Damgård Revisited: How to Construct a Hash Function. In *CRYPTO 2005 (LNCS)*, Victor Shoup (Ed.), Vol. 3621. Springer, Heidelberg, 430–448.

[27] Jean Paul Degabriele, Kenneth G. Paterson, Jacob C. N. Schuldt, and Joanne Woodage. 2016. Backdoors in Pseudorandom Number Generators: Possibility and Impossibility Results. In *CRYPTO 2016, Part I (LNCS)*, Matthew Robshaw and Jonathan Katz (Eds.), Vol. 9814. Springer, Heidelberg, 403–432. https://doi.org/10.1007/978-3-662-53018-4_15

[28] Itai Dinur. 2016. New Attacks on the Concatenation and XOR Hash Combiners, See [35], 484–508. https://doi.org/10.1007/978-3-662-49890-3_19

[29] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. 2015. A Formal Treatment of Backdoored Pseudorandom Generators. In *EUROCRYPT 2015, Part I (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9056. Springer, Heidelberg, 101–126. https://doi.org/10.1007/978-3-662-46800-5_5

[30] Markus Dürmuth, Tim Güneysu, Markus Kasper, Christof Paar, Tolga Yalcin, and Ralf Zimmermann. 2012. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In *ESORICS 2012 (LNCS)*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.), Vol. 7459. Springer, Heidelberg, 716–733.

[31] Markus Dürmuth and Thorsten Kranz. 2014. On Password Guessing with GPUs and FPGAs. In *Technology and Practice of Passwords - International Conference on Passwords (Lecture Notes in Computer Science)*, Vol. 9393. Springer, 19–38. https://doi.org/10.1007/978-3-319-24192-0_2

[32] Stefan Dziembowski and Krzysztof Pietrzak. 2008. Leakage-Resilient Cryptography. In *49th FOCS*. IEEE Computer Society Press, 293–302.

[33] Shon Eizenhoefer. 2006. BitLocker Drive Encryption Hardware Enhanced Data Protection. (2006). https://download.microsoft.com/download/5/b/9/5b97017b-e28a-4bae-ba48-174cf47d23cd/cpa064_wh06.ppt

[34] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, Thomas Ristenpart, and Cornell Tech. 2015. The Pythia PRF Service.. In *USENIX Security Symposium*. 547–562.

[35] Marc Fischlin and Jean-Sébastien Coron (Eds.). 2016. *EUROCRYPT 2016, Part I*. LNCS, Vol. 9665. Springer, Heidelberg.

[36] Marc Fischlin and Anja Lehmann. 2008. Multi-property Preserving Combiners for Hash Functions. In *TCC 2008 (LNCS)*, Ran Canetti (Ed.), Vol. 4948. Springer, Heidelberg, 375–392.

[37] Marc Fischlin, Anja Lehmann, and Krzysztof Pietrzak. 2008. Robust Multi-property Combiners for Hash Functions Revisited, See [4], 655–666.

[38] Marc Fischlin, Anja Lehmann, and Krzysztof Pietrzak. 2014. Robust Multi-Property Combiners for Hash Functions. *Journal of Cryptology* 27, 3 (July 2014), 397–428. https://doi.org/10.1007/s00145-013-9148-7

[39] Clemens Fruhwirth. 2005. New methods in hard disk encryption. (07 2005). http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf

[40] Georg Fuchsbauer. 2017. Subversion-zero-knowledge SNARKs. Cryptology ePrint Archive, Report 2017/587. (2017). http://eprint.iacr.org/2017/587.

[41] Trusted Computing Group. 2007–2017. Trusted Platform Module (TPM). (2007–2017). https://trustedcomputinggroup.org/work-groups/trusted-platform-module/.

[42] Jonathan J. Hoch and Adi Shamir. 2008. On the Strength of the Concatenated Hash Combiner When All the Hash Functions Are Weak, See [4], 616–630.

[43] Lars R. Knudsen, John Erik Mathiassen, Frédéric Muller, and Søren S. Thomsen. 2010. Cryptanalysis of MD2. *Journal of Cryptology* 23, 1 (Jan. 2010), 72–90.

[44] Xiaochao Li, Chunhui Cao, Pengtao Li, Shuli Shen, Yihui Chen, and Lin Li. 2016. Energy-Efficient Hardware Implementation of LUKS PBKDF2 with AES on FPGA. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 402–409. https://doi.org/10.1109/TrustCom.2016.0090

[45] Katja Malvoni, Solar Designer, and Josip Knezovic. 2014. Are Your Passwords Safe: Energy-Efficient Bcrypt Cracking with Low-Cost Parallel Hardware. In *8th USENIX Workshop on Offensive Technologies*. USENIX Association. https://www.usenix.org/conference/woot14/workshop-program/presentation/malvani

[46] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. 2004. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In *TCC 2004 (LNCS)*, Moni Naor (Ed.), Vol. 2951. Springer, Heidelberg, 21–39.

[47] Ralph Charles Merkle. 1979. *Secrecy, authentication, and public key systems*. Ph.D. Dissertation. Stanford University.

[48] Arno Mittelbach. 2012. Hash Combiners for Second Pre-image Resistance, Target Collision Resistance and Pre-image Resistance Have Long Output. In *SCN 12 (LNCS)*, Ivan Visconti and Roberto De Prisco (Eds.), Vol. 7485. Springer, Heidelberg, 522–539.

[49] Moni Naor (Ed.). 2007. *EUROCRYPT 2007*. LNCS, Vol. 4515. Springer, Heidelberg.

[50] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions. *Self-published* (2009), 1–16.

[51] Krzysztof Pietrzak. 2007. Non-trivial Black-Box Combiners for Collision-Resistant Hash-Functions Don't Exist, See [49], 23–33.

[52] Niels Provos and David Mazieres. 1999. Bcrypt algorithm. USENIX.

[53] Thomas Reed. 2018. GrayKey iPhone unlocker poses serious security concerns. (2018).

[54] Ling Ren and Srinivas Devadas. 2017. Bandwidth Hard Functions for ASIC Resistance. Cryptology ePrint Archive, Report 2017/225. (2017). http://eprint.iacr.org/2017/225.
[55] RSA Laboratories. 2012. https://www.emc.com/collateral/white-papers/h11302-pkcs5v2-1-password-based-cryptography-standard-wp.pdf. (2012). https://www.emc.com/collateral/white-papers/h11302-pkcs5v2-1-password-based-cryptography-standard-wp.pdf.
[56] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. 2007. New Message Difference for MD4. In *FSE 2007 (LNCS)*, Alex Biryukov (Ed.), Vol. 4593. Springer, Heidelberg, 329–348.
[57] SHA3 2015. Secure Hash Standard. National Institute of Standards and Technology, NIST FIPS PUB 180-4, U.S. Department of Commerce. (Aug. 2015).
[58] Hovav Shacham and Brent Waters. 2008. Compact proofs of retrievability.. In *Asiacrypt*, Vol. 5350. Springer, 90–107.
[59] Sailesh Simhadri, James Steel, and Benjamin Fuller. 2017. Reusable Authentication from the Iris. Cryptology ePrint Archive, Report 2017/1177. (2017). https://eprint.iacr.org/2017/1177.
[60] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190. (2017). http://eprint.iacr.org/2017/190.
[61] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. 2007. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities, See [49], 1–22.
[62] Laurent Sustek. 2011. Hardware security module. In *Encyclopedia of Cryptography and Security*. Springer, 535–538.
[63] TrueCrypt Foundation. 2014. TrueCrypt. (2014). http://truecrypt.sourceforge.net.
[64] Meltem Sönmez Turan, Elaine B. Barker, William E. Burr, and Lidong Chen. 2010. *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications.* Technical Report. Gaithersburg, MD, United States.
[65] VeraCrypt. 2018. VeraCrypt. (2018). https://www.veracrypt.fr/en/Home.html.
[66] Andrea Visconti, Simone Bossi, Hany Ragab, and Alexandro Calò. 2015. On the Weaknesses of PBKDF2. In *CANS 15 (LNCS)*, Michael Reiter and David Naccache (Eds.). Springer, Heidelberg, 119–126. https://doi.org/10.1007/978-3-319-26823-1_9
[67] Friedrich Wiemer and Ralf Zimmermann. 2014. High-speed implementation of bcrypt password search using special-purpose hardware. In *2014 International Conference on ReConFigurable Computing and FPGAs*. IEEE, 1–6. https://doi.org/10.1109/ReConFig.2014.7032529

A PROOF OF THEOREM 4.2

In this appendix, we demonstrate that Bog is IRO if at least one of its consistuent hash functions is IRO. We first describe the operation of a simulator $\mathcal{S}$ and then we argue that this simulator demonstrates indifferentiability from a random oracle (IRO).

A.1 Simulator design

The simulator $\mathcal{S}$ stores its state within a few data structures. First, $\mathcal{S}$ stores a table from all input queries x_i to their corresponding outputs y_i. Second, $\mathcal{S}$ stores a forest $\mathcal{T}$ (i.e., a set of disjoint trees); it is initialized to be empty and its purpose is to track how different instances of $\mathcal{H}$ are related as subroutines within a single invocation of Bog. Third, $\mathcal{S}$ stores a set C containing the capacity portion of every output y_i; it is an invariant of the construction of $\mathcal{S}$ that they will all be unique, so a set is an appropriate data structure.

When a new query x is made, $\mathcal{S}$ extracts the init_flag and capacity from x. Then, $\mathcal{S}$ executes exactly one of the following three algorithms to prepare its response and update its internal data structures.

Consistent response: Use if x is identical to a prior query x_i. In this case, $\mathcal{S}$ simply returns y_i.

Tree instantiation: Use if init_flag = 1. In this case, $\mathcal{S}$ desires to emulate an execution of $\mathcal{H}(x)$ at the initial step of a new instance of Bog. $\mathcal{S}$ does the following:

- Parse x as (1, Pass, salt, out_len, plugins).
- Set plugins equal to the empty list [].
- Execute a 0-plugin instance of RObog. Parse the first block of the response key as $(\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}})$.
- Sample capacity′ uniformly among strings not in C, and then add capacity′ to C.
- Set $y \triangleq (\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}}, \text{capacity}')$.
- Create a new tree $\mathcal{T}_x$ in the forest whose root (and only) node contains the mapping $x \mapsto y$.
- Return the response y.

Tree extension: Use if init_flag = 0 and capacity equals the capacity bits within some previous output y^*. In this case, $\mathcal{S}$ desires to emulate an execution of $\mathcal{H}(x)$ within the middle of an instance of Bog.

- Fetch the tree $\mathcal{T}_{x^*}$ whose node(s) contain capacity.
- Retrieve all nodes on the path from root x^* to the node containing y^*. We denote this list as $x^1 \mapsto y^1$, $x^2 \mapsto y^2$, ..., $x^m \mapsto y^m$, where $x^* = x^1$ and $y^* = y^m$.
- Compute the differentials $\Delta^j = \mathsf{a}^j_{\mathsf{out}} \oplus \mathsf{a}^{j+1}_{\mathsf{in}}$ for all $j \in [m]$, where $\mathsf{a}^j_{\mathsf{out}}$ is contained in y^j and $\mathsf{a}^{j+1}_{\mathsf{in}}$ is contained in x^{j+1}. For completeness, here we set x^{m+1} equal to the current input x.
- For all $j \in [m]$, create the constant plugin p_j that outputs Δ^j independent of its input p_{in} and r.
- Create the list plugins by inserting the p_j functions so that they execute in order. That is: for every j, read the sched bits of y^j and insert p_j in the location that Bog's scheduler would choose when given sched.
- Parse x^* as (1, Pass, salt, out_len, plugins)
- Execute RObog on input (Pass, salt, out_len, plugins). Parse the first block of the response key as $(\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}})$.
- Sample capacity′ uniformly among strings not in C, and then add capacity′ to C.
- Set $y \triangleq (\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}}, \text{capacity}')$.
- From the node $x^m \mapsto y^m$ within the tree $\mathcal{T}_{x^*}$, add a new child node $x \mapsto y$.
- Return the response y.

Random response: Use if init_flag = 0 and capacity is not contained in any node of any tree of $\mathcal{T}$. In this case, $\mathcal{S}$ interprets this invocation of $\mathcal{H}$ as independent of any calls that relate to Bog.

- Sample $(\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}})$ uniformly at random.
- Sample capacity′ $\notin$ C uniformly at random.
- Add capacity′ to C.
- Return $y = (\text{sched}, \mathsf{p}_{\mathsf{in}}, \mathsf{a}_{\mathsf{out}}, \text{capacity}')$.

A.2 IRO Analysis

In this section, we present a series of lemmas that collectively demonstrate the performance and security properties necessary to prove Theorem 4.2. We begin with a counting argument about $\mathcal{S}$'s performance.

Lemma A.1. *$\mathcal{S}$ makes at most q oracle queries to RObog, and $\mathcal{S}$ runs in time $O(q^2)$.*

Proof. For each query processed, $\mathcal{S}$ makes exactly 1 oracle call to RObog in the "tree instantiation" and "tree extension" cases and 0 calls in the other two cases. Furthermore, the most local work performed within $\mathcal{S}$ occurs in the tree extension case, in which $\mathcal{S}$

does work proportional to the depth of some tree $\mathcal{T}x^*$. This depth is at most q. □

Next, we consider the likelihood that the distinguisher $\mathcal{D}$'s queries to the $\mathcal{H}$ or $\mathcal{S}$ oracle always return outputs with distinct capacity values. Let's call this property *capacity-uniqueness*.

Lemma A.2. *If $q \leq 2^c$, then $\mathcal{S}$ is capacity-unique. Also, $\mathcal{H}$ is capacity-unique with probability at least $1 - \frac{q^2}{2 \cdot 2^c}$.*

Proof. We can validate that distinct inputs to $\mathcal{S}$ always return outputs with distinct capacity by inspecting the $\mathcal{S}$ construction. The "consistent response" case is inconsequential to the statement and all other cases sample the output capacity among strings not in C. Furthermore, each invocation of $\mathcal{S}$ adds at most one element to C. It follows that as long as $q \leq 2^c$, then C remains non-saturated and thus the sampling procedure succeeds. Additionally, $\mathcal{H}$ is a truly random function and thus the claim about its capacity-uniqueness is simply a restatement of the birthday bound. □

In the preceding proof, we denoted the capacity set C as *saturated* if $|\mathsf{C}| = 2^c$, or in other words if the responses to $\mathcal{D}$ span all possible values of the capacity bits. This terminology will prove useful later in the proof.

From the lemma, it follows that in the "tree extension" routine with $\mathcal{S}$ that the "previous output y^*" containing the same capacity as the input query must be unique. Ergo, the choice of the "tree whose node(s) contain capacity" in the first step of the algorithm is also unique.

The next lemma provides even greater precision on the nodes that can share capacity.

Lemma A.3. *Consider any node $v : x^\dagger \mapsto y^\dagger$ within $\mathcal{S}$'s forest, and let capacity† denote the capacity portion of $y^\dagger$. For every node $v' : x' \mapsto y'$, it is the case that the capacity bits of x' equal capacity† if and only if node v' is a child of the node v.*

Proof. For the 'if' direction: if v' is a child of node v then it must have been produced by the tree instantiation algorithm applied to x', since that is the only algorithm that appends leaves to trees. By construction, this node would only be added if x' and $y^\dagger$ have the same capacity.

For the 'only if' direction: if the capacity bits of x' equal capacity†, then $\mathcal{S}$ would follow the tree instantiation algorithm, and not any of the other three algorithms, when computing $\mathcal{S}(x')$. □

With these lemmas, we can prove our main statement about the correctness of $\mathcal{S}$.

Lemma A.4. *The relationship between $\mathcal{D}$'s queries to $\mathcal{S}$ and RObog is* consistent *with the expected behavior of the bog construction, unless $\mathcal{S}$ saturates all possible choices of capacity bits.*

Proof. The real Bog construction daisy-chains together invocations of $\mathcal{H}$ where the output of one invocation and the input of the next invocation have the same capacity, beginning with an input of a special format (1, Pass, salt, out_len, plugins).

Ergo, Lemma A.3 implies that the only way that the distinguisher $\mathcal{D}$ can check sponge-consistency is to run RObog on an input that was also fed into $\mathcal{S}$'s tree instantiation routine (i.e., the root x of some tree $\mathcal{T}_x$). Furthermore, $\mathcal{D}$ can only check sponge-consistency for rooted paths within $\mathcal{T}_x$.

If $\mathcal{D}$ runs the real Bog on an empty list of plugins, then Bog construction simply runs one invocation of its round transformation $\mathcal{H}$, outputs its rate bits, and halts. Analogously, in our construction, $\mathcal{S}$'s tree instantiation step queries RObog for the rate bits and thus provides a consistent response.

If $\mathcal{D}$ runs the real Bog on a non-empty list of plugins, then Bog iteratively invokes $\mathcal{H}$ where the outputs of one invocation and the inputs of the next invocation have the same capacity and satisfy the relation $\mathsf{a}^j_{\mathsf{out}} \oplus \mathsf{p}^j_{\mathsf{out}} = \mathsf{a}^{j+1}_{\mathsf{in}}$. Analogously, the tree extension step of $\mathcal{S}$ designs plugins that ensure the same relation between one round's outputs and the next round's inputs. Note that the actual plugins used by $\mathcal{S}$ are quite different than those used within the real Bog, but this is acceptable since the accuracy of the construction depends only on the output of each plugin, not how it uses its resources to compute $\mathsf{p}_{\mathsf{out}}$.

Finally, the tree instantiation, tree extension, and random response algorithms above all assume that C is not full, and thus they will fail if so. $\mathcal{S}$ requires the capacity-uniqueness property to ensure that the outputs of the "random response" method never need to be consistent with any invocation of RObog. □

To complete the proof of Theorem 4.2, all that remains is to bound the probability of C being saturated (i.e., $|\mathsf{C}| = 2^c$).

Lemma A.5. *The probability that the simulator $\mathcal{S}$ saturates in response to a sequence of $q \ll 2^c$ queries is at most $\epsilon = \frac{q(q+1)}{2^{c+1}}$.*

The proof of this final lemma follows an identical variational distance argument as the one Bertoni et al. use in [17, Lemma 4]. We omit a presentation of the argument here as it is purely combinatorial in nature and does not provide any new intuition about Bog.

B PROOF OF THEOREM 4.5

In this section, we prove Theorem 4.5 that demonstrates the unpredictability of Bog's output.

Let $\mathcal{A}$ be an attacker as defined above. We design a new attacker $\mathcal{B}$ (with access to the same oracles) that can attack either the collision resistance of $\mathcal{H}$ or the unpredictability of a plugin.

$\mathcal{B}$ acts as $\mathcal{A}$'s challenger. After the setup process is complete, $\mathcal{B}$ inserts her challenge unpredictable function into the location of one of $\mathcal{A}$'s m non-bypassed plugins p_{i^*}. Then, $\mathcal{B}$ responds to $\mathcal{A}$'s query requests honestly, recording (but not altering!) all inputs and outputs in the process. Furthermore, $\mathcal{B}$ aborts if $\mathcal{A}$ queries the challenge plugin p_{i^*} at least |Dict| times or if $\mathcal{A}$ finds a collision in $\mathcal{H}$.

By assumption, $\mathcal{B}$ completes the indifferentiability game without aborting with probability at least $1/m - \epsilon$. In this case, $\mathcal{A}$ provides $\mathcal{B}$ with a password-key dictionary (Dict, Keys).

We make the following claim: if $\mathcal{A}$ succeeds in the IRO game, then $\mathcal{B}$ has the ability to predict (with probability 1) an input-output pair to p_{i^*} without ever having queried for it. In the remainder of the proof, we show the value of this claim and then we prove it.

First, the claim implies that $\mathcal{B}$'s probability of success is precisely $\mathsf{Adv}_{\mathcal{A}}/|\mathsf{Dict}|$. This probability is upper-bounded by ϵ by the definition of unpredictability, from which the claim follows.

Second, we prove the claim that if $\mathcal{A}$'s dictionary (Dict, Keys) is correct then $\mathcal{B}$ can predict an input-output pair (x^*, y^*) such that $p_{i^*}(x^*) = y^*$ without ever having explicitly queried for it. $\mathcal{B}$ essentially follows a 'meet in the middle' algorithm:

- For all elements $w \in$ Dict, $\mathcal{B}$ calculates Bog in the forward direction, stopping just before the challenge plugin p_{i^*}. We emphasize that $\mathcal{B}$ need not make any oracle queries to calculate Bog; instead $\mathcal{B}$ simply reviews her already-recorded answers to $\mathcal{H}$ and the plugins $p_1, \ldots, p_{i^*-1}$ when making these calculations, choosing any outputs uniformly at random if they weren't queried before. Let X denote the list of all such intermediate state.
- For all keys $z \in$ Keys, $\mathcal{B}$ calculates Bog in the *right-to-left* direction from the output stage until just after p_{i^*}. Even though $\mathcal{B}$ lacks access to inverse oracles (and indeed an inverse to $\mathcal{H}$ or the plugins need not even exist), $\mathcal{B}$ can proceed in the right-to-left direction precisely because she doesn't need to make oracle invocations and instead can simply check the already-recorded answers from $\mathcal{A}$'s queries. That is: $\mathcal{B}$ finds the preimage of z in $\mathcal{H}$, and then the preimage of that value in p_m, and so on. Let Y denote the list of all such intermediate state.

If $\mathcal{A}$'s response is correct, then p_{i^*} maps each element in the list X to the list Y in order; that is, $\text{map}(p_{i^*}, X) = Y$. Furthermore, we asserted above that $\mathcal{A}$ made fewer than $|\text{Dict}| = |X|$ queries and that all queries have distinct responses, so there must exist some $x_j \in X$ that $\mathcal{A}$ (and thus also $\mathcal{B}$) never queried. Finally, $\mathcal{B}$ outputs the input-output tuple (x_j, y_j) as her prediction.