On the Security of Internet-scale Services

by

Adam C Everspaugh

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 31 July 2017

The dissertation is approved by the following members of the Final Oral Committee:
Nigel Boston, Professor, Mathematics
Somesh Jha, Professor, Computer Sciences
Barton Miller, Professor, Computer Sciences
Thomas Ristenpart, Associate Professor, Computer Science
Michael Swift, Professor, Computer Sciences

This dissertation is dedicated to my wonderful family: Kirsten, Zoe, and Finn; and to my grandmother, Dorothy Williams, who believed a doctorate to be a noble pursuit.

ACKNOWLEDGMENTS

It is my privilege to thank my advisors, Profs. Thomas Ristenpart and Michael Swift, for their insight and guidance. My thanks to Ari Juels for his expertise and encouragement. Thanks also to the members of my doctoral committee for your effort and help in improving this thesis. Finally, I'm honored to thank my fellow student co-authors for their insight, effort, and camaraderie.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

We examine three widely-deployed security-critical services (random number generation, user authentication, and protection of data-at-rest) in the context of modern internet-scale services. Internet-scale services are deployed in large data centers or multitenant cloud computing environments; these services are distributed across many machines and may process billions of requests per day while providing high availability. In each case, the security services we examine were designed for a distinct (smaller scale) context and fail to provide the required security when applied to internet-scale services. For each case, we develop and evaluate new, principled designs to restore the required level of security in this large-scale computing environment.

1 INTRODUCTION

Internet-scale services are hosted in a markedly different computing environment than traditional applications from the era pre-dating the 1990s. These traditional computing applications were often deployed and operated on a single machine. Internet-scale services are housed in massive warehouses dubbed data centers [11]. These warehouses contain thousands of individual servers connected by high speed local area networks. Services are distributed across tens or hundreds of servers and may have software components and data replicated (or distributed) in multiple data centers in distinct regions across continents or around the globe.

Some of these data centers are multi-tenant environments: a single company owns and operates the facility and servers and sells access to these servers to third-party customers. In many cases, processes from different tenants are hosted on the same physical servers, using containers or virtual machines to isolate processes between tenants. Third-party hosting is so convenient and cost-effective that many of today's internet-scale services are hosted in third-party data centers (sometimes called *public cloud providers*). Even organizations with substantial, privately-owned computing resources like Dropbox (file storage) and Netflix (media streaming) use third-party services to provide additional, on-demand resources or to gain additional availability for their services [51, 28].

Where individual servers and applications of a previous era might have dozens to hundreds of users, internet-scale services like web search and e-commerce have millions to billions of users. These services seek to be always-available, provide exceptional performance, and handle billions of requests per day. Further, these services must do so even as developers continuously deploy updates, software fixes, and new features.

Unlike computing in a previous era, many of the developers and service operators may never see the hardware that they are using. These servers operate predominately unattended.

Despite the radically different environment for internet-scale services, many of the security mechanisms that were used in the previous era have been brought forward and applied to internet-scale services. In many cases, these security mechanisms are applied without adaption or consideration for the new environment or the scale at which these services must operate. In this work, we investigate whether security mechanisms that are currently in-use provide adequate security; and where they fail, we design, develop, and evaluate new mechanisms that are adapted to the data center environment of internet-scale services.

Over the following three chapters this work examines three critical security services that are widely deployed: the operating system's random number service; password-based authentication; and encryption of data-at-rest.

Random numbers. In Chapter 2, we examine the operating system's random number generator (RNG). We perform detailed empirical analysis of the RNG during operating system boot, then collect and analyze inputs and outputs to the RNG in four environments, both native and virtual machine environments. Analysis of inputs demonstrates sufficient entropy to securely seed the RNG – this stands counter to prior speculation in the academic literature that virtual machines and data center environments would be entropy-poor environments.

We also examine RNG outputs when virtual machine snapshots are used and discover that all major operating systems (Linux, FreeBSD, and Windows) demonstrate catastrophic RNG output failures when snapshots are used more than once (producing repeated outputs following snapshot resumption). To fix this failing, we designed and implemented a novel RNG that performs properly when snapshots are reused and also features an improved design that lends itself to inspection and formal security analysis.

Password-based authentication. There have been a sufficient number of password database breaches publicly reported that we skip analysis of the security of state-of-the-art for password-based authentication. Instead, in Chapter 3 we design, implement, and evaluate a novel replacement for the existing techniques. The replacement is itself designed as a cloud service, which we call *Pythia*, and is based on a new cryptographic protocol that we call a verifiable, partially-oblivious pseudorandom function (PRF). The Pythia PRF protocol splits password checking (securely) between two servers: one that stores hardened passwords, and another that stores a secret key. In addition to provided password verification, the protocol supports secret key rotation that can be used to recover security when either the hardened password database or secret key are compromised.

Encryption of data-at-rest. Finally, in Chapter 4, we examine updatable encryption: a data owner places encrypted information on a remote, cloud storage system and holds the secret key herself. Periodically, the data owner wishes to rotate the secret key, but requires an efficient (and secure) technique to do so without sending the secret key to the storage provider and without being required to download, re-encrypt, and re-upload the entire corpus of encrypted information. We establish formal security definitions for this setting and examine the technique, *AE-hybrid*, that is widely-used in practice. Formal analysis reveals several weaknesses in AE-hybrid and we demonstrate two novel techniques for improving security: the first is a small change to AE-hybrid that recovers basic security and retains similar performance, and the second is a new updatable encryption scheme that gives very strong security guarantees at the cost of being computationally expensive.

2 SECURE RANDOM NUMBERS

Operating systems provide a (secure) random number service that seeks to collect a large amount of unpredictable state used to seed a pseudorandom number generator (PRNG). The PRNG can then generate an arbitrary number of outputs that are uniformly distributed. This random number service is typically called an RNG: random number generator. Modern designs are modeled properly as a PRNG-with-input, see [39]. When the seed is sufficiently large and unpredictable, and the PRNG output leaks no information about the seed, the RNG is considered secure: an attacker learning previous outputs has no advantage in guessing future outputs of the RNG.

Folklore concerns. There exists significant folklore from [57, 91, 58] that system RNGs, such as the one in the Linux operating system, may provide poor security in virtualized settings like Amazon's EC2. [93] hypothesized that the Linux RNG, when run within the Xen virtualization platform on EC2, outputs predictable values very late in the boot process. [47] first hypothesized vulnerabilities arising from the reuse of random numbers when using virtual machine snapshots. [85] were the first to show evidence of these and called them *reset vulnerabilities*. They demonstrated that user-level cryptographic processes can suffer catastrophic loss of security when run in a VM that is resumed multiple times from the same snapshot. Left as an open question in that work is whether reset vulnerabilities also affect system RNGs. Finally, common folklore states that software entropy sources are inherently worse on virtualized platforms due to frequent lack of keyboard and mouse, interrupt coalescing by VM managers, and more. Despite this speculation, there was no work (prior to this one) measuring and evaluating the security of Linux or other system RNGs in virtualized environments.

OS	RNG	Reset Security	Boot Security	Tracking Security
Linux	GRI	No	No	No
	`/dev/(u)random`	No	No	Yes
FreeBSD	`/dev/random`	No	?	Yes
Windows	rand_s	No	?	?
	CryptGenRandom	No	?	?
	RngCryptoServicesProvider	No	?	?
Linux	Whirlwind	Yes	Yes	Yes

Table 2.1: Security comparison of system RNGs. A question mark (?) indicates "Unknown". Reset security refers to safety upon VM snapshot resumption, boot security means sufficient entropy is generated prior to first use, and tracking security is forward- and backward-security in the face of compromise.

The work in this chapter fills this gap. It analyzes Linux and its two RNGs: the kernel-only RNG (used for stack canaries and address-space layout randomization) as well as the more well-known RNG underlying the `/dev/urandom` and `/dev/random` devices. Through careful instrumentation, we capture all inputs to these RNGs in a variety of virtualized settings, including on local Xen and VMware platforms as well as on Amazon EC2 instances. We then perform various analyses to estimate the security of the RNGs. Our work reveals that:

- We demonstrate in Section 2.3 that, contrary to folklore, software entropy sources, in particular cycle counters from interrupts, provide significant uncertainty to an adversary during normal operating system startup.
- However, during system startup the first use of the kernel-only RNG as well as the first use of `/dev/urandom` are vulnerable. There exists a boot-time entropy hole, where insufficient entropy has been collected before use of the RNGs. Later outputs of the RNG, however, appear intractable to predict, suggesting the concerns of [93] are unwarranted. This is discussed in Section 2.3.

- Finally, the `/dev/urandom` RNG suffers from catastrophic snapshot reset vulnerabilities, which answers the open question of [85]. We demonstrate this issue in Section 2.4 and show that, for example, resets can lead to exposure of secret keys generated after snapshot resumption.

Our results are qualitatively the same across the different VM management environments, though note that EC2 does not currently support snapshots and therefore does not (currently) suffer from reset vulnerabilities.

We also perform limited experiments with FreeBSD and Windows, and show that reset vulnerabilities affect FreeBSD's `/dev/random` and Microsoft Windows rand_s as well (see Section 2.4). This suggests that problems with virtualized deployments are not confined to the Linux RNGs.

We offer a new RNG design and implementation for Linux, called Whirlwind in Section 2.5. It directly addresses the deficiencies that we uncovered, as well as other long-known problems with the Linux RNG. Here we are motivated by, and build off of, a long line of prior work: pointing out the troubling complexity of the `/dev/random` and `/dev/urandom` RNG by [52], [62], and [39]; showing theoretical weaknesses in the entropy accumulation process, see [39]; designing multi-pool RNGs without explicit entropy counters by [56] and [69]; and showcasing the utility of instruction and operation timing to quickly build entropy, see [74], [1], and [76].

Whirlwind combines a number of previously suggested techniques in a new way, along with several new techniques. It serves as a drop-in replacement for both of the Linux RNGs, and provides better security (see Table 2.1). In addition to security, the design focuses on simplicity, performance, theoretical soundness, and virtualization safety (though it will perform well for native settings as well). At its core is a new cryptographic hashing mode, inspired by but different from the recent construction of [39], plus: a simple two-pool system, simpler interface, streamlined mostly-CPU-lock-free entropy addition, a method for bootstrapping entropy during boot and VM resumption, and direct compatibility with hypervisor-provided randomness. We emphasize that the security of Whirlwind never relies on any one source in particular, and instead uses multiple inputs sources to ensure the highest possible uncertainty even in the face of some entropy sources being compromised.

In terms of performance, Whirlwind matches that of the current Linux RNG and in some cases performs better. We show experimentally that it suffers from none of the problems for virtualized settings that render the current Linux RNG vulnerable.

Finally, in Section 2.6 we explore hypervisor-based countermeasures for legacy guest VMs with the old RNG. In particular, we investigate whether the hypervisor can defend against reset vulnerabilities by injecting entropy into the guest RNG via (artificially generated) interrupts during resumption. We show that the host OS can force Linux `/dev/random` to refresh itself, but that it takes at least a few seconds and requires a large number of interrupts. Such limitations suggest that legacy-compatible approaches are unsatisfying in the long term, and we instead suggest moving to a new RNG such as Whirlwind.

2.1 Background

The Linux RNGs

The Linux kernel provides three RNG interfaces which are designed to provide cryptographically strong random values: `/dev/random`, `/dev/urandom`, and get_random_int (GRI).

The `/dev/(u)random` RNG. We call the primary RNG for the Linux operating system the `/dev/(u)random` RNG. Linux exposes two pseudo-devices via the file system that are the interfaces to `/dev/(u)random`. The first in-

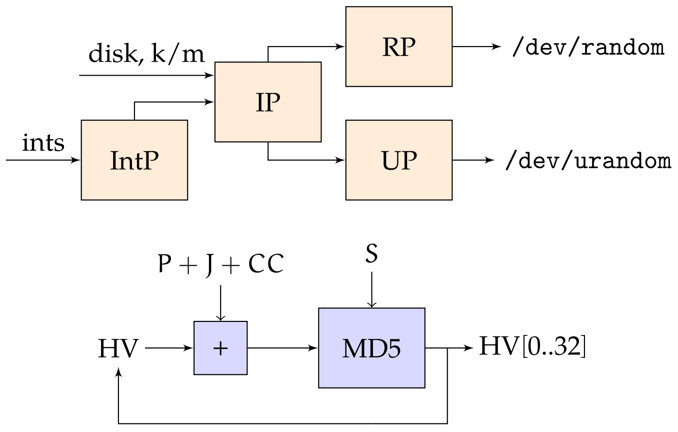


Figure 2.1: The Linux RNGs. **(Top)** Data flow through the `/dev/(u)random` RNG. **(Bottom)** The kernel-only RNG, get_random_int (GRI).

terface, `/dev/random`, may block until enough entropy is available, while the second, `/dev/urandom`, is non-blocking. On the systems we examined, applications and the Linux operating system itself use exclusively `/dev/urandom` and never read from `/dev/random`. The RNG consists of (1) entropy gathering mechanisms that produce *descriptions* of system events; (2) several entropy *pools* to which these descriptions are mixed with a cryptographically weak generalized feedback shift register; (3) logic for how and when entropy flows between pools (described below); and (4) APIs for consumers to query to obtain randomness. To retrieve random numbers, an application opens one of the device files, performs a read, and (presumed-to-be) random bytes are returned. Additionally, an application may write to either device, in which case the `/dev/(u)random` RNG mixes the contents of the write buffer into both secondary entropy pools (also described below) but does not update any entropy estimates. For example, during boot a file containing output from `/dev/urandom` during the preceding shutdown is written back into `/dev/(u)random`. Read and write requests are always made in units of bytes. The `/dev/urandom` RNG also has a kernel-only interface get_random_bytes() that does not use the pseudo-device but is functionally identical to `/dev/urandom`.

An entropy pool is a fixed-size buffer in kernel memory along with associated state variables. These variables include the current mixing location for new inputs and an entropy count measured in bits. There are four pools as shown on Figure 2.1. Below, we omit details regarding the non-cryptographic mixing functions. Detailed descriptions appear in [39, 62].

Interrupt pool (IntP): The kernel IRQ handler adds a description of each interrupt to a 128-bit interrupt pool (called a "fast pool" in the source code). There is one IntP per CPU to eliminate contention. Each interrupt delivery takes a description (cycle counter xor'd with kernel timer, IRQ number, and the instruction pointer at the time the interrupt is received) and mixes it into the pool using a cryptographically weak function. The entire contents of each IntP are mixed into the input pool IP using another (more complex generalized feedback register) mixing function every 64 interrupts or if a second has passed since the last mixing into IP. At the same time, the input pool entropy count denoted $\mathsf{IP.ec}$ is incremented (credited) by one (bit), which represents a conservative estimate.

Input pool (IP): The 4096-bit input pool has the interrupt pool mixed into it as just mentioned, and as well has device-specific event descriptions (kernel timer value, cycle counter, device-specific information) of keyboard, mouse, and disk events mixed in using the cryptographically weak mixing function. We will only consider settings with no keyboard or mouse (e.g., servers), and so only disk events are relevant. (Network interrupts go to IntP.)

Non-blocking pool (UP): A 1024-bit pool is used for the non-blocking `/dev/urandom` interface. Upon a request for $8n$ bits of randomness, let

$$\alpha_u = \min(\min(\max(n, 8), 128), \lfloor \mathsf{IP.ec}/8 \rfloor - 16).$$

If $\mathsf{UP.ec} < 8n$ and $8 \leqslant \alpha_u$ the RNG transfers data from the input pool IP to UP. Put another way, a transfer occurs only if $\mathsf{UP.ec} < 8n$ and $\mathsf{IP.ec} \geqslant 192$. If a transfer is needed, the RNG extracts α_u bytes from IP, mixes the result into UP, decrements $\mathsf{IP.ec}$ by $8\alpha_u$, and increments $\mathsf{UP.ec}$ by $8\alpha_u$. If a transfer is not needed or not possible (by the restrictions above), then UP is left alone. In the end, the RNG decrements $\mathsf{UP.ec}$ by $8n$, extracts $8n$ bits from UP, and return those bits to the calling process.

Blocking pool (RP): A 1024-bit pool is used for the blocking `/dev/random` interface. Upon a request for $8n$ bits of randomness, let

$$\alpha_r = \min(\min(\max(n, 8), 128), \lfloor \mathsf{IP.ec}/8 \rfloor).$$

If $\mathsf{RP.ec} \geqslant 8n$ then it immediately extracts $8n$ bits from RP, decrements $\mathsf{RP.ec}$ by $8n$, and returns the extracted bits. Otherwise it checks if $\alpha_r \geqslant 8$ and, if so, transfers α_r bytes from IP to RP, incrementing and decrementing entropy counters appropriately by $8\alpha_r$. It then immediately extracts $\lfloor \mathsf{RP.ec}/8 \rfloor$ bytes from RP, decrements $\mathsf{RP.ec}$ appropriately, and returns the extracted bits to the calling process. If on the other hand $\alpha_r < 8$, then it blocks until $\alpha_r \geqslant 8$.

Table 2.2 summarizes the conditions required for transferring data from one pool to the next. The design of `/dev/(u)random` intimately relies on ad-hoc entropy estimates, which may be poor. We will also see, looking ahead, that the entropy counters cause trouble due to their use in deciding when to add entropy to the secondary pools. For example, we observe that there exists a simple *entropy starvation* attack against `/dev/urandom` by a malicious user process that continuously consumes from `/dev/random` (e.g., using the command `dd if=/dev/random`). In this case, reads from `/dev/urandom` will never trigger a transfer from IP.

Output extraction: We give an overview of the cryptographic extraction routine for IP, UP, and RP. To begin, the contents of the pool are first hashed with SHA-1 and then mixed into the same pool using the non-

Transfer	When	Condition
IntP $\to$ IP	Interrupt arrival	64 interrupts or 1 second
IP $\to$ UP	n bytes requested from `/dev/urandom`	$\mathsf{UP.ec} < 8n$ $\mathsf{IP.ec} \geqslant 192$
IP $\to$ RP	n bytes requested from `/dev/random`	$\mathsf{RP.ec} \leqslant 8n$ $\mathsf{IP.ec} \geqslant 64$

Table 2.2: Conditions for transfers between entropy pools.

cryptographic mixing function. Then 64 bytes of the pool are hashed with a modified SHA-1 algorithm that uses the first 5 bytes of the previous hash as the IV. This 20-byte digest d is then reduced to a 10-byte output block $d' = d[0..3] \oplus d[12..15] \| d[4..7] \oplus d[16..19] \| d[8..9] \oplus d[10..11]$ where $\|$ is concatenation. This entire routine is repeated until the ouput is at least as long as the requested number of bytes. The final output block is truncated as needed. The entropy count for the pool is decremented by the number of output bits that were generated (excluding any truncated bits). When extracting from IP, the output value is mixed into either UP or RP (whichever generated the request). When extracting from UP or RP the output value becomes the output of `/dev/urandom` or `/dev/random` respectively.

get_random_int: the kernel-only RNG. GRI is a simple RNG, shown in Figure 2.1, that provides 32-bit values exclusively to callers inside the kernel. GRI is primarily used for Address Space Layout Randomization (ASLR) and StackProtector "canary" values used to thwart stack-smashing attacks. The GRI RNG is designed to be very fast and does not consume entropy from the pools in `/dev/(u)random`.

The GRI RNG uses two values stored in kernel memory: a per-CPU 512-bit hash value HV and a global 512-bit secret value S, which is initially set to all zeros. During the `late_init` phase of boot, the kernel sets the secret value S to 512-bits obtained from `/dev/urandom`.

Each time it is called, GRI adds the process ID (PID) P of the current process, the current kernel timer value J (called *jiffies*), and the lower-32 bits of the timestamp cycle counter CC into the first 32-bit word of the hash value HV, and then sets HV to the MD5 hash of HV and the secret value S. That is, it computes $\mathsf{HV} = \mathsf{H}(\ (\mathsf{HV}[1\mathinner{..}32] + \mathsf{P} + \mathsf{J} + \mathsf{CC}) \| \mathsf{HV}[33\mathinner{..}512] \| \mathsf{S})$ where "$+$" is integer addition modulo 2^{32}, and $\mathsf{H}(\cdot)$ is the MD5 hash. The first 32 bits of HV are returned to the caller, and the new HV value becomes the stored hash value for the next call.

Use of hardware RNGs. If available, the `/dev/(u)random` RNG uses architecture-specific hardware RNGs during initialization and output generation. During boot, `/dev/(u)random` reads enough bytes from the hardware RNG to fill each pool and uses the weak mixing function to mix in these values. This is done for the input, nonblocking, and blocking pools, but not for the interrupt pool. During output generation, `/dev/(u)random` XORs 10-bytes from the hardware RNG into each 10-byte block of output that is produced. GRI returns a 32-bit value from the hardware RNG in place of the software implementation described above.

Virtualization

In this work, we focus on the efficacy of the Linux RNGs when operating in virtualized environments without the aid of a hardware RNG. In a virtualized environment, one or more guest virtual machines (VMs) run on a single physical host, and the hypervisor mediates access to some hardware components (e.g., the network interface, disks, etc.). There is a management component for starting, stopping, and configuring virtual machines. In Xen, this is called Dom0, while in hosted virtual machines (e.g., VMware Workstation) this is the host operating system.

A VM can be started in one of three ways. First, it can execute like a normal physical system by booting from a virtual disk. As it executes, it can update the state on the disk, and its next boot reflects those changes. Second, a VM can be repeatedly executed from a fixed *image*, which is a file that contains the persistent state of the guest OS. In this case, changes made to the OS state are discarded when the VM shuts down, so the OS always boots from the same state. This is the default case, for example, in infrastructure-as-a-service cloud computing systems including Amazon EC2. Third, a VM can start from a *snapshot*, which is a file that contains the entire state of a running VM at some point in its execution. This includes not only the file system but also memory contents and CPU registers. Both Xen and VMware support pausing a running VM at an arbitrary point in its execution and generating a snapshot. The VM can be resumed from that snapshot, which means it will continue executing at the next instruction after being paused. If a VM continues running after the snapshot, restarting from a snapshot effectively rolls back execution to the time when the snapshot was taken.

It has long been the subject of folklore that RNGs, and in particular, `/dev/(u)random`, may not perform as well when run within a VM, see: [93, 57, 91], and [58]. First, hypervisors often *coalesce* interrupts into batches before forwarding them to a given guest domain to improve performance. Second, memory pages are typically zeroed (set to all zeroes to erase any "dirty" data) by the hypervisor when new physical memory pages are allocated to a guest VM. Zeroing memory pages is required to ensure that dirty memory does not leak information between different guests on the same host machine. Third, several system events used for entropy by `/dev/(u)random` are not relevant in popular uses of virtualization, in particular keyboard and mouse events do not occur in virtualized servers.

RNG Threat Models

The Linux RNGs are used by a variety of security-critical applications, including cryptographic algorithms and for system security mechanisms.

Should RNG values be predictable to an adversary or the same (unknown) value repeatedly used, the RNG-using applications become vulnerable to attack. As just a few examples, `/dev/urandom` is used to seed initial TCP/IP sequence numbers and by cryptographic libraries such as OpenSSL to generate secret keys, while GRI is used as mentioned above for ASLR and stack canaries.

RNGs are therefore designed to face a variety of threats from attackers both off-system and (unprivileged) local attackers. We assume that the attacker always knows the software and hardware stack in use (i.e., kernel versions, distribution, and underlying hypervisor). The threats to RNG systems are:

- *State predictability*: Should the entropy sources used by the RNG not be sufficiently unpredictable from the point of view of the attacker, then the RNG state (and so its output) may be predictable. For example, [50] show that a low-granularity time stamp (e.g., seconds since the epoch) is a bad entropy source because it is easily guessed.

- *State compromise*: The attacker gets access to the internal state of the RNG at some point in time and uses it to learn future states or prior states (forward-tracking and back-tracking attacks respectively). Forward-tracking attacks may use RNG outputs somehow obtained by the attacker as *checkpoints*, which can help narrow a search by allowing the attacker to check if guessed internal states of the RNG are correct. VM snapshots available to an attacker, for example, represent a state compromise.

- *State reuse*: With full-memory VM snapshots, the same RNG state may be reused multiple times and produce identical RNG outputs. Since the security of a random number is its unpredictability, this can eliminate the security of the operation using a repeated RNG output.

- *Denial-of-service*: One process attempts to block another process from using the RNG properly.

Our focus will be on the design of the RNGs, and so we will not attempt to exploit cryptanalytic weaknesses in the underlying cryptographic primitives MD5 and SHA-1.

2.2 Measurement Study Overview

In the following sections we report on measurements in order to answer several questions about the security of the Linux RNGs when used on virtual platforms. In particular:

- When booting from a VM image, how quickly is the RNG state rendered unpredictable? (Section 2.3)
- Does VM snapshot reuse lead to reset vulnerabilities? (Section 2.4)

Along the way we build a methodology for estimating uncertainty about the RNG state, and, as a result, assessing the suitability of various sources of entropy. Of course, one cannot hope to fully characterize software entropy sources in complex, modern systems, and instead we will use empirical estimates as also done by prior RNG analyses, see [74, 49]. When estimating the complexity of attacking an RNG, we will be conservative whenever possible (letting the adversary know more than realism would dictate). Where vulnerabilities appear to arise, however, we will evidence the issues with real attacks.

To accomplish this, we perform detailed measurements of the Linux RNGs when rebooting a virtual machine and when resuming from a snapshot. We produced an instrumented version of the Linux kernel v3.2.35, which we refer to as the *instrumented kernel*. The instrumentation records all inputs submitted to the RNGs, all calls made to the RNGs to produce outputs, changes to the entropy counts for each of `/dev/(u)random`'s pools, and any transfers of bits between entropy pools. To avoid significant overheads, the resulting logs are stored in a static buffer in memory, and are written to disk at the end of an experiment. Our changes are restricted to the file: /drivers/char/random.c.

There were surprisingly non-trivial engineering challenges in instrumenting the RNGs, as the breadth of entropy sources, inherent non-determinism (e.g., event races), and the potential for instrumentation to modify timing (recall that time stamps are used as entropy sources) make instrumentation delicate. For brevity we omit the details. However, we did validate the correctness of our instrumentation by building a user-level simulator of the RNGs. It accepts as input log files as produced by the instrumented kernel, and uses these to step through the evolution of the state of the RNGs. This allowed us to verify that we had correctly accounted for all sources of non-determinism in the RNG system, and, looking ahead, we use this simulator as a tool for mounting attacks against the RNGs. For any computationally tractable attacks, we also verify their efficacy in an unmodified Linux kernel.

We have publicly release open-source versions of the instrumented kernel as well as simulator so others can reproduce our results and/or perform their own analyses in other settings. Links to open-source code for this project can be found on the author's website.

We use the following experimental platforms. For local experiments, we use a 4-core Intel Xeon E5430 2.67 GHz CPU (64-bit ISA) with 13 GB of main memory. We use Ubuntu Linux v12.10 in the *Native setup*, and we use the same OS for host and guest VMs. The *Xen setup* uses Xen v4.2.1, and the single Xen guest (domU) is configured with a single CPU and 1 GB of main memory. The cycle counter is not virtualized on Xen experiments (the default setting). The *VMware setup* uses VMware Workstation 9.0.0 with guest given a single CPU and 2 GB of main memory. On VMware

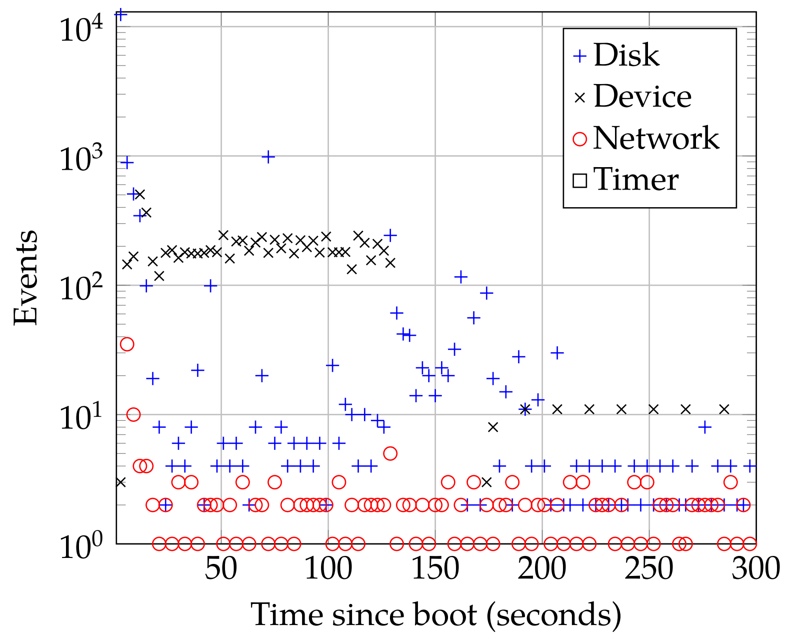


Figure 2.2: The number of inputs to `/dev/(u)random` RNG by type of event during boot on VMware. The y-axis contains number of events (on a logarithmic scale) bucketed in 3-second bins.

the cycle counter *is* virtualized (the default).

Although we performed experiments with Ubuntu, our results should apply when other Linux distributions are used in either the host and/or guest. Finally in our *EC2 setup*, we built an Amazon Machine Image (AMI) with our instrumented kernel running on Ubuntu Linux v12.04 (64-bit ISA). All experiments launched the same AMI on a fresh EBS-backed m1.small instance in the US East region. In our experimental setups, there exist no keyboard or mouse inputs, which is consistent with VM deployments in data centers.

2.3 Boot-time RNG Security

We examine the behavior of the two Linux RNGs (GRI and `/dev/(u)random`) during boot, in particular seeking to understand the extent to which there exist boot-time entropy holes (insufficient entropy collection before the first uses of the RNGs). As mentioned, [93] raised the concern that the Linux RNGs, when running on Amazon EC2, are so entropy-starved that cryptographic key generation towards the end of boot could be compromised. Our results refute this, showing that uncertainty in the RNGs is collected rather rapidly during boot across a variety of settings. We do, however, expose a boot-time entropy hole for the very first uses of both GRI and `/dev/(u)random`. In both cases the result is that stack canaries generated early in the boot process do not provide the targeted uncertainty: they are limited to 27 bits of uncertainty (instead of 64 bits) due to weak RNG outputs.

We perform analyses using the instrumented kernel in the Native, Xen, VMware, and Amazon EC2 setups (described in Section 2.2). We perform 200 boots in each environment, and analyze the resulting log files to assess the security of the RNGs. After boot, the VM is left idle. We break down our discussion by RNG, starting with `/dev/(u)random`.

`/dev/(u)random` boot-time analysis

The left graph in Figure 2.2 displays the quantity and types of inputs to the RNG for a single boot in the VMware setup (the other VMware traces are similar). The x-axis is divided into 100 equal-sized buckets (3 seconds each) and the y-axis represents the number of occurrences of each input to the RNG state observed during a single time bucket (on a logarithmic scale). The majority of RNG inputs during boot are from disk events and other device interrupts while timer events are rare. The other platforms (Native, Xen, and EC2) were qualitatively similar.

Estimating attack complexity. We wish to estimate the security of the *outputs* from `/dev/(u)random`, and so we examine its *inputs* to establish a lower bound on the complexity of predicting the RNG's internal state. Given that we target only lower bounds, we are conservative and assume the attacker has a significant amount of information about inputs and outputs to the RNG. When these conservative estimates show a possible vulnerability, we check for attacks by a more realistic attacker.

To establish a lower bound, we define the following conservative attack model. The attacker is assumed to know the initial state of the RNG (this is trivially true when booting VM images, due to zeroed memory) and the absolute cycle counter at boot time (the exact value is not typically known). To estimate the security of output `i` of `/dev/(u)random`, we assume the attacker has access to *all* preceding RNG outputs and the exact cycle counter for each output generation, including the $\mathtt{i}^{th}$ output. This means we are assessing a kind of checkpointing or tracking attack in which the attacker can utilize knowledge of previous RNG outputs generated by typical requests to reduce her search space.

We will additionally assume that the exact sequence of RNG input types and the values of all event descriptions except the cycle counter are known to the attacker. This makes the cycle counter the only source of unpredictability for the attacker. The reason we do this is that, in fact, the other inputs included in event descriptions such as IRQ appear to provide relatively little entropy. (Of course a real attacker would need to predict these values as well, but again we will be generous to the attacker.)

Input and output events. For a given platform (Xen, VMware, EC2, or native) we analyze traces of inputs and outputs of `/dev/(u)random` collected using our instrumented kernel. We call a single boot of the instrumented kernel a *trial*. Each trial produces a *trace*, a sequence of recorded input and output events of the form:

$$\mathsf{Tr}_k = (y_1, \mathtt{ctr}_1, CC_1), (y_2, \mathtt{ctr}_2, CC_2), (y_3, \mathtt{ctr}_3, CC_3), \ldots$$

where $0 \leqslant k < 200$ is the index for a given trial, and

$$y \in \{\mathsf{keyboard}, \mathsf{mouse}, \mathsf{disk}, \mathsf{IRQ\text{-}0}, \ldots, \mathsf{IRQ\text{-}n}, \mathsf{output}\}$$

is the type of event recorded, $\mathtt{ctr}$ is the counter for events of this type that appear in trace Tr_k, and CC is the 64-bit cycle counter at the time the event is recorded. The output type represents any output from `/dev/(u)random` and the remaining values indicate different sources of input events. In the context of our analysis, we often use the terms trial and trace interchangeably.

Across all 200 traces, we group input events with matching type y and counter $\mathtt{ctr}$ into a vector $\vec{p}$. For example, for $y = \mathsf{IRQ\text{-}16}$ and $\mathtt{ctr} = 20$, we take the cycle counter from the 20*th* occurrence of an interrupt on IRQ 16 from each trace and produce a vector $\vec{p}$. Let $\vec{p}\,[k]$ designate the k^{th} entry in this vector, and so $\vec{p}\,[k]$ is the cycle counter associated with an input from trial k.

Similarly, we identify `/dev/(u)random` output events by their position in the list of output events in a given trace. To analyze the security of output i in trial k we need to determine which input events occurred after the previous output, $i-1$, but before output i. We define this set of events as $S_i = \{\vec{p} \mid CC_{i-1} < \vec{p}\,[k] < CC_i\}$ where CC_i is the cycle counter associated with output i in trial k. Let $\ell_i = |S_i|$ be the length of S_i in a given trial.

Grouping input events into these sets is critical to the analysis: an attacker must correctly predict all inputs in S_i in order to to guess the internal state of the RNG when output i is generated. The complexity of this attack then grows exponentially with the length of S_i assuming each

Algorithm 1 findAlpha($\vec{p}$)

```
$max_\alpha \leftarrow 0$
for $\alpha = 1$ to 64 do
    if ksUniformTest($\vec{p}, \alpha$) then
        $max_\alpha \leftarrow \max(max_\alpha, \alpha)$
    end if
end for
return $max_\alpha$
```

Computes the maximum number of lower bits α that pass the KS test for uniformity. The input $\vec{p}$ is a vector of 64-bit cycle counter values.

Algorithm 2 minAlpha($\vec{p}$, E)

```
$min_\alpha \leftarrow$ uniformAlpha($\vec{p}$)
for $\vec{e} \in E$ do
    $\vec{\delta} \leftarrow \vec{p} - \vec{e}$
    $\alpha \leftarrow$ uniformAlpha($\vec{\delta}$)
    $min_\alpha \leftarrow \min(min_\alpha, \alpha)$
end for
return $min_\alpha$
```

Computes the minimum number of lower bits α that appear uniformly distributed for any given input event considering all possible offsets with previous input and output events. Here E is the set of input and output vectors that strictly precede $\vec{p}$.

input presents some uncertainty to the attacker.

We define $\alpha \geqslant 0$ as the number of lower bits of a group of input cycle counters $\vec{p}$ that appear to be uniformly distributed. For any $\vec{p}$, it is likely the case that some number of upper bits are biased (not uniformly distributed) but still provide some amount of uncertainty to the adversary. For simplicity we ignore these and focus only on the lowest α bits for any $\vec{p}$.

Statistical test for uniformity. To determine how many low bits appear to provide uncertainty, we use the Kolmogorov-Smirnov (KS) 1-sample test from [94]. The KS test examines the maximum difference between the cumulative distribution function of a set of samples compared to a reference distribution (in our case, the uniform distribution). When the maximum difference is above some threshold for a given significance level, the KS test rejects this set of samples. For a candidate value α and a given $\vec{p}$, we begin by masking the upper $(64 - \alpha)$ bits of each cycle counter in $\vec{p}$. We then compare this set of t values to the uniform distribution over $[0, 2^{\alpha} - 1]$ using the KS test. We find the largest α that passes the KS test. See Algorithm 1.

Typical significance levels for the KS test are 0.1, 0.05, 0.025, and 0.001 ([94]); we chose 0.1 which is most conservative (it favors smaller values for α). For this use of the KS test, the significance level can be thought of as the false-negative rate of the test; that is, how likely is the KS test to reject some collection of values that are randomly sampled from a uniform distribution. So our choice of a significance level of 0.1 indicates that there is a 10% chance that a given α value is rejected even if the lower α bits are sampled from a uniform distribution. This is the reason that a significance level of 0.1 produces the most conservative results compared to smaller values.

Any given input may be highly correlated with some previous input or output event and an attacker can use this correlation to her advantage when guessing cycle counters. To account for this, we also apply the above tests to relative cycle counter values. That is, for any vector of input cycle counters $\vec{p}$, we collect the set E of all vectors of cycle counters for input and output events that strictly precede $\vec{p}$. For any $\vec{e} \in E$, we compute the relative cycle counters between each component: $\vec{\delta} = \vec{p} - \vec{e}$, where $-$ is component-wise (vector) subtraction. Then we compute the maximum α that passes the KS test for uniformity among the relative cycle counter values in $\vec{\delta}$. We repeat this for every vector in E and keep only the minimum α value computed. This is shown in Algorithm 2.

Note that Algorithm 1 considers all candidates $0 \leqslant \alpha \leqslant 64$. In limited trials, we always observed $\alpha \leqslant 24$, so for efficiency of the analysis, we use this limit on the maximum α tested.

	Native			**Xen**			**VMware**			**EC2**		
i	T_i	ℓ_i	κ_i	T_i	ℓ_i	κ_i	T_i	ℓ_i	κ_i	T_i	ℓ_i	κ_i
1	0	0	0	0	0	0	0	0	0	0	0	0
2	0.9	48	129	0.1	9	129	1.0	66	784	1.1	15	134
5	1.0	0	129	0.2	0	700	1.0	0	784	1.1	0	785
10	1.2	0	129	2.1	3	1024	5.2	75	784	1.4	0	1024
15	3.6	0	129	2.1	1	1024	5.2	0	1024	2.6	1	1024

Table 2.3: Complexity estimates for predicting the early `/dev/(u)random` outputs generated during system startup. T_i is the maximum time in seconds that output i was generated; ℓ_i is median number of inputs preceding output i; and κ_i is the minimum cumulative complexity observed on this platform.

We also experimented with using a χ^2 test in place of the KS test in the procedures just described. The results were nearly identical for the same significance levels, with the KS test being slightly more conservative (it chose smaller α values). We therefore use the KS test and only report on it.

Computing complexity. The number of inputs preceding any given output varies from trial to trial on the same platform due to slight timing differences during the boot sequence. Recall that ℓ_i is the number of inputs that precede output i in a given trial. This value is a key component to the complexity of predicting the RNG's state since each input increases the attacker's search space by a multiplicative factor (assuming each input carries some uncertainty). So we compute the complexity of predicting a given set of inputs individually for each trial.

We compute the complexity of predicting the cycle counters in the input set S_i for trial k as: $s_i = \sum_{\vec{p} \in S_i} \mathsf{minAlpha}(\vec{p})$. This is the same as computing the logarithm of the size of the search tree for cycle counters when the attacker must only predict the lower $\alpha_{\vec{p}}$ bits for each $\vec{p}\,[k] \in S_i$ and these bits are each chosen independently and uniformly at random.

To determine the complexity of predicting output i in trial k we com-

VMware GRI			
i	T_i	κ_i	τ_i
1	0.3	22	27
2	0.3	33	44
5	0.4	76	94
10	0.4	109	171
15	0.5	172	248

Table 2.4: Complexity estimates for GRI outputs. τ_i is the attack complexity (in bits) of the attack we implemented.

pute $\kappa_i = \max\{s_1, s_2, \ldots, s_i\}$. We use maximum instead of a sum for simplicity, since in general the cumulative complexity is dominated by the most complex input set. Again, we compute κ_i individually for each trial k and then examine the minimum value across all trials.

To summarize, 2^{κ_i} represents a lower bound on an adversary's ability to predict the i^{th} output of the RNG during boot assuming that the low $\alpha_{\vec{p}}$ bits of the cycle counter for each input $\vec{p}$ are set uniformly at random. Unless specified otherwise, this will be the standard method for computing lower-bounds on attack complexity, and although it is a heuristic, we believe it to be a good one.

Table 2.3 shows the complexities for the platforms we tested during the first few seconds of boot. These values were computed using $t = 200$ boots on each platform using our instrumented kernel. In all cases the first output is vulnerable; see discussion below. Beyond that, our analysis shows that the lower-bounds on attacks increase very rapidly, with Xen and the native platform exhibiting the smallest complexity for the second output, an attack complexity of at least 2^{129}. The native platform reaches a min $\kappa_i = 1024$ (the minimum κ_i taken over all trials) at output $i = 140$ which is generated 4.0 seconds after boot (not shown). After 5 seconds on all platforms the attack complexity reaches the maximal value for this RNG: 1024 bits. Note that the times of outputs reported in this table are relative to the time the Linux kernel is initialized, which does not include the time in the VM manager's startup and guest boot loader (e.g., this is about 3.5 seconds in VMware).

We observe that very large input sets dominate the cumulative attack complexity κ_i, which is not surprising. In all trials on all platforms, we observed $\max \ell_i \geqslant 395$ in the first 5 seconds after boot, that is, all boots have at least one set of 395 or more inputs. This means that each input cycle counter needs to carry only 2.6 bits of uncertainty on average for κ_i to reach its maximum value of 1024. On a platform with a 1.8 GHz clock (the slowest platform we tested, EC2), this represents an average jitter for input events of 2.9 ns.

Note that this analysis assumes that the cycle counters of input events are not under the control of an attacker and that cycle counter values are not leaked to an attacker through a side channel. Although such attacks may be possible, they require an attacker to control or influence nearly all inputs to the RNG or gain knowledge of nearly all bits of each of the tens of thousands of inputs that occur during boot.

First output entropy hole. Note that Table 2.3 shows that the complexity of predicting the first output is zero. The first output of `/dev/(u)random` *always* occurs before any inputs are added to the RNG. Because the VM is supplied with zeroed memory pages, the Linux RNGs always start in a predictable state and so this output is deterministic. We observe this behavior on both VMware and Xen. The first output, always the 64-bit hexadecimal value `0x22DAE2A8 862AAA4E`, is used by boot_init_stack_protector. The current cycle counter is added to this RNG output (the canary equals $CC + (CC \ll 32)$ where CC is cycle counter) to initialize the stack canary of the init process. Fortunately, the cycle counter adds some unpredictability, but our analysis of the first GRI output (see Section 2.3) indicates that cycle counters early in boot carry about 27 bits of uncertainty, which is significantly weaker than the ideal security of 64-bits for a uniformly random stack canary.

Minimum Entropy Analysis

We also perform analysis similar to the method used in [74] where the authors examine inputs to `/dev/(u)random` during boot on embedded platforms. In that method, the authors compute Pearson correlation coefficients ($-1 \leqslant \rho \leqslant 1$) between all pairs of inputs, exclude any input with $|\rho| > 0.4$, and then compute a lower bound estimate of the security of `/dev/(u)random` by summing the empirical minimum entropy (H_∞) for each input during boot. The authors ignore security of individual outputs and instead produce a final lower bound for the RNG once boot has completed.

We perform a similar analysis on our datasets. We group our inputs according to input type and construct input sets as we do the in the KS analysis. We then mask the cycle counters using the minimum α value computed using the KS method and compute Pearson correlation coefficient ρ for every pair of inputs and exclude any with $|\rho| > 0.4$ (no such pairs were found). We use 0.4 as our threshold simply because this is the same value used in prior work of [74]. This is a somewhat arbitrary threshold, but generally speaking $|\rho| \leqslant 0.4$ implies little or no correlation among the values being compared.

We compute the empirical min-entropy H_∞ for each input by counting the number of unique values that appear after masking all but the lower α bits of each cycle counter and taking the $\log_2$. We then compute a lower bound on the security of each output i by summing the empirical min-entropy of each input in the set. Again, we compute two values, the first is the complexity of predicting all the inputs in a given set for output i. Recall that for a given trial k, S_i is the set of inputs that precede output i (but occur after output $i-1$). We define the lower-bound for the complexity of predicting this set as $\sigma_i = \sum_{\vec{p} \in S_i} H_\infty(\vec{p})$. For a given trial, the cumulative

	Native		**Xen**		**VMware**		**EC2**	
i	σ_i	λ_i	σ_i	λ_i	σ_i	λ_i	σ_i	λ_i
1	0	0	0	0	0	0	0	0
2	77	77	68	68	452	445	114	69
5	0	77	0	485	0	452	0	390
10	0	77	0	809	515	445	0	668
15	0	77	0	809	0	515	0	668

Table 2.5: Complexity lower bounds for predicting inputs during boot using the min-entropy analysis method after masking all but the lower α bits of each cycle counter. σ_i is the median input complexity (among all trials) for output i and λ_i is the minimum cumulative complexity (among all trials).

complexity λ_i for output i is again the maximum input set complexity $\lambda_i = \max\{\sigma_1, \ldots, \sigma_i\}$. Since the size of input sets vary from trial to trial, we compute σ and λ individually for each trial and then report on the median σ values and minimum λ values among all trials in our dataset.

Table 2.5 shows the results of this analysis. In all cases these results are more conservative than the method using just the KS test, but we note that with $t = 200$ trials, each input is limited to contributing at most $\log_2 200 = 7.6$ bits of entropy whereas the KS test often estimates $\alpha > 7.6$. We note that many inputs show $H_\infty(\vec{p}) = 7.6$, and so the estimate appears restricted by the number of trials in our dataset. Still, after masking for only the lower α bits, we find no pairs of inputs with correlations larger than 0.4 on any platform which provides further evidence that these lower bits are uniformly distributed and thus difficult for an attacker to predict.

GRI boot-time analysis

To predict the 32-bit output of the GRI RNG, an attacker needs to know the state of the GRI before the call (HV and S, 128-bits and 512-bits, respectively) as well as the inputs used (J, CC, and P). When booting on a VM, the initial state of HV and S are all zeroes. S remains zero until it is initialized from `/dev/(u)random` (after approximately 100 calls to GRI in our observations). If the correlation between the jiffies counter J and the cycle counter CC is known (they are based on the same underlying clock), then the only unknown to an attacker is CC at the time each output is generated. The worst case scenario occurs on VMware where the cycle counter is virtualized by default, and so begins at 0 each time a VM is booted. In our experiments with VMware, we observed only 2 unique values of J at the time the first call to GRI is made. So if an attacker correctly guesses the CC and its associated J value, then future values of J can be computed using the ratio of cycles to timer ticks. We therefore focus only on the cycle counter.

We use a complexity estimate similar to that in the last section, except that when bounding the complexity for output i of the GRI RNG we do not assume the attacker knows the prior outputs. If we did, then each output would only have as much uncertainty as a single cycle counter carries — the GRI RNG does not attempt to deal with checkpointing attacks and never adds entropy except during output generation. For GRI, we define s_i to be the minimum number of lower bits α that appear uniformly distributed across the cycle counters used when output i is generated across all t trials. We use the same algorithm for computing α as we use for `/dev/(u)random`. Our computation of κ_i for GRI differs, we define κ_i as the sum of all preceding s_j values: $\kappa_i = \sum_{j \in [i]} s_j$. Again, this is because we are excluding checkpointing attacks.

Table 2.3 shows the resulting complexity estimates κ_i for the first few outputs i of GRI from 200 boots on VMware (results on Xen and EC2 were similar). If we exclude the secret value S from GRI, which is a known value at the start of boot, then GRI has a maximal security state of 128-bits (the size of its hash chaining variable). GRI reaches this state after 10 calls, well before the secret value S is initialized at approximately the 100*th* call. For the second output and beyond, predicting the internal state by guessing inputs is no easier than guessing any single 32-bit output value. The first value, however, shows less than ideal security for κ_1. We explore this next.

Predicting early GRI outputs. To confirm that, in fact, there is a vulnerability, we build an attack that works as follows. First, we collect a dataset of multiple boots using our instrumented kernel. From each of t traces, we group all the cycle counters from the first call to GRI, all the cycle counters from the second, and so on as we did with previous complexity estimates. Now, however, we select a *range* of cycle counters at each depth to include in the attack. To make the attack more efficient, we search the smallest contiguous range that covers a fraction (we use 80%) of the observed cycle counters in the dataset. This excludes some outliers and provides a moderate speedup of the attack time. We let τ_i denote the logarithm of the search space resulting from this process. Table 2.3 shows the values of τ_i for the first few outputs using our dataset of 200 boots on VMware with the instrumented kernel. Again, only the first output is weaker than the desired 32-bits of security.

To evaluate this interpolated attack model we analyzed the first call to GRI experimentally. We constructed a kernel with minimal instrumentation to capture only the calls to GRI and performed 100 boots on VMware. We remove one trace from this dataset (the victim) and train an attack on the remaining traces to identify a range of possible values for the cycle counter CC. The remaining values (HV, J, P and S) are trivially known for the first call on this platform. We use a GRI simulator and iterate over the identified range of values for CC. The attack is successful and we verify that we can produce the full internal state HV, not just the output value. This is useful for validation since collisions are very likely when one tests up to 2^{27} guesses for a 32-bit number; the probability of a collision is 1 in 32.

A successful attack indicates that security is less than it should be for the first output. However, we note that taking advantage of this would require the ability to test which of the 2^{27} values are correct. This value is the stack canary for the `kthreadd` (kernel thread daemon) process. It is not clear that this weakness can be exploited, but this represents a failure of the RNG.

2.4 Snapshot Resumption RNG Security

Modern VM managers allow users to pause a running VM, make a copy of the entire state (called a snapshot) of the VM including CPU registers, memory, and disk, and later use that copy to restart the VM in the exact state at which it was paused. Both Xen and VMware support this, though Amazon EC2 does not, nor do any other clouds to our knowledge. Nevertheless, snapshots are often used in other settings such as backups, security against browser compromise, and elsewhere (see [85, 47]).

We consider two threats related to VM snapshots. First, we consider VM reset vulnerabilities discussed in [85, 47], where resuming from a snapshot multiple times may lead to the RNG outputting the same values over and over again. Second, we consider an attacker that obtains a copy of the VM snapshot (e.g., if it is distributed publicly), meaning the attacker has effectively compromised the state of the RNG at the time of the snapshot. Here the question is whether the attacker can predict outputs from the RNG or if, instead, the RNG is able to build up sufficient fresh entropy to recover from the compromise.

Reset vulnerabilities (`/dev/(u)random`)

We show that `/dev/(u)random` suffers from VM reset vulnerabilities: the RNG will return the same output in two different VM resumptions from the same snapshot. There are several different situations that give rise to this vulnerability, all related to the values of the relative entropy counters

Situation	Snapshot state	Repeats until	# bits
(1) Cached entropy	$\mathsf{UP.ec} \in [8, 56]$	$\mathsf{UP.ec} = 0$	$\mathsf{UP.ec}$
(2) Racing fast pool	Any	IntP overflow	∞
(3) Transfer threshold	$\mathsf{IP.ec} < 192$	$\mathsf{IP.ec} \geqslant 192$	∞

Table 2.6: Three situations leading to reset vulnerabilities with `/dev/urandom`. The symbol ∞ represents no limit on the number of repeated output bits before the condition in the third column is met.

and other state at the time the snapshot is taken. Table 2.6 summarizes three situations that we have observed lead to reset vulnerabilities with regards to `/dev/urandom`. Note that these situations are not mutually exclusive, though we will exercise them individually in our experiments. As discussed below, these situations can also cause repeated outputs from `/dev/random`.

We use the following method to exhibit reset vulnerabilities. A guest VM boots under its default, unmodified kernel and runs for 5 minutes to reach an idle state and starts a userland measurement process designed to: detect a VM reset, capture the input pool entropy count (using /proc/fs) upon resumption, and perform a series of 512-bit reads from `/dev/urandom` every 500 µs until the experiment completes. To perform VM reset detection, the userland measurement process runs a loop that samples the (non-virtualized) cycle counter using the `rdtsc` instruction and then sleeps briefly (100 µs). When a sufficiently large discrepancy between subsequent cycle counters is detected (we use 6.6 billion cycles, which is about 2 seconds), the detection process exits the loop and begins reading values from `/dev/urandom`. Thus we begin capturing outputs from `/dev/urandom` immediately after snapshot resumption. For each experiment, we captured 10 snapshots while the system is idle, performed 10 resets from each snapshot and examined the resulting RNG outputs.

We performed experiments on both Xen and VMware. However, we experienced occasional errors when resuming from a snapshot on Xen: the guest would occasionally declare the filesystem readonly (presumably because of an error upon resumption) and so below we only report on resumptions that succeed. We experienced no such errors using VMware.

For each 512-bit output produced by `/dev/urandom`, we declare an output a *repeat* if a full match of all 512 bits occurs in any output from a different reset of the same snapshot. Note that at 512 bits, a repeat can only occur if the same RNG state was used (otherwise SHA-1 collisions would have occurred).

(1) Cached entropy. Recall that if the entropy estimate of a secondary pool (UP or RP) has an entropy count greater or equal to the number of output bits requested, then the output is generated directly from the secondary pool without pulling fresh bits from the input pool IP. We also note that no cycle counter (or other time stamp value) is added into the hash at this point in time, which means that the output of such calls after a reset are fully determined by the state of the secondary pool at the time of the snapshot.

If the `/dev/urandom` entropy count has a value of $\mathsf{UP.ec} = 8n$ for $n > 0$ at the time of snapshot, then the bits in the non-blocking UP pool will be used to satisfy any request of size $\leqslant 8n$ bits without transferring bits from the input pool. Since the output generation algorithm is deterministic, this results in repeated output of size $\leqslant 8n$ bits under these conditions. $\mathsf{UP.ec}$ has a maximum value of 56 bits because of the internal mechanics of the RNG and so the maximum repeated output length is n bytes where $n \leqslant \mathsf{UP.ec} \leqslant 7$. The conditions are the same for `/dev/random`.

(2) Racing the fast pool. Even if a transfer from the input pool occurs after reset, this alone is does not prevent repeat outputs. To generate unique outputs, the RNG requires at least one new input to the input pool *and* a transfer from the input pool to the secondary pool (UP or RP). After a reset, the most likely addition to the input pool is from the function add_interrupt_randomness() as these account for an overwhelming majority of `/dev/(u)random` inputs. As described earlier, these inputs are buffered in the interrupt pool (also called the fast pool) until an overflow event occurs and the contents of the interrupt pool are mixed into the input pool. This creates a race condition between interrupt pool overflow events and reads from `/dev/(u)random`. An overflow event occurs every 64 interrupts or if 1 second has passed since the last overflow when an interrupt input is received. During this window, reads to `/dev/urandom` of arbitrary size will produce repeated outputs.

For `/dev/random`, repeated outputs will occur during the same window until `/dev/random` blocks for new entropy. Thus the maximum number of repeated bits from `/dev/random` is 4088. To exercise this situation for `/dev/urandom` we used the experimental procedure above. Because we are comparing 512-bit output values, we can rule out repeats caused by situation (1), discussed above. To exclude situation (3) discussed below (which doesn't involve the input or fast pool), we want the input pool entropy count to be much higher than 192. We achieve this by downloading a large file (1GB) prior to capturing the snapshot. The inbound packets from the download drive interrupts in the guest kernel which increases the input pool entropy count. All resumption had an initial input pool entropy count of at least 1,283 on both Xen and VMware.

On Xen, one snapshot produced no repeated outputs (we didn't win the race), and the remaining 7 snapshots exhibited at least one repeated 512-bit output (the first output requested) after resumption. Of these the maximum duration for repeats was 1.7s after resumption. This demonstrates that the RNG does a poor job of updating its state after resumption, due to the (overly) complicated pool structure and pool-transfer rules.

On VMware, in 20 snapshots with 10 resets from each snapshot, we were not able to exhibit this vulnerability using an unmodified kernel.

(3) Input pool entropy count below threshold. The input pool entropy count IP.ec must reach the transfer threshold of 192 bits before fresh inputs are transferred from the input pool to the non-blocking pool UP. While the RNG is in this state, an unlimited quantity of repeatable output values can be generated from `/dev/urandom`. For `/dev/random` of course, this is not true, as repeat values will only be provided until the entropy estimate for the blocking RP pool is exhausted (as per situation (1) above).

To arrange this situation, immediately before capturing the snapshot, we execute a 10 second read from `/dev/random` to reduce the input pool entropy count below 192 and trigger this condition.

On both VMware and Xen, the maximum value for IP.ec upon resumption was 48 — sufficient to put the RNG into situation (3). On both VMware and Xen, we observed that *all snapshots* produced repeat outputs for the duration of the experiment (30 seconds). This indicates that if IP.ec is very low when a snapshot is captured, it may take more than 30 seconds for the `/dev/random` RNG to reach a secure state.

Entropy starvation attack for situation (3). In Section 2.1 we observed that there exists a simple entropy starvation attack against `/dev/urandom`, where a (malicious) user-level process simply performs continuous reads from `/dev/random`. The internal logic of the RNG is such that in this case the input pool will always transfer to the blocking RP pool, and never the UP pool. This can be used to extend the amount of time that `/dev/urandom` produces repeated outputs in situation (3) where the input pool entropy count is below the threshold to transfer bits from IP to UP. An adversary with the ability to run an unprivileged process on the system can easily engage this condition by reading from `/dev/random`. If a remote attacker makes (legitimate) requests to a public interface that triggers large or frequent reads from `/dev/random`, then the same effect may be possible without requiring a local account.

The experimental procedure above was used with the following deviations. We first execute a continuous read from `/dev/random` using the command `dd if=/dev/random`. After reset, the measurement process performs 512-bit reads from `/dev/urandom` every 1 second for a duration of 120 seconds. Upon resumption, *all* snapshots exhibited repeated 512-bit outputs for the duration of the experiment on both VMware and Xen.

Impact on OpenSSL. The experiments above show that reset vulnerabilities exist in `/dev/(u)random`, and give applications stale random values after resumption. We now briefly investigate the potential for this to lead to exploitable vulnerabilities against applications relying on `/dev/urandom` for randomness after a VM resumption. We focus on OpenSSL v1.0.1e and RSA key generation. When calling `openssl genrsa` from the command line, OpenSSL seeds its internal RNG with 32 bytes read from `/dev/urandom` as well as the current system time, process ID, and dirty memory buffers. We instrument this version of OpenSSL in order to observe internal values of the key generation process. We then set up a VM running an unmodified Linux kernel on VMware that will, immediately after being reset, execute the command `openssl genrsa` from the shell. We observe that just connecting to the VM via SSH to prepare it for a snapshot typically drives the input pool entropy count below 192 before we take a snapshot. This is caused because a number of processes are created during login and each new process consumes many bytes from `/dev/urandom` to initialize stack canaries and perform ASLR.

We captured 27 snapshots, performed 2 resets from each snapshot and then analyzed the resulting outputs from the OpenSSL instrumentation and OpenSSL's normal output. A single snapshot produced an *identical* prime `p` in the private key in both resets, but other values in the private key differed. Presumably, after the prime `p` was generated, differing dirty memory buffers caused the OpenSSL RNGs to diverge. (Knowing one prime value of a private key is sufficient to derive the other and destroys the security of an RSA private key.) Of the remaining 26 snapshots, many had identical `/dev/urandom` output, but typically the dirty memory buffers differed early enough in execution to produce unique outputs. These dirty memory buffers are likely different between resets because Address Space Layout Randomization (ASLR) (determined in part by GRI) shifts around the OpenSSL memory layout.

To validate this hypothesis, we disabled ASLR on the guest with the command `echo 0 > /proc/sys/kernel/randomize_va_space` and repeat our experiment for 30 snapshots with 2 resets from each snapshot. Of these, 23 snapshots produced repeated output from `/dev/urandom` and *identical RSA private keys*. The other 7 snapshots had input at least 1 differing value into the OpenSSL RNG after reset — variously this differing value was one of `/dev/urandom` output, PID, or system time.

We note that unlike prior reset vulnerabilities shown in [85], these are the first to be shown in which the system RNG is invoked after VM resumption. [85] ask whether consuming fresh random bytes from the system RNG after a reset is sufficient to eliminate reset vulnerabilities in applications. These results answer that question in the negative, and highlights clear problems with the `/dev/(u)random` design for settings where snapshots are employed.

Reset vulnerabilities (GRI)

As described in Section 2.1, the output of the GRI RNG depends only on the state values HV and secret S and the inputs cycle counter, jiffies and PID (CC, J, P). Across multiple resets from the same snapshot, it's very plausible for the same process (with same PID P) to be the first to request an output. So the only new information after a snapshot resumption is the cycle counter value. For a virtualized cycle counter, in which the cycle counter value will always start from the same value (stored in the snapshot), we might expect reset vulnerabilities. In fact we observe no repeated values output by GRI across any of its invocations in any of the 50 resets on VMware that we performed. This can likely be attributed to small variations in timing between snapshot resumption and the first call to GRI. For 10,000 Xen resets, with the non-virtualized RDTSC, we did not see any repeats as well.

Reset vulnerabilities (other RNGs)

FreeBSD. To see if reset vulnerabilities affect other RNG designs, we also perform a limited set of experiments with snapshot resumptions using an (uninstrumented) version of FreeBSD within VMware. A description of FreeBSD's design appears in [56]. We used a similar userland tool to detect resets and sample from FreeBSD's `/dev/random` interface (same as `/dev/urandom` on FreeBSD). Repeat outputs on reset were common on FreeBSD but the duration of repeats depended on the internal state of the RNG and parameters of each experiment. The maximum length of time that we observed repeats in our experiments was 100 seconds (taking 1 byte samples every 1 second) and the maximum number of repeated output bytes we observed was 7.5 KB (taking 512-bit samples every 1 ms for 120 ms).

Windows 7. We perform similar experiments on Microsoft Windows 7 running in VMware using multiple different random number generator interfaces. We tested: the rand_s interface (for C-programs using stdlib); the CryptGenRandom interface (for Win32 applications); and the Rng-CryptoServiceProvider interface (for .NET applications).

In experiments with rand_s, *all* resets produced repeated outputs when reset from the same snapshot more than once, though the quantity of repeats varied. In one experiment, we performed 10 resets from the same snapshot and collected 32-bit outputs every 1 second for a total of 2000 samples (collected over more than 30 minutes). We found more than 500 (25%) repeated outputs shared between each pair of resets, and some pairs have 1000 (50%) repeated outputs. We also observe that all 2000 outputs generated in the first reset are found in some combination of the following nine resets.

Experiments with the CryptGenRandom and RngCryptoServiceProvider interfaces produced repeats as well. In all experiments we observed identical output sequences from all resets of the same snapshot. In particular, we collected 256-bit outputs every 30 seconds for a duration of 30 minutes. In our experiments, there is no evidence of a time limit after which the Windows RNGs stop producing repeated outputs (unlike the Linux and FreeBSD implementations). These security vulnerabilities have been reported to Microsoft.

These experiments on FreeBSD and Windows demonstrate that RNG reset vulnerabilities are not limited to the Linux RNGs.

Snapshot Compromise Vulnerabilities

If a snapshot is disclosed to an attacker, then one must assume that all of the memory contents are available to them. Not only is there likely to be data in memory of immediate damage to an unwitting future user of the snapshot (e.g., secret keys cached in memory), but the RNG state is exposed. While we can't hope to prevent cached secret keys from leaking, we might hope that the RNG recovers from this state compromise when later run from the snapshot. As we saw above, predicting future `/dev/(u)random` in various situations is trivial since the attacker can often just run the snapshot (on similar hardware). When not in these situations, however, and for GRI, we would like to estimate the complexity of using the compromised state to predict outputs generated after a snapshot resumption.

We use the same methodology as used above with Xen, with the workload that reads from `/dev/urandom` repeatedly after snapshot resump-

	`/dev/(u)random`			GRI	
i	T_i	ℓ_i	κ_i	T_i	κ_i
1	0.7 ms	2	0	21 s	22
2	1.4 ms	2	20	21 s	33
5	4.1 ms	2	27	21 s	66
10	7.1 ms	2	27	21 s	105

Table 2.7: The minimum estimated complexity κ_i to predict the first few outputs of `/dev/(u)random` and GRI after a Xen guest is reset from a snapshot. T_i is the minimum time that output i is generated (relative to resumption); ℓ_i is the median sequence length.

tion. We then use our methodology from Section 2.3 to give lower-bound estimates on the complexity of predicting the very first few outputs to `/dev/urandom` or GRI.

Table 2.7 shows our estimated attack complexity after reset. The complexity estimates for the `/dev/urandom` outputs are much smaller than for their boot time counterparts (Table 2.3 in Section 2.3). The security of the GRI outputs is similar to boot because GRI security under our model is driven only by the cumulative uncertainty of the cycle counters from each output request. However, `/dev/urandom` outputs have security dominated by the input sequence length ℓ_i. There are far fewer inputs during a resumption than at boot. This suggests possible vulnerability to prediction attacks, but for brevity we do not pursue them further having already shown above that repeats give rise to predictable outputs.

2.5 The Whirlwind RNG

In this section we detail the Whirlwind RNG, which provides a simpler, faster, and more secure randomness service. While our measurement study focused primarily on virtual environments, the design of Whirlwind seeks to provide security for a variety of settings and in general be a drop-in replacement for both `/dev/(u)random` and GRI. As such, we must handle a variety of goals:

- *Simplicity*: The current `/dev/(u)random` design is complex, requiring significant effort to understand and audit its design and implementation (with 1041 lines of code); see [52]. In contrast, Whirlwind targets simplicity and requires 676 lines of code.

- *Virtualization security*: Unlike all prior RNG designs we are aware of, Whirlwind is explicitly designed to provide security even in virtualized environments that might entail VM snapshot and image reuse.

- *Fast entropy addition*: Whirlwind uses a simple entropy gathering function designed to be fast, usually it requires only 0.5 μs on our 2.67 GHz platform, though for 1/8 of the invocations it applies the SHA-512 compression function ([80]). Despite using a slower hash function (SHA-512), we show it to be about as fast as entropy addition in the current `/dev/(u)random`. Whirlwind uses per-CPU input buffers to reduce lock contention and permit the amount of buffered inputs to scale with the number of CPUs.

- *Cryptographically sound*: We propose a new design for the cryptographic core of Whirlwind, inspired by the recent work of [39]. Whirlwind dispenses with the linear feedback shift registers of Linux `/dev/(u)random`, and achieves the robustness security goal detailed in [39].

- *Immediately deployable*: The basic Whirlwind design is a drop-in replacement for Linux `/dev/(u)random`, and requires no hypervisor support.

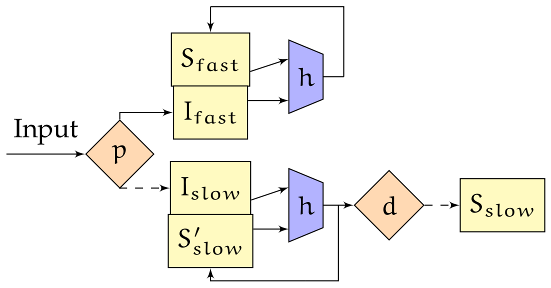


Figure 2.3: Block diagram of the Whirlwind RNG. Every p^{th} input is directed to the slow pool; after d updates it is used in output generation. h is the SHA-512 compression function.

Whirlwind design

Figure 2.3 depicts the main components of Whirlwind. It uses two entropy pools, a fast pool and a slow pool, as done in FreeBSD's Yarrow RNG described in [56]. The fast pool consists of a per-CPU input buffer I_{fast} and a single (global) seed value S_{fast} for the fast pool. The slow pool consists of a per-CPU input buffer I_{slow}, a private (internal) seed S'_{slow}, and a public seed S_{slow}. In our implementation all input buffers are 1024 bits in size which corresponds to one full message block for SHA-512. All three seeds in our implementation are 512 bits, which represents a chaining value for SHA-512. We denote the SHA-512 compression function by h and the SHA-512 hash function by H. Let n be the number of bits of output for both h and H. We initialize the seeds values as: $S_{fast} \leftarrow h(IV, 1)$ and $S_{fast} \leftarrow h(IV, 2)$ where IV is the SHA-512 initialization vector and 1 and 2 are encoded in some unambiguous manner.

Inputs are written to the fast pool I_{fast} by default and every p^{th} input is diverted to the slow pool I_{slow}. In our implementation $p = 10$ which ensures that the fast pool receives the majority of inputs and thus changes rapidly even in low-entropy conditions. Each input is 128-bits and consists of the input source's unique identifier (created by the GCC macro `__COUNTER__` and encoded using 32 bits), the lower 32 bits of the cycle counter (or jiffies on platforms without a valid cycle counter), and 64 bits of optional, source-provided information. Input buffers are per-CPU, obviating the need for locking to process most inputs. We denote the macro used for adding inputs by ww_add_input().

When an input pool is full (after 8 inputs are written to a pool), a SHA-512 compression function application is performed, with the chaining variable equal to the pool seed value S_{fast} or S'_{slow} and the message block equal to the input pool. The result becomes the new seed for that pool. Locks are used to ensure that the compression function is computed atomically. Thus, Whirlwind is computing a hash over the sequence of inputs in an online fashion. This ensures the robustness security property introduced by [39] and which they showed Linux's `/dev/(u)random` fails to achieve. Robustness requires (informally speaking) that no matter where entropy resides in the sequence of inputs to the RNG, the RNG outputs always benefit from the added entropy.

In the case of the slow pool, the internal seed S'_{slow} is used as the hash chaining value and upon every d^{th} hash the internal seed S'_{slow} is copied to the public seed S_{slow}. This ensures that the slow pool represents a multiple of d times as many inputs as the fast pool. In our implementation $d = 50$, which, combined with $p = 10$, means the public slow seed is updated every 500 inputs.

Consumers within the kernel request random values from Whirlwind using get_random_int() or get_random_bytes(). From user mode, processes read random values via the existing `/dev/random` or `/dev/urandom` read interfaces. Whirlwind handles *all* such requests in the same manner and, in particular, we have completely removed the GRI RNG and we do not differentiate between `/dev/random` and `/dev/urandom`. The current implementation does not support writing to the RNG from user-level processes, though it would be easy to add.

Algorithm 3 describes output generation in pseudocode. When Whirl-

Algorithm 3 ww_generate_bytes(b)

```
$s_1 \leftarrow S_{fast}$
$s_2 \leftarrow S_{slow}$
$t \leftarrow \lceil 8b/n \rceil$
$ctr \leftarrow$ atomic_inc$(Ctr, t) - t$
$hw \leftarrow$ read_hw_random()
ww_add_input()
for $i = 0$ to $t$ do
    $CC \leftarrow$ get_cycle_counter()
    output[i] $\leftarrow H(3 \| s_1 \| s_2 \| (ctr + i) \| CC \| P \| hw)$
end for
ww_add_input()
$S_{fast} \leftarrow h(S_{fast}, 0^{1024})$
return first $b$ bytes of output
```

Routine for generating b bytes of output from the Whirlwind RNG. The variable Ctr is a global output counter.

wind receives an output request for b bytes, the RNG first copies the slow and fast pool seeds from static (global) memory into local memory on the stack. Whirlwind then prepares a response by computing a SHA-512 hash over the concatenation of: (1) the local copy of the slow pool seed; (2) the local copy of the fast pool seed; (3) a 64-bit request counter Ctr; (4) the current cycle counter CC; and (5) 64-bits read from a CPU hardware RNG (e.g., RDRAND), if available. The request counter Ctr is atomically pre-incremented for the number of blocks requested (to reserve counter values for output generation) and is incremented locally for each block of output. This ensures that even if concurrent requests have identical values (seeds, P, CC) the outputs are guaranteed to be unique. Two inputs are fed back into the RNG for each output requested. Finally, a single application of h is used to ensure forward security.

Initializing Whirlwind. We also include one special mechanism for quickly initializing (or refreshing) the entropy of Whirlwind, which is needed to prevent a boot-time entropy hole (like the ones in the legacy

Algorithm 4 ww_bootstrap()

```
for $i \leftarrow 0$ to $\ell$ do
    $CC \leftarrow$ get_cycle_counter()
    ww_add_input()
    $k \leftarrow CC \bmod \ell_{max}$
    for $j \leftarrow 0$ to $k$ do
        $a \leftarrow (j/(CC+1)) - (a * i)$
    end for
end for
```

The Whirlwind entropy bootstrapping mechanism used during boot and snapshot resumption. The values ℓ and ℓ_{max} are configured parameters (default 100, 1024).

RNG, see Section 2.3) and to recover from a VM reset. For boot time, we would have liked to use the recent suggestion of [74] to quickly generate entropy in the initial stages of boot via timing of functions in the kernel init function. Unfortunately, this is not fast enough for us, since we observe reads to the RNG early in init. We therefore use an approach based on timing of instructions that may take a variable number of cycles, which has been suggested and used previously, see [1, 76]. This provides non-determinism (by way of contention and races within the CPU state), as shown in prior studies by [68]. Pseudocode is shown in Algorithm 4. In our implementation we have $\ell = 100$ and $\ell_{max} = 1024$.

Whirlwind calls this entropy timing loop before the first use of the RNG during boot, and at the start of resumption from a snapshot. The latter takes advantage of Xen's *resume callback*, which is a virtual interrupt delivered to the guest OS when it first starts following a snapshot resumption. Similar facilities exist in other hypervisors.

Entropy sources. It is easy to add entropy sources to Whirlwind, by simply inserting ww_add_input() in appropriate places. This requires no understanding of RNG internals (such as the role of entropy estimates), unlike in the existing Linux `/dev/(u)random`. In terms of performance, submitting an input to the RNG is fast, but may still require a single

SHA-512 compression function call on the critical path. While we expect that, in deployment, Whirlwind might use a wider set of entropy sources, for comparison purposes, we restrict our experiments here to use only the same set of entropy sources as used by the current `/dev/(u)random` implementation in Linux as well as those called in ww_bootstrap() and ww_generate_bytes().

Hypervisor-provided entropy. As we show below, the existing software-based sources are already sufficient to provide security during boots and resets. Some users may nevertheless desire (for defense-in-depth) support for the Xen management Dom0 VM (also running Whirlwind) to provide an additional entropy source for a guest VM's Whirlwind RNG. In current practice, host-to-guest entropy injection is facilitated via virtual hardware RNGs, that then are fed into the Linux `/dev/(u)random` by way of a user-level daemon (rngd). Unlike these systems, we will ensure host-provided entropy is inserted into Whirlwind immediately after a VM resumption, before any outputs are generated.

To do so, we pass additional entropy with the *Xenstore* facility in Xen, which uses shared memory pages between Dom0 (the management VM) and the guest VM to provide a hierarchical key-value store. We modified Dom0 to read 128 bytes from `/dev/urandom` and write the value to Xenstore. During a resume callback, Whirlwind detects that a reset occurred, reads the value from Xenstore and adds the value to the RNG via repeated input events. All this requires less than 30 lines of modification to Xen's operation library `libxl`. The entire operation requires 75 ms on average, and the rareness of the operation (once per resumption) makes this tolerable.

Other instantiations. For concreteness, we chose several suggested values of (sometimes implicit) parameters, but it is easy to modify the Whirlwind implementation to support different choices. For instance, instead of letting `h` be the SHA-512 compression function, one could use the full

SHA-512 (or some other secure hash, such as SHA-3), which leads to the RNG computing a hash chain. The approach detailed is faster because it reduces the number of compression function calls. One might also use SHA-256, smaller or larger seed values (to trigger hashing more or less frequently), and the like. Additionally, we choose the output generation hash H as the full SHA-512. Again, this can be replaced with any suitable hash function or even AES in a one-way mode such as Davies-Meyer mode ([97]).

Security evaluation

We evaluate the boot-time and reset security of Whirlwind. We perform 50 reboots in Xen using an instrumented version of the Linux kernel using Whirlwind. We also perform 50 resets from a single Xen snapshot captured while idling (5 minutes after boot); a user level process requests 512-bit outputs from the RNG every 500 μs after resumption. As before, the instrumentation records all inputs and outputs to the Whirlwind RNG. We then perform complexity analysis as done for the legacy `/dev/(u)random` (see Section 2.3), which again ignores all input sources except the cycle counter. This provides a conservative estimate of unpredictability from the attacker's perspective. As intended, the adversarial uncertainty regarding the Whirlwind internal state hits 1024 (the maximal amount) before the first use of the RNG either during boot or after a reset. An immediate implication is that reset vulnerabilities are avoided: the probability of repeated output arising from reuse of the same snapshot is negligible.

We have not yet evaluated Whirlwind's entropy accumulation on low-end systems, such as embedded systems studied by[74, 53]. In particular, here the cycle timing loop may provide less uncertainty because embedded system CPUs themselves have less non-determinism. In these settings, however, we do not expect to be using VM snapshots (making this use moot) and for generating entropy at boot we can use the techniques of [74].

Throughput (`/dev/urandom`)		
Block size	**Whirlwind (MB/s)**	**Legacy (MB/s)**
4 bytes	0.6	1.6
16 bytes	2.3	4.9
64 bytes	9.3	9.0
256 bytes	21.8	12.0

(Larger is better)

Latency (`/dev/urandom`)		
Block Size	**Whirlwind** (μs)	**Legacy** (μs)
4 bytes	6.9	2.5
16 bytes	6.9	3.3
64 bytes	6.9	7.0
256 bytes	11.7	21.3

(Smaller is better)

Table 2.8: Comparing performance of the Whirlwind and legacy `/dev/urandom` implementations using `dd` to read 10,000 blocks of various sizes. Latency values are derived from throughput measurements.

Performance evaluation

We turn to evaluating the performance of Whirlwind, particularly in comparison to the existing `/dev/(u)random` and GRI RNGs. Our Whirlwind implementation uses SHA-512 as opposed to SHA-1 (resp. MD5 for GRI), so we expect to see a performance penalty from the use of stronger cryptography. To compare, we evaluated the throughput of reading from `/dev/urandom` and GRI for both Whirlwind and the legacy RNGs. While the system is otherwise idle, we execute reads of 10,000 blocks on the `/dev/urandom` interface for various block sizes using `dd`. We repeat this 100 times for each block size and report the average throughput in Figure 2.8. As expected, the legacy RNG performs slightly better at smaller block sizes ($\leqslant$ 16 byte), but is outperformed by Whirlwind at 64 and 256 byte block sizes.

We also compare measured performance of adding inputs to the new and legacy RNGs. We add minimal instrumentation to time these operations and measure performance during VM boots, resets, and during idle time. We also use these runs to measure performance of reading from GRI. The resulting performance data (shown in Figure 2.9) indicate the various functions were timed more than 100,000 times for each RNG. The results are that while input processing for `/dev/(u)random` is as fast in Whirlwind as in the legacy RNG, the GRI output interface requires 10.3 µs (one standard deviation is ±1.8) for Whirlwind but the legacy RNG requires only 1.0 µs (±0.5). The standard deviation is higher for Whirlwind, as this implementation more frequently performs hash operations than the legacy RNG.

Note that GRI is only used during process creation. In order to understand whether the GRI slowdown will cause problems in applications, we run the fork benchmark from LMbench by [71] 100 times. The average latency of fork is 414 µs (with standard deviation ±5 µs) for the legacy RNG, and 418 µs (±5 µs) for kernel with Whirlwind. Thus Whirlwind incurs only 1% overhead in this (worst-case) benchmark, and so we believe this is not a problem for practical use.

Lastly, we evaluate the overhead of ww_bootstrap() (Algorithm 4) used at boot and snapshot resumption. The time to execute the timing loops has a mean of 0.7 ms over 50 runs. Boot and snapshot resumption are rare operations, suggesting this level of overhead will not impact deployments.

Overall we conclude that Whirlwind has performance closely matching the existing RNGs, and in some cases even better despite using more expensive (and more secure!) cryptographic primitives. For this, we get a significantly simplified design and improved security.

Execution Time	**Whirlwind (μs)**	**Legacy (μs)**
GRI	10.3 (±1.8)	1.0 (±0.5)
`/dev/(u)random` Input	0.5 (±0.4)	0.5 (±0.1)

Table 2.9: Performance results for outputs from GRI and input processing for `/dev/(u)random` inputs over 100,000 invocations. Standard deviation shown in parentheses.

2.6 Preventing Reset Vulnerabilities in Legacy Guests

While Whirlwind prevents VM snapshot reset vulnerabilities, it requires updating (at least) the kernel. We therefore consider in this section how one might try to prevent, particularly, reset vulnerabilities in legacy guests. We first consider a setting in which we can add user-level daemons to the guest, but cannot modify the kernel. We then consider a setting in which we cannot modify the guest VM at all.

Legacy hypervisor. We first consider a setting with a legacy hypervisor and management VM, but where we are able to install a user level daemon or kernel module into the guest VM. The goal of the daemon is to heuristically detect when a snapshot occurs. For VMs with non-virtualized time, the cycle counter (`rdtsc` instruction) returns the cycles since physical system boot. Following a resumption, the cycle counter is likely to show a large discontinuity, either a positive or negative jump. (A negative jump is possible when the machine has been rebooted or when a snapshot is reset on a different machine.) Thus, a reset can be detected by periodically polling the cycle counter, detecting large jumps (forward or backward), and assuming the cause is a reset. The overhead of such an approach is small. If the daemon sleeps 1 ms between two detection attempts, then the overhead for such daemon process uses less than 0.5% of the total CPU time. A kernel implementation that only polls when the CPU is active can avoid some potential adverse affects, e.g., preventing a CPU from going into power-saving mode.

When a reset occurs, we use the cycle timing loop discussed in the last section to generate entropy. These cycles can then be directly written to the (legacy) `/dev/urandom` interface to inject entropy. As we have evaluated the uncertainty generated by cycle timing (Section 2.5), for brevity we omit measurements here. Even so, there exists a tension between performance (how long the daemon sleeps for) and the window of opportunity for reset vulnerabilities.

Legacy guest VM. The prior approach requires installing code on the guest VM, which may not always be possible (e.g., for already-deployed VM images). We therefore investigate whether the hypervisor or a management Dom0 can be used to inject additional entropy into a guest following a snapshot resumption. The goal is to ensure that the state of the guest's legacy `/dev/(u)random` RNG becomes sufficiently refreshed so as to prevent reset vulnerabilities. Unfortunately, this is a very constrained setting, since we must work with the existing deficiencies of the `/dev/(u)random` RNG.

Our observation is that because the `/dev/(u)random` RNG uses interrupts as the primary source of entropy, the management Dom0 (or possibly even a remote host) can intentionally inject a flood of random interrupts after a reset by way of network packets. As described in Section 2.1, interrupts go first into the interrupt pool IntP, eventually flow to the input pool IP, and when the input pool entropy counter $\mathsf{IP.ec}$ exceeds 192 bits, these inputs will be pulled into the `/dev/urandom` pool UP where they affect the `/dev/urandom` outputs. (We focus on `/dev/urandom` since we observe no consumers of `/dev/random` in the default installations we tested.) Furthermore, interrupts only flow from IntP to IP at most every 1 second or after 64 interrupts, and then only add 1 count to the input pool en-

Inter-send time (μs)	**Transition Time**
100	3.6s
500	11.2s
1,000	16.6s
10,000	38.0s
no interrupts	$>$ 50s

Table 2.10: Average time after VM resumption for `/dev/urandom` entropy pool to be refreshed when injecting interrupts from Dom0 at various frequencies (shown as inter-send time in microseconds).

tropy counter IP.ec. This means, in the worst case, we need approximately 10,000 interrupts delivered to the guest in order to *guarantee* `/dev/urandom` will produce no repeated outputs (assuming, of course, that no reads to `/dev/random` occur).

To test this, we capture a snapshot of the legacy system while it idles. Upon resumption, a user process in Dom0 sends packets at various frequencies to the guest. We repeat VM resets 50 times for the frequencies shown in Figure 2.10. This table shows that with no intervention, it takes on average more than 50 seconds for the `/dev/urandom` pool UP to receive a single transfer from the input pool, and injecting interrupts from Dom0 significantly speed this up. However, we are unable to refresh the input pool in less than 3.6 s, due in part to the fact that during resumption there is a time window during which interrupts are delivered slowly (presumably because the hypervisor is busy doing resumption-related work).

Discussion. While both of our legacy-compatible countermeasures in this section provide some protections against the reset vulnerabilities in `/dev/(u)random`, we feel that they are at best stop-gap measures. The difficulty of dealing with this from Dom0 underscores the need to move to an improved RNG (namely, Whirlwind).

2.7 Related Work

Many high profile RNG failures have been reported over the years, including ones leading to: [50] showed attacks against the Secure Socket Layer (SSL) implementation of an early Netscape web browser; [5] demonstrates cheating at online poker; [41] showed insecure random values in Microsoft Windows; [98] found predictable host keys in Debian OpenSSL; [14] demonstrated jailbreaks against the Sony's PlayStation3; [53] factored RSA private keys generated on embedded systems; [59] predicted outputs in the OpenSSL RNG on Android; and [16] factored RSA private keys protecting digital IDs on government-issued smart cards in Taiwan.

Several previous papers analyzed the Linux `/dev/(u)random` RNG. [52] provided the first: they reverse-engineer the design of the RNG from the source code (attesting to its complexity!); highlight problems in the hashing steps (that were subsequently fixed and the version we analyze includes these fixes); and point out that in some constrained environments such as embedded systems or network routers there might be insufficient entropy provided to the RNG.

[53] show that embedded Linux systems suffer from a boot-time entropy hole which leads to exposure of cryptographic secret keys generated on affected devices. [74] look to fill this boot-time entropy hole by way of timing functions in kernel initialization. [49] perform an in-depth, empirical analysis of entropy transfers in Linux `/dev/(u)random`, and show that most consumers are in the kernel.

[39], building off earlier work by [8], suggest that the cryptographic extraction component of RNGs should be robust, meaning an RNG should guarantee entropy is collected no matter the rate of entropy in the input stream. They show that `/dev/(u)random` is not robust, but do not show attacks that would affect practice. Part of the Whirlwind design is inspired by their online hashing based extractor, though they use universal hash functions and we use cryptographic ones.

None of the above consider RNG performance in modern virtualized environments. We also do not know of any analyses of the GRI RNG before our work.

Turning to virtualized settings, [47] hypothesized that VM reset vulnerabilities may exist when reusing VM snapshots, and analyses by [85], uncovered actual vulnerabilities in user-level processes such as Apache mod_ssl that cache randomness in memory before use. In these settings, the user-level process never invoked `/dev/urandom` (or `/dev/random`) after VM resumption, and in particular they left as an open question whether system RNGs suffer from reset vulnerabilities as well. We answer this question, unfortunately, in the positive, suggesting that using `/dev/urandom` right before randomness use is not a valid countermeasure with the existing design, though it will be with Whirlwind.

[93] hypothesize that booting multiple times from the same VM image in an infrastructure-as-a-service (IaaS) setting such as Amazon's EC2 may enable attackers to predict `/dev/(u)random` RNG outputs that can lead to SSH host key compromises. Our analyses suggest that such an attack is infeasible for all uses of the Linux RNGs beyond the first during boot.

[96] point out the potential for a malicious hypervisor to snoop on the entropy pools of a guest VM. [57] investigate entropy pool poisoning attacks, where one guest VM in a cloud setting attempts to interfere with another's entropy pool by (say) sending interrupts at a known frequency to the guest. Theirs is a negative result, with their experiments showing that the attack fails. Our measurements corroborate this: even just a few bits of uncertainty about cycle counters leads to an unpredictable RNG state even in the current `/dev/(u)random` implementation. We also investigate using such interrupt injection as a defense.

Finally, we use CPU timing jitter as an entropy source as used in other systems, such as the haveged entropy daemon and the CPU jitter RNG, see [1, 76].

2.8 Discussion

In this work, we performed the first analysis of the security of the Linux system random number generators (RNGs) when operating in virtualized environments including Xen, VMware, and Amazon EC2. While our empirical measurements estimate that cycle counters in these settings (whether virtualized or not) provide a ready source of uncertainty from an attacker's point of view, deficiencies in the design of the `/dev/(u)random` RNG make it vulnerable to VM reset vulnerabilities which cause catastrophic reuse of internal state values when generating supposedly "random" outputs. Both the `/dev/(u)random` and kernel-only GRI RNGs also suffer from a small boot-time entropy hole in which the very first output from either is more predictable than it should be.

Our second main contribution is a new design for system RNGs called Whirlwind. It rectifies the problems of the existing Linux RNGs, while being simpler, faster, and using a sound cryptographic extraction process. We have implemented and tested Whirlwind in virtualized environments. Our results showed that Whirlwind enjoys performance equal (and sometimes even better) than the previous RNG.

3 PASSWORD-BASED AUTHENTICATION

Internet services commonly use passwords to authenticate remote users. It is by far the most prevalent technique presently in use, despite the burden it places on individuals that routinely have more than one-hundred distinct accounts[1]. Internet-scale services may have millions or billions of users, making password databases themselves desirable targets for attackers. Indeed, there have been a number of high-profile disclosures of popular internet services suffering password database compromises. These include thefts of 117 *million* passwords from the service LinkedIn ([81]), and more than 1 *billion* passwords stolen from Yahoo ([95]), among many other reported thefts.

Many internet-scale services *harden* user passwords before storing them. Hardening is accomplished by applying a pseudorandom function (PRF) to the input and storing the output. A strong PRF has pre-image resistance, meaning that it is not efficient for an attacker to invert the PRF–that is, using the output to compute its corresponding input. However, a password hardened this way can be recovered with a dictionary attack: an attacker iterates through a so-called dictionary of likely passwords, applies the PRF, and checks for a match. A dictionary attack only works when password complexity is sufficiently low—but low complexity is typical for human-generated passwords.

A cryptographically keyed PRF, like HMAC ([60]), prohibits dictionary attacks provided the deployment prevents the attacker from compromising both the hardened passwords *and* the PRF key. There is some limited evidence of keyed PRFs being used in practice, see [75, 3]. One major limitation of existing keyed PRFs like HMAC is that key rotation is challenging. If a service wants to rotate the PRF key on a regular basis, the

[1]Individuals respond to this burden by using password management software or simply reusing common passwords across accounts. The latter technique is (unfortunately) quite common.

service must either require that individuals provide their original password for re-hardening under the new key, or else the service must "wrap" the existing hardened password with a new application of the PRF. Both techniques are infeasible given either a large number of users or a large number of key rotations.

Threats. As mentioned previously, we address the common threat to password-based authentication systems: attackers may access the collection of hardened passwords, exfiltrate the collection, then perform extensive offline attacks. This type of threat model is commonly called the snapshot compromise. That is, an attacker is assumed to have access to an entire snapshot of the password database (or authentication server) at a given point in time. This snapshot would include the full state of the authentication server, including any secrets held on the authentication server, either on-disk, in processing code, or in memory.

In multiple machine configurations, as in Pythia, we assume that simultaneous compromise of all machines is unlikely whenever there is a reasonable argument that these machines are in distinct security domains (e.g. they perform different functions or are administered by distinct owners). We assume eventual compromise of all machines is likely however.

Notably, we ignore denial of service attacks, either by overwhelming services with floods of network traffic or requests, or locking individual accounts with spurious authentication attempts. We note that all password-based authentication techniques deal with these threats currently and there is ample empirical evidence that the common mitigation techniques are effective.

Password-hardening service. To improve on the state-of-the-art, we present a password-hardening *service* that is useful in a number of deployment scenarios. This service is called Pythia and is built on a novel cryptographic construction described later. Pythia hardens passwords using a keyed PRF, but also enables rate-limiting of guesses (to defeat online dictionary attacks), and rekeying of the PRF for both proactive and reactive response to compromises. A few deployment scenarios for Pythia:

- *Enterprise deployment:* A single enterprise can deploy Pythia as an internal authentication service, giving query access only to front-end systems that they control (e.g. a web server).
- *Public cloud service:* A public cloud provider can deploy Pythia as a multi-tenant service for their customers. Multi-tenant here means that different customers query the same Pythia service, and the cloud provider manages the service, ensemble pre-key table, etc. This enables organizations to obtain the benefits of using Pythia for internet-service (e.g., web servers) while leaving management of the Pythia to experts.
- *Public internet service:* Pythia can be deployed as a service to individual users that could access the service from anywhere on the internet. This opens up a variety of applications for individuals: device and file encryption via hardened passwords, generating a strong master key for encrypting password vaults, or hardening bitcoin "brainwallets" as described in Section 3.5.

In Section 3.3 we describe the high level API for Pythia as a multi-tenant password-hardening service.

Partially-oblivious PRFs. We introduce *partially oblivious pseudorandom functions* (PO-PRFs) in Section 3.2 to achieve a primitive that supports batch key rotation. We give a PO-PRF protocol in the random oracle model (ROM) similar to the core of the identity-based non-interactive key exchange protocol of [90]. This same construction was also considered as a left-or-right constrained PRF by [26]. That said, the functionality achieved by our PO-PRF is distinct from these prior works and new security analyses are required.

Performance and scalability. We demonstrate that despite relying on computationally-intensive bilinear pairings and requiring communication over the network, a client using Pythia enjoys low-latency queries. We implemented a prototype service and report on it's performance in Section 3.4. We show that Pythia queries require only 5 ms in a local area network setting which is as fast or faster than state-of-the-art password-hardening techniques like scrypt or PBKDF2. Further, our prototype is architected to scale up to service large numbers of simultaneous requests using the standard techniques of horizontal scaling and load balancing. We show that using only modest storage (20 GB) a Pythia server can support 100M clients (web servers) and an arbitrary number of end-users. All together, Pythia enables an extraordinary security gain over the state-of-the-art that can be accomplished on commodity hardware.

Applications. Pythia lends itself naturally to a number of password-protected applications, namely file encryption and access control. Use of Pythia in these settings is straightforward and so we omit detailed discussion, but we do describe an interesting application in Section 3.5: a client hardens a type of password-protected virtual-currency account called a *brainwallet* (see [27]). Use of Pythia in this application prevents offline brute-force attacks of the type that have been common in Bitcoin.

Threshold security. Using Pythia to harden password presents an increase in security, but ties the availability and durability of its clients to the availability and durability of the Pythia service. To alleviate this, we describe a threshold security technique in Section 3.6. Briefly, a threshold security system permits a client to engage with multiple Pythia servers, spread secrets among them, and as long as some predetermined subset of those servers are accessible (and remain secure), the client maintains availability, durability, and security.

3.1 Overview and Challenges

We now give a high-level overview of Pythia, the motivations for its features, what prior approaches we investigated, and the threat models we assume. First we fix some terminology and a high-level conceptual view of what a PRF service would ideally provide. The service is provisioned with a master secret key msk. This will be used to build a tree that represents derived sub-keys and, finally, output values. See Figure 3.1, which depicts an example derivation tree associated with Pythia as well as which portions of the tree are held by the server (within the large box) and which are held by the client (the leaves). Keys of various kinds are denoted by circles and inputs by squares.

From the msk we derive a number of *ensemble keys*. Each ensemble key is used by a client for a set of related PRF invocations — the ensemble keys give rise to isolated PRF instances. We label each ensemble key in the diagram by $\mathsf{K}[w]$. Here w indicates a client-chosen *ensemble selector*. An *ensemble pre-key* $\mathsf{K}[w]$ is a large random value chosen and held by the server. Together, msk and $\mathsf{K}[w]$ are used to derive the ensemble key $\mathsf{k}_w =$ HMAC$(\mathsf{msk}, \mathsf{K}[w])$. A table is necessary to support cryptographic erasure of (or updates to) individual ensemble keys, which amounts to deleting (or updating) a table entry.

Each ensemble key can be used to obtain PRF outputs of the form $\mathsf{F}_{\mathsf{k}_w}(\mathsf{t}, \mathsf{m})$ where F is a (to-be-defined) PRF keyed by k_w, and the input is split into two parts. We call t a *tweak* following [63] and m the message. Looking ahead t will be made public to Pythia while m will be private. This is indicated by the shading of the PRF output boxes in the figure.

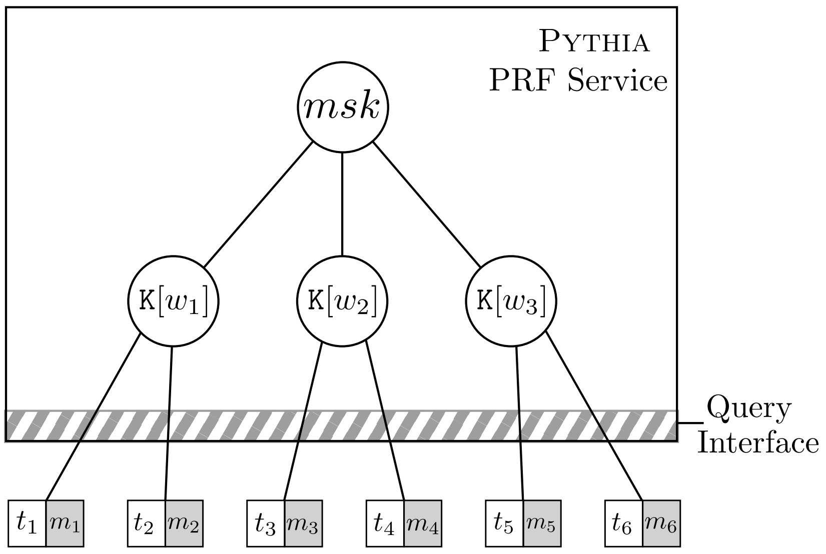


Figure 3.1: Diagram of PRF derivations enabled by PYTHIA. Everything inside the large box is operated by the server, which only learns tweaks and not the shaded messages.

Deployment scenarios. To motivate our design choices and security goals, we relay several envisioned deployment scenarios for PYTHIA.

Enterprise deployment: A single enterprise can deploy PYTHIA internally, giving query access only to other systems they control. A typical setup is that PYTHIA fields queries from web servers and other public-facing systems that are, unfortunately, at high risk of compromise. PRF queries to PYTHIA harden values stored on these vulnerable servers. This is particularly suited to storing check-values for passwords or other low-entropy authentication tokens, where one can store $\mathsf{F}_{\mathsf{k}_w}(\mathsf{t}, \mathsf{m})$ where t is a randomly chosen, per-user identifier (a salt) and m is the low-entropy password or authentication token. Here w can be distinct for each server using PYTHIA.

Public cloud service: A public cloud such as Amazon EC2, Google Compute Engine, or Microsoft Azure can deploy PYTHIA as an internal, multi-tenant service for their customers. Multi-tenant here means that different customers query the same Pythia service, and the cloud provider manages the service, ensemble pre-key table, etc. This enables smaller organizations to obtain the benefits of using Pythia for other cloud properties (e.g., web servers running on virtual machine instances) while leaving management of Pythia itself to experts.

Public Internet service: One can take the public cloud service deployment to the extreme and run Pythia instances that can be used from anywhere on the Internet. This raises additional performance concerns, as one cannot rely on fast intra-datacenter network latencies (sub-millisecond) but rather on wide-area latencies (tens of milliseconds). The benefit is that Pythia could then be used by arbitrary web clients, for example we will explore this scenario in the context of hardening brainwallets via Pythia.

One could tailor a PRF service to each of these settings, however it is better to design a single, application-agnostic service that supports all of these settings simultaneously. A single design permits reuse of open-source implementations; standardized, secure-by-default configurations; and simplifies the landscape of PRF services.

Security and functionality goals. Providing a single suitable design requires balancing a number of security and functionality goals. The most obvious requirements are for a service that: provides low-latency protocols (i.e., single round-trip and amenable for implementation as simple web interfaces); scales to hundreds of millions of ensembles; and produces outputs indistinguishable from random values even when adversaries can query the service. To this list of basic requirements we add:

- *Message privacy*: The PRF service must learn nothing about m. Message privacy supports clients that require sensitive values such as passwords to remain private even if the service is compromised, or to promote psychological acceptability in the case that a separate organization (e.g., a cloud provider) manages the service.

- *Tweak visibility*: The server must learn tweak t to permit fine-grained rate-limiting of requests.[2] In the password storage example, a distinct tweak is assigned to each user account, allowing the service to detect and limit guessing attempts against individual user accounts.

- *Verifiability:* A client must be able to verify that a PRF service has correctly computed $\mathsf{F}_{\mathsf{k}_w}$ for a ensemble selector w and tweak/message pair t, m. This ensures, after first use of an ensemble by a client, that a subsequently compromised server cannot surreptitiously reply to PRF queries with incorrect values.[3]

- *Client-requested ensemble key rotations:* A client must be permitted to request a rotation of its ensemble pre-key $\mathsf{K}[w]$ to a new one $\widehat{\mathsf{K}[w]}$. The server must be able to provide an update token Δ_w to roll forward PRF outputs under $\mathsf{K}[w]$ to become PRF outputs under $\widehat{\mathsf{K}[w]}$, meaning that the PRF is *key-updatable* with respect to ensemble keys. Additionally, Δ_w must be *compact*, i.e., constant in the number of PRF invocations already performed under w. Clients can mandate that rotation requests be authenticated (to prevent malicious key deletion). A client must additionally be able to *transfer* an ensemble from one selector w to another selector w'.

- *Master secret rotations:* The server must be able to rotate the master secret key msk with minimal impact on clients. Specifically, the PRF must be key-updatable with respect to the master secret key msk so that PRF outputs under msk can be rolled forward to a new master secret $\widehat{\mathsf{msk}}$. When such a rotation occurs, the server must provide a compact update token δ_w for each ensemble w.

[2]In principle, the server need only be able to link requests involving the same t, not learn t. Explicit presentation of t is the simplest mechanism that satisfies this requirement.

[3]This matters, for example, if an attacker compromises the communication channel but not the server's secrets (msk and $\mathsf{K}[w]$). Such an attacker must not be able to convince the client that arbitrary or incorrect values are correct.

- *Forward security:* Rotation of an ensemble key or master secret key results in complete erasure of the old key and the update token.

Two sets of challenges arise in designing PYTHIA. The first is cryptographic. It turns out that the combination of requirements above are not satisfied by any existing protocols we could find. Ultimately we realized a new type of cryptographic primitive was needed that proves to be a slight variant of oblivious PRFs and blind signatures. We discuss the new primitive, and our efficient protocol realizing it, in the next section. The second set of challenges surrounds building a full-featured service that provides the core cryptographic protocol, which we treat in Section 3.3.

3.2 Partially-oblivious PRFs

We introduce the notion of a (verifiable) partially-oblivious PRF. This is a two-party protocol that allows the secure computation of $F_{k_w}(t, m)$, where F is a PRF with server-held key k_w and t, m are the input values. The client can verify the correctness of $F_{k_w}(t, m)$ relative to a public key associated to k_w. Following our terminology, t is a tweak and m is a message. We say the PRF is partially oblivious because t is revealed to the server, but m is hidden from the server.

Partially oblivious PRFs are closely related to, but distinct from, a number of existing primitives. A standard oblivious PRF like that of [46], or its verifiable version from [55], would hide both t and m, but masking both prevents granular rate limiting by the server. Partially blind signatures like those of [2] allow a client to obtain a signature on a similarly partially blinded input, but these signatures are randomized and the analysis is only for unforgeability which is insufficient for security in all of our applications.

We provide more comparisons with related work in Section 3.7 and a formal definition of this primitive appears in [43]. Here we will present the protocol that suffices for PYTHIA. It uses an admissible bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ over groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of prime order q, and a pair of hash functions $H_1 : \{0,1\}^* \to \mathbb{G}_1$ and $H_2 : \{0,1\}^* \to \mathbb{G}_2$ (that we will model as random oracles). More details on pairings are provided by [43]. A secret key k_w is an element of $\mathbb{Z}_p$. The PRF F that the protocol computes is:

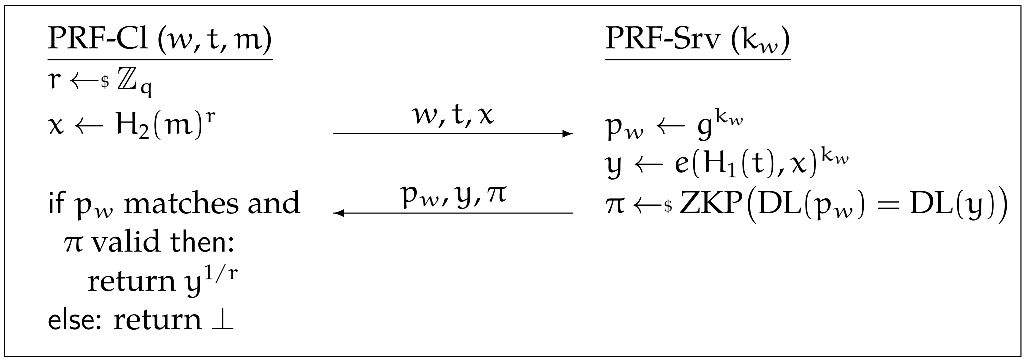
$$F_{k_w}(t, m) = e\big(H_1(t), H_2(m)\big)^{k_w} .$$

This construction coincides with the [90] construction for non-interactive identity-based key exchange, where t and m would be different identities and k_w a secret held by a trusted key authority. Likewise, this construction is equivalent to the left-or-right constrained PRF of [26]. The contexts of these prior works are distinct from ours and our analyses will necessarily be different, but we note that all three settings similarly exploit the algebraic structure of the bilinear pairing. See Section 3.7 for further discussion of related work.

The client-server protocol computes $F_{k_w}(t, m)$ in a partially-oblivious manner and is shown in Figure 3.2. There we let g be a generator of $\mathbb{G}_1$. We now explain how the protocol achieves our requirements described in the last section.

Blinding the message: In our protocol, the client blinds the message m, hiding it from the server, by raising it to a randomly selected exponent $r \leftarrow_{\$} \mathbb{Z}_q$. As $e\big(H_1(t), H_2(m)^r\big) = e\big(H_1(t), H_2(m)\big)^r$, the client can unblind the output y of PRF-Srv by raising it to $1/r$. This protocol hides m unconditionally, as $H_2(m)^r$ is a uniformly random element of $\mathbb{G}_2$.

Verifiability: The protocol enables a client to verify that the output of PRF-Srv is correct, assuming the client has previously stored p_w. The



```
PRF-Cl $(w, t, m)$                                              PRF-Srv $(k_w)$
$r \leftarrow_{\$} \mathbb{Z}_q$
$x \leftarrow H_2(m)^r$            $\xrightarrow{w, t, x}$        $p_w \leftarrow g^{k_w}$
                                                                $y \leftarrow e(H_1(t), x)^{k_w}$
if $p_w$ matches and               $\xleftarrow{p_w, y, \pi}$     $\pi \leftarrow_{\$} \mathrm{ZKP}\big(\mathrm{DL}(p_w) = \mathrm{DL}(y)\big)$
  $\pi$ valid then:
    return $y^{1/r}$
else: return $\perp$
```

Figure 3.2: The partially-oblivious PRF protocol used in Pythia. By interacting with a server, the client computes the value $F_{k_w}(t, m)$. The value π is a non-interactive zero-knowledge proof that the indicated discrete logarithms match. The client also verifies that the value p_w is the same value provided in previous interactions with the server.

server accompanies the output y of the PRF with a zero-knowledge proof π of correctness.

Specifically, for a public key $p_w = g^{k_w}$, where g is a generator of $\mathbb{G}_1$, the server proves $\mathrm{DL}_g(p_w) = \mathrm{DL}_{\tilde{x}}(y)$. Standard techniques (see, e.g., [31]) permit efficient ZK proofs of this kind in the random oracle model. [4] The notable computational costs for the server are one pairing and one exponentiation in $\mathbb{G}_T$; for the client, one pairing and two exponentiations in $\mathbb{G}_T$. [5]

Efficient key updates: The server can quickly and easily update the key k_w for a given ensemble selector w by replacing the table entry $s = \mathtt{K}[w]$ with a new, randomly selected value s', thereby changing $k_w = \mathrm{HMAC}(msk, s)$ to $k'_w = \mathrm{HMAC}(msk, s')$. It can then transmit to the client an update token of the form $\Delta_w = k'_w / k_w \in \mathbb{Z}_q$.

[4] Some details: The prover picks $v \leftarrow_{\$} \mathbb{Z}_q$ and then computes $t_1 = g^v$ and $t_2 = \tilde{x}^v$ and $c \leftarrow H_3(g, p_w, \tilde{x}, y, t_1, t_2)$. Let $u = v - c \cdot k$. The proof is $\pi = (c, u)$. The verifier computes $t'_1 = g^u \cdot p_w^c$ and $t'_2 = \tilde{x}^u y^c$. It outputs true if $c = H_3(g, p_w, \tilde{x}, y, t'_1, t'_2)$.

[5] The client's pairing can be pre-computed while waiting for the server's reply.

The client can update any stored PRF value:

$$F_{k_w}(t, m) = e\big(H_1(t), H_2(m)\big)^{k_w}$$

by raising it to Δ_w; it is easy to see that:

$$F_{k_w}(t, m)^{\Delta_w} = F_{k'_w}(t, m).$$

The server can use the same mechanism to update msk, which requires generating a new update token for each w and pushing these tokens to clients as needed.

Unblinded variants. For deployments where obliviousness of messages is unnecessary, we can use a faster, unblinded variant of the Pythia protocol that dispenses with pairings.

The only changes are that the client sends m to the server, there is no unblinding of the server's response, and, instead of computing

$$\tilde{x} \leftarrow e(H_1(t), x)$$

the server computes

$$\tilde{x} \leftarrow H_3(t \| m) \;.$$

All group operations in this unblinded variant are over a standard elliptic curve group $\mathbb{G} = \langle g \rangle$ of order q and we use a hash function $H_3 \colon \{0,1\}^* \to \mathbb{G}$.

An alternative unblinded construction would be to have the server apply the [25] short signatures to the client-submitted $t \| m$; verification of correctness can be done using the signature verification routine, and we can thereby avoid ZKPs. This BLS variant may save a small amount of bandwidth.

These unblinded variants provide the same services (verifiability and

Command	**Description**
$\mathsf{Init}(w\ [,\mathtt{options}])$	Create table entry $\mathtt{K}[w]$ (for ensemble key k_w)
$\mathsf{Eval}(w, t, m)$	Return PRF output $F_{k_w}(t, m)$
$\mathsf{Reset}(w,\ \mathtt{authtoken})$	Update $\mathtt{K}[w]$ (and thus k_w); return update token Δ_w
$\mathsf{GetAuth}(w)$	Send one-time authentication token $\mathtt{authtoken}$ to client

Figure 3.3: The core PYTHIA API.

efficient key updates) and security with the obvious exception of the secrecy of the message m. In some deployment contexts an unblinded protocol may be sufficient, for example when the client can maintain state and submit a salted hash m instead of m directly. In this context, the salt should be held as a secret on the client and never sent to the server.

3.3 The PYTHIA Service Design

Figure 3.3 gives the core API exposed by PYTHIA to a client. We first describe its function in terms of the lifecycle of an ensemble key. We later describe additional API elements that are specific to a particular deployment or application class.

Ensemble initialization. To begin using the PYTHIA service, a client creates an ensemble key for selector w by invoking $\mathsf{Init}(w\ [,\mathtt{options}])$. PYTHIA generates a fresh, random table entry $\mathtt{K}[w]$. Recall that ensemble key $k_w = \text{HMAC}(msk, \mathtt{K}[w])$. So Init creates k_w as a byproduct.

Ideally, w should be an unguessable byte string. (An easily guessed one may allow attackers to squat on a key selector, thereby mounting a denial-of-service (DoS) attack.) For some applications, as we explain below, this isn't always possible. If an ensemble key for w already exists, then the Pythia service returns an error to the client. Otherwise, the client receives a message signifying that initialization is successful.

PRF evaluation. To obtain a PRF value, a client can perform an evaluation query $\mathsf{Eval}(w, t, m)$, which returns $F_{k_w}(t, m)$. Here t is a tweak and m is a message. To compute the PRF output, the client and server perform a one-round cryptographic protocol (meaning a single message from client to server, and one message back). We present details in Section 3.2, but remind the reader that t is visible to the server in the client-server protocol invoked by Eval, while m is blinded.

The server rate-limits requests based on the tweak t, and can also raise an alert if the rate limit is exceeded.

Ensemble-key reset. A client can request that an ensemble key k_w be reset by invoking $\mathsf{Reset}(w)$. This reset is accomplished by overwriting $\mathtt{K}[w]$ with a fresh, random value. The name service returns a compact (e.g., 256-bit) update token Δ_w that the client may use to update all PRF outputs for the ensemble. It stores this token locally, encrypted under a public key specified by the client, as explained below.

Note that reset *results in erasure of the old value of* k_w. Thus a client that wishes to delete an ensemble key k_w permanently at the end of its lifecycle can do so with a Reset call.

Reset is an authenticated call, and thus requires the following capability.

Authentication. To authenticate itself for API calls, the client must first invoke GetAuth, which has the server transmit an (encrypted) authentication token $\mathit{authtoken}$ to the client out-of-band (e.g. via email). The token expires after a period of time determined by a configuration parameter in Pythia. Of course, in some deployments one may want authentication to be performed in other ways, such as tokens dispensed by administrators

Selector option	Description
Email	Contact email for selector
Resettable	Whether client-requested rotations allowed
Limit	Establish rate-limit per t
Time-out	Date/time to delete k_w
Public-key	Key under which to encrypt and store update and authentication tokens
Alerts	Whether to email contact upon rate limit violation

Figure 3.4: Optional settings for establishing key selectors in Pythia.

Command	Description
$\mathsf{Transfer}(w, w' \,[, \mathit{options}])$	Creates new ensemble w'; outputs update token $\Delta_{w \to w'}$; resets k_w
$\mathsf{SendTokens}(w, \mathit{authtoken})$	Sends stored update tokens to client
$\mathsf{PurgeTokens}(w, \mathit{authtoken})$	Purges all stored update tokens for ensemble w

Figure 3.5: Additional Pythia API

(for enterprise settings) or simply given out on a first-come-first-serve basis for each ensemble identifier (for public Internet services).

Pythia also offers some additional API calls, given in Figure 3.5, which we now describe.

Additional Pythia API Details

Many Pythia-dependent services benefit from or require additional API features beyond the primary ones discussed above. We now detail these features and there usage.

Key-management options. The client can specify a number of options in the call Init regarding management of the ensemble key k_w. The client can provide a contact email address to which alerts and authentication tokens may be sent. (If no e-mail is given, no API calls requiring authentication are permitted and no alerts are provided. Other versions of PYTHIA may support other authentication and alerting methods.)

The client can specify whether k_w should be resettable (default is "yes"). The client can specify a limit on the total number of $\mathsf{F}_{\mathsf{k}_w}$ queries that should be allowed before resetting $\mathsf{K}[w]$ (default is unlimited) and/or an absolute expiration date and time in UTC at which point $\mathsf{K}[w]$ is deleted (default is no time-out). Either of these options overrides the resettable flag. The client can specify a public key $\mathsf{pk}_{\mathsf{cl}}$ for a public-key encryption scheme under which to encrypt authentication tokens and update tokens (for Reset, Transfer, as described below, and for master secret key rotations). Finally, the client can request that alerts be sent to the contact email address in the case of rate limit violations. This option is ignored if no contact email is provided. The options are summarized in Figure 3.4.

Ensemble transfer. A client can create a new ensemble w' (with the same options as in Init) while receiving an update token that allows PRF outputs under ensemble w to be rolled forward to w'. This is useful for importing a password database to a new server. The PYTHIA service returns an update token $\Delta_{w \to w'}$ for this purpose and stores it encrypted under $\mathsf{pk}_{\mathsf{cl}}$. For the case $w' = w$, this call also allows option updates on an existing ensemble w.

Update-token handling. The PYTHIA service stores update tokens encrypted under $\mathsf{pk}_{\mathsf{cl}}$, with accompanying timestamps for versioning. The API call SendTokens causes these to be e-mailed to the client, while the API call PurgeTokens causes update-token ciphertexts to be deleted from PYTHIA.

Note that once an update token is deleted, old PRF values to which the token was not applied become cryptographically erased — they become random values unrelated to any messages. A client can therefore delete the key associated with an ensemble by calling Reset and PurgeTokens.

Master secret rotations. Pythia can also rotate its master secret key msk to a new key msk'. Recall that ensemble keys are computed as $\mathsf{k}_w = \text{HMAC}(\mathsf{msk}, \mathsf{K}[w])$, so rotation of msk results in rotation of all ensemble keys. To rotate to a new msk', the server computes k_w for all ensembles w with entries in K, and stores δ_w encrypted under $\mathsf{pk}_{\mathsf{cl}}$. If no encryption key is set, then the token is stored in the clear. This is a forward-security issue while it remains, but only for that particular key ensemble. At this point msk is safe to delete. Clients can be informed of the key rotation via e-mail.

Subsequent SendTokens requests will return the resulting update token, along with any other stored update tokens for the ensemble. If multiple rotations occur between client requests, then these can be aggregated in the stored update token for each ensemble. This is trivial if they are stored in the clear (just multiply the new token against the old) and also works if they are encrypted with an appropriately homomorphic encryption scheme such as ElGamal ([42]).

3.4 Implementation

We implemented a prototype Pythia PRF service as a web application accessed over HTTPS. All requests are first handled by an nginx web server with uWsgi as the application server gateway that relays requests to a Django back-end. The PRF-Srv functionality is implemented as a Django module written in Python. Storage for the server's key table and rate-limiting information is done in MongoDB.

We use the Relic cryptographic library of [4] (written in C) with our own Python wrapper. We use [10] 254-bit prime order curves (BN-254). These curves provide approximately 128-bits of security.

In our experiments the service is run on a single (virtual) machine, but our software stack permits components (web server, application sever, database) to be distributed among multiple machines with updates to configuration files.

For the purpose of comparison, we implemented three variants of the Pythia service. The first two are the unblinded protocols described in Section 3.2. In these two schemes, the client sends m in the clear (possibly hashed with a secret salt value first) and the server replies with $y = H_1(t\|m)^k$. In the first scheme, denoted UNB, the server provides $p = g_1^k$ and a zero-knowledge proof where g_1 is a generator of $\mathbb{G}_1$. The second scheme, denoted BLS, uses a BLS signature for verification. The server provides $p = g_2^k$ where g_2 is a generator of $\mathbb{G}_2$ and the client verifies the response by computing and comparing the values: $e(y, g_2) = e(H_1(t\|m), p)$.

Our partially-oblivious scheme is denoted PO.

For the evaluation below we use a Python client implementing PRF-Cl for all three schemes using the same libraries indicated above for the server and httplib2 to perform HTTPS requests.

Performance

For performance and scalability evaluation we hosted our Pythia server implementation on Amazon's Elastic Compute Cloud (EC2) using a c4.xlarge instance which provides 8 virtual CPUs (Intel Xeon third generation, 2.9GHz), 15 GB of main memory, and solid state storage. The web server, nginx, was configured with basic settings recommended for production deployment including one worker process per CPU.

Group	Time (μs)		
	Group Op	Exp	Hashing
$\mathbb{G}_1$	5.7	175	77
$\mathbb{G}_2$	6.7	572	210
$\mathbb{G}_T$	9.8	1145	–
pairing operation (e) takes 1005 μs			

Table 3.1: Time taken by each operation in BN-254 groups. Hashing times are for 64-byte inputs.

Latency. We measured client query latency for each protocol using two clients: one within the same Amazon Web Service (AWS) availability zone (also c4.xlarge) and one hosted at the University of Wisconsin–Madison with an Intel Core i7 CPU (3.4 GHz). We refer to the first as the LAN (local-area network) setting and the second as the WAN (wide-area network) setting. In the LAN settings we used the AWS internal IP address. All queries were made over TLS and measurements include the time required for clients to blind messages and unblind results (PO), as well as verify proofs provided by the server (unless indicated otherwise). All machines used for evaluation were running Ubuntu 14.04.

Microbenchmarks for group operations appear in Table 3.1. We decompose a single PRF evaluation into components and show the timing of each in Table 3.2. All results are mean values computed over 10,000 operations. These values were captured on an EC2 c4.xlarge instance using the Python profiling library line_profiler. The most expensive operations, by a large margin, are exponentiation in $\mathbb{G}_t$ and the pairing operation. By extension, PO sign, prove, and verify operations become expensive.

We measured latencies averaged over 1,000 PRF requests (with 100 warmup requests) for each scheme and the results appear in Figure 3.3. Computation time dominates in the LAN setting due to the almost negligible network latency. The WAN case with cold connections (no HTTP KeepAlive) pays a performance penalty due to the four round-trips required to set up a new TCP and TLS connection. While even 400 ms

Server Op	Time (ms)		
Table	1.2		
Rate-limit	0.9		
	UNB	**BLS**	**PO**
Sign	0.3	0.3	1.5
Prove	0.5	0.3	2.5

Client Op	UNB	BLS	PO
Blind	-	-	0.3
Unblind	-	-	1.2
Verify	0.9	2.0	4.0

Table 3.2: Computation time for major operations to perform a PRF evaluation divided into **(Left)** server-side and **(Right)** client-side operations. Table retrieves K[w] from database; Rate-limit updates rate-limiting record in database; and Sign generates the PRF output.

	Latency (ms)					
	LAN			**WAN**		
Scheme	Cold	Hot	No π	Cold	Hot	No π
UNB	7.0	3.8	2.4	389	82	80
BLS	7.9	4.9	2.4	392	85	80
PO	14.9	11.8	5.2	403	96	84
RTT ping	0.1			82		

Table 3.3: Average latency to complete a PRF-Cl with client-server communication over HTTPS. LAN: client and server in the same EC2 availability zone. WAN: server in EC2 US-West (California) and client in Madison, WI. Hot connections made with HTTP KeepAlive enabled; cold connections with KeepAlive disabled. No π: KeepAlive enabled; prove and verify computations are skipped.

latencies are not prohibitive in our applications, straightforward engineering improvements would vastly improve WAN timing: using TLS session resumption, using lower-latency secure protocol like QUIC ([89]), or even switching to a custom UDP protocol (for an example one for oblivious PRFs, see [12]).

Throughput. We used the distributed load testing tool autobench to measure maximum throughput for each scheme. We compare to a static page containing a typical PRF response served over HTTPS as a baseline. We used two clients in the same AWS region as the server. All connections

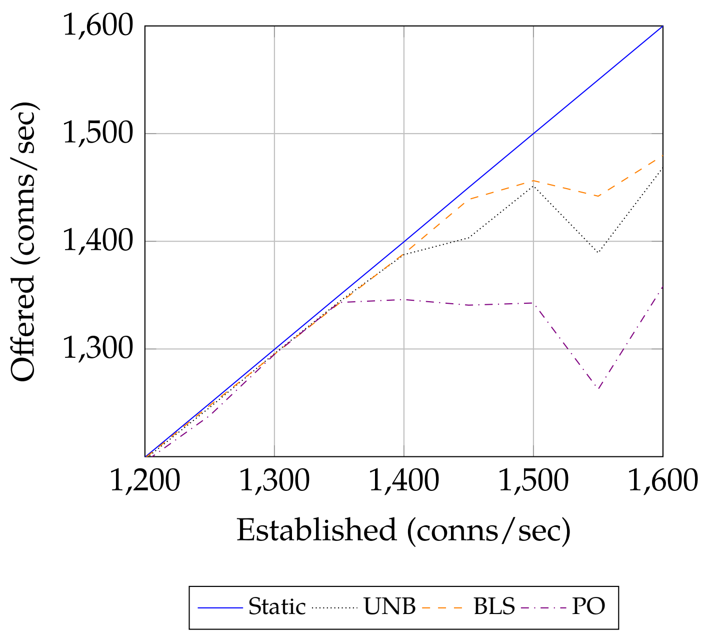


Figure 3.6: Throughput of PRF-Srv requests and a static page request over HTTPS measured using two clients and a server hosted in the same EC2 availability zone.

were cold: no TLS session resumption or HTTP KeepAlive. Results appear in Figure 3.6.

The maximum throughput for a static page is 2,200 connections per second (cps); UNB and BLS 1,400 cps; and PO 1,350 cps. Thus our Pythia implementation can handle a large number of clients on a single EC2 instance. If needed, the implementation can be scaled with standard techniques (e.g., a larger number of web servers and application servers on the front-end with a distributed key-value store on the back-end).

Storage. Our implementation stores all ensemble pre-key table (K) entries and rate-limiting information in MongoDB. A table entry is two 32 byte values: a SHA-256 hash of the ensemble selector w and its associated value $\mathtt{K}[w]$. In MongoDB the average storage size is 195 bytes per entry (measured as the average of 100K entries), including database overheads and indexes. This implementation scales easily to 100 M clients with under 20 GB of storage.

To rate-limit queries, our implementation stores tweak values along with a counter and a timestamp (to expire old entries) in MongoDB. Tweak values are also hashed using SHA-256 which ensures entries are of constant length. In our implementation each distinct tweak requires an average of 144 bytes per entry (including overheads and indexes). Note however that rate limiting entries are purged periodically as counts are only required for one rate-limiting period. Our implementation imposes rate-limits at hour granularity. Assuming a maximum throughput of 2,000 requests per second, rate-limiting storage never exceeds 1 GB.

All told, with only 20 GB stored data, Pythia can serve over 100 M clients and perform rate-limiting at hour granularity. Thus fielding a database for Pythia can be accomplished on commodity hardware.

3.5 Hardened Brainwallets

Brainwallets are a common but dangerous way to secure accounts in the popular cryptocurrency Bitcoin, as well as in less popular cryptocurrencies such as Litecoin. Here we describe how the Pythia service can be used directly as a means to harden brainwallets. This application showcases the ease with which a wide variety of applications can be engineered around Pythia.

How brainwallets work. Every Bitcoin account has an associated private–public key pair $(\mathsf{sk}, \mathsf{pk})$. The private key sk is used to produce digital (ECDSA) signatures that authorize payments from the account. The public key pk permits verification of these signatures. It also acts as an account identifier; a Bitcoin address is derived by hashing pk (under SHA-256 and RIPEMD-160) and encoding it (in base 58, with a check value).

Knowledge of the private key sk equates with control of the account. If a user loses a private key, she therefore loses control over her account. For example, if a high entropy key sk is stored exclusively on a device such as a mobile phone or laptop, and the device is seized or physically destroyed, the account assets become irrecoverable.

Brainwallets offer an attractive remedy for such physical risks of key loss. A brainwallet is simply a password or passphrase P memorized by a Bitcoin account holder. The private key sk is generated directly from P. Thus the user's memory serves as the only instrument needed to authorize access to the account.

In more detail, the passphrase is typically hashed using SHA-256 to obtain a 256-bit string $\mathsf{sk} = \text{SHA-256}(\mathsf{P})$. Bitcoin employs ECDSA signatures on the `secp256k1` elliptic curve; with high probability ($\approx 1 - 2^{-126}$), sk is less than the group order, and a valid ECDSA private key.[6]

Since a brainwallet employs only P as a secret, and does not necessarily use any additional security measures, an attacker that guesses P can seize control of a user's account. Account addresses are posted publicly in the Bitcoin system (in the "blockchain"), an attacker can easily confirm a correct guess. Brainwallets are thus particularly vulnerable to brute-force, offline guessing attacks.

A Pythia-hardened brainwallet

Pythia offers a simple, powerful means of protecting brainwallets against offline attack. Hardening P in the same manner as an ordinary password yields a strong key $\tilde{\mathsf{P}}$ that can serve in lieu of P to derive sk.

[6]Some websites employ stronger key derivation functions. For example, WrapWallet by [61] derives sk from an XOR of each of PBKDF2 and scrypt applied to P and permits use of a user-supplied salt.

To use Pythia, a user chooses a unique identifier id, e.g., her e-mail address, an account identifier acct, and a passphrase P. The identifier acct might be used to distinguish among Bitcoin accounts for users who wish to use the same password for multiple wallets. The client then sends $(w = \mathsf{id}, \mathsf{t} = \mathsf{id}\|\mathsf{acct}, \mathsf{m} = \mathsf{P})$ to the Pythia service to obtain the hardened value $\mathsf{F}_{\mathsf{k}_w}(\mathsf{t}, \mathsf{m}) = \tilde{\mathsf{P}}$. Here, id is used both as an account identifier and as part of the salt. Message privacy in Pythia ensures that the service learns nothing about P. Then $\tilde{\mathsf{P}}$ is hashed with SHA-256 to yield sk. The corresponding public key pk and address are generated in the standard way from sk as described in [18].

Pythia forces a would-be brainwallet attacker to mount an online attack to compromise an account. Not only is an online attack much slower, but it may be rate-limited by Pythia and detected and flagged. As the Pythia service derives $\tilde{\mathsf{P}}$ using a user-specific key, it additionally prevents an attacker from mounting a dictionary attack against multiple accounts. While in the conventional brainwallet setting, two users who make use of the same secret P will end up controlling the same account, Pythia ensures that the same password P produces distinct per-user key pairs.

Should an attacker compromise the Pythia service and steal msk and K, the attacker must still perform an offline brute-force attack against the user's brainwallet. So in the worst case, a user obtains security with Pythia at least as good as without it.

Additional security issues. A few subtle security issues deserve brief discussion:

- *Stronger KDFs:* To protect against brute-force attack in the event of Pythia compromise, a resource-intensive key-derivation function may be desirable, as is normally used in password databases. This can be achieved by replacing the SHA-256 hash of $\tilde{\mathsf{P}}$ above with an appropriate KDF computation.

- *Denial-of-service:* By performing rate-limiting, Pythia creates the risk of targeted denial-of-service attacks against Bitcoin users. As Bitcoin is pseudonymous, use of an e-mail address as a Pythia key-selector suffices to prevent such attacks against users based on their Bitcoin addresses alone. Users also have the option, of course, of using a semi-secret $\mathtt{id}$. A general DoS attack against the Pythia service is also possible, but of similar concern for Bitcoin itself, see [19].

- *Key rotation:* Rotation of an ensemble key k_w (or the master key $\mathtt{msk}$) induces a new value of $\tilde{P}$ and thus a new $(\mathtt{sk}, \mathtt{pk})$ pair and account. A client can handle such rotations in the naïve way: transfer funds from the old address to the new one.

- *Catastrophic failure of* Pythia*:* If a Pythia service fails catastrophically, e.g., $\mathtt{msk}$ or K is lost, then in a typical setting, it is possible simply to reset users' passwords. In the brainwallet case, the result would be loss of virtual-currency assets protected by the server—a familiar event for Bitcoin users described in [70]. This problem can be avoided, for instance, using a threshold implementation of Pythia, as mentioned in Section 3.6 or storing $\mathtt{sk}$ in a secure, offline manner like a safe-deposit box for disaster recovery.

3.6 Threshold Security

In order to gain both redundancy and security, we give a threshold scheme that can be used with a number of Pythia servers to protect a secret under a single password. This scheme uses the [92] secret sharing threshold scheme and gives (k, n) threshold security. That is, initially, n Pythia servers are contacted and used to protect a secret s, and then any k servers can be used to recover s and any adversary that has compromised fewer than k Pythia servers learns no information about s.

Preparation. The client chooses an ensemble key selector w, tweak t, password P, and contacts n Pythia servers to compute $q_i = \text{PRF-Cl}_i(w, t, P)$ mod p for $0 < i \leqslant n$. The client selects a random polynomial of degree $k - 1$ with coefficients from $\mathbb{Z}_p^*$ where p is a suitably large prime: $f(x) = \sum_{j=0}^{k-1} x^j a_j$. Let the secret $s = a_0$. Next the client computes the vector $\Phi = (\phi_1, ..., \phi_n)$ where $\phi_i = f(i) - q_i$. The client durably stores the value Φ, but does not need to protect it (it's not secret). The client also stores public keys p_i from each Pythia server to validate proofs when issuing future queries.

Recovery. The client can reconstruct s if she has Φ by querying any k Pythia servers giving k values q_i. These q_i values can be applied to the corresponding Φ values to retrieve k distinct points that lie on the curve $f(x)$. With k points on a degree $k-1$ curve, the client can use interpolation to recover the unique polynomial $f(x)$, which includes the curve's intercept $a_0 = s$.

Security. If an adversary is given Φ, w, t, the public keys p_i, a ciphertext based on s, and the secrets from $m < k$ Pythia servers, the adversary has no information that will permit her to verify password guesses offline. Compared to [92], this scheme reduces the problem of storing n secrets to having access to n secure OPRFs and durable (but non-secret) storage of the values Φ and public keys p_i.

Verification. Verification of server responses occurs within the Pythia protocol. If a server is detected to be dishonest (or goes out of service), it can be easily replaced by the client without changing the secret s. To replace a Pythia server that is suspected to be compromised or detected as dishonest, the client reconstructs the secret s using any k servers, executes Reset operations on all remaining servers: this effects a cryptographic erasure on the values Φ and $f(x)$. The client then selects a new, random polynomial, keeping a_0 fixed, and generates and stores an updated Φ' that maps to the new polynomial.

3.7 Related Work

We investigated a number of designs based on existing cryptographic primitives in the course of our work, though as mentioned none satisfied all of our design goals. Conventional PRFs built from block ciphers or hash functions fail to offer message privacy or key rotation. Consider instead the construction $H(t\|m)^{k_w}$ for $H\colon \{0,1\}^* \to \mathbb{G}$ a cryptographic hash function mapping onto a group $\mathbb{G}$. This was shown secure as a conventional PRF by [77] assuming decisional Diffie-Hellman (DDH) is hard in $\mathbb{G}$ and when modeling H as a random oracle.

It supports key rotations (in fact it is key-homomorphic, see [22]) and verifiability can be handled using non-interactive zero-knowledge proofs (ZKP) as in Pythia. But this approach fails to provide message privacy if we submit both t and m to the server and have it compute the full hash.

One can achieve message-hiding by using blinding: have the client submit $X = H(t\|m)^r$ for random $r \in \mathbb{Z}_{|\mathbb{G}|}$ and the server reply with X^{k_w} as well as a ZKP proving this was done correctly. The resulting scheme is originally due to [34], and suggested for use by [45] in the context of threshold password-authenticated secret sharing (see also [7, 29, 67, 37]). There an end user interacts with one or more blind signature servers to derive a secret authentication token. If $\mathbb{G}$ comes equipped with a bilinear pairing, one can dispense with ZKPs. The resulting scheme is from [21]; it uses a blinded version of BLS signatures from [25]. However, neither approach provides granular rate limiting when blinding is used: the tweak t is hidden from the server. Even if the client sends t as well, the server cannot verify that it matches the one used to compute X and attackers can thereby bypass rate limits.

To fix this, one might use [45] with a separate secret key for each tweak. This would result in having a different key for each unique w, t pair. Message privacy is maintained by the blinding, and querying $w, \mathsf{t}, \mathsf{H}(\mathsf{t}' \| \mathsf{m})^{\mathsf{r}}$ for $\mathsf{t} \neq \mathsf{t}'$ does not help an attacker circumvent per-tweak rate limiting. But now the server-side storage grows in the number of unique w, t pairs, a client using a single ensemble w must now track N public keys when they use the service for N different tweaks, and key rotation requires N interactions with the PRF server to get N separate update tokens (one per unique tweak for which a PRF output is stored). When N is large and the number of ensembles w is small as in our password storage application, these inefficiencies add significant overheads.

Another issue with the above suggestions is that their security was only previously analyzed in the context of one-more unforgeability (by [84]) as targeted by blind signatures (see [33]) and partially blind signatures of [2]. (Some were analyzed as conventional PRFs, but that is in a model where adversaries do not get access to a blinded server oracle.) The password onion application requires more than unforgeability because message privacy is needed. (A signature could be unforgeable but include the entire message in its signature, and this would obviate the benefits of a PRF service for most applications.) These schemes, however, can be proven to be one-more PRFs, the notion we introduce, under suitable one-more DDH style assumptions using the same proof techniques found in [43].

Fully oblivious PRFs of [46] and their verifiable versions from [55] also do not allow granular rate limiting. We note that the [55] constructions of verifiable OPRFs in the RO model are essentially the [45] protocol above, but with an extra hash computation, making the PRF output $\mathsf{H}'(\mathsf{t} \| \mathsf{m} \| \mathsf{H}(\mathsf{t} \| \mathsf{m})^{\mathsf{k}_w})$. The notion of one-more unpredictability in [43] captures the necessary requirements on the inner cryptographic component, and might modularize and simplify their proofs. Their transform is similar to the unique blind signature to OPRF transformation of [30]. None of these efficient oblivious PRF protocols support key rotations (with compact tokens or otherwise) as the final hashing step destroys updatability.

The setting of capture-resilient devices shares with ours the use of an off-system key-holding server and the desire to perform cryptographic erasure [66, 65]. They only perform protocols for encryption and signing functionalities, however, and not (more broadly useful) PRFs. They also do not support granular rate limiting and master secret key rotation.

Our main construction coincides with prior ones for other contexts. The [90] identity-based non-interactive key exchange protocol computes a symmetric encryption key as $e(H_1(ID_1), H_2(ID_2))^k$ for k a master secret held by a trusted party and ID_1 and ID_2 being the identities of the parties. See [82] for a formal analysis of their scheme. [26] suggest the same construction as a left-or-right constrained PRF. The settings and their goals are different from ours, and in particular one cannot use either as-is for our applications. Naïvely one might hope that returning the constrained PRF key $H_1(t)^{k_w}$ to the client suffices for our applications, but in fact this totally breaks rate-limiting. Security analysis of our protocol requires new techniques, and in particular security must be shown to hold when the adversary has access to a half-blinded oracle — this rules out the techniques used in [82, 26].

The key-updatable encryption of [22] and proxy re-encryption from [20] both support key rotation and could be used to encrypt password hashes in a way that supports compact update tokens and prevents offline dictionary attacks. But this would require encryption and decryption to be handled by the hardening service, preventing message privacy.

Verifiable PRFs as defined by [72, 64, 40, 38] allow one to verify that a known PRF output is correct relative to a public key. Previous verifiable PRF constructions are not oblivious, let alone partially oblivious.

Threshold and distributed PRFs (see [73, 77, 38]) as well as distributed key distribution centers from [77] enable a sufficiently large subset of servers to compute a PRF output, but previous constructions do not provide the granular rate limiting and key rotation we desire. However, it is clear that there are situations where applications would benefit from a threshold implementation of Pythia, for both redundancy and distribution of trust, as discussed in Section 3.6 for the case of brainwallets.

3.8 Discussion

We presented the design and implementation of Pythia, a modern PRF service. Prior works have explored the use of remote cryptographic services to harden keys derived from passwords or otherwise improve resilience to compromise. Pythia, however, transcends existing designs to simultaneously support granular rate limiting, efficient key rotation, and cryptographic erasure. This set of features, which stems from practical requirements in applications such as enterprise password storage, proves to require a new cryptographic primitive that we refer to as a partially oblivious PRF.

Unlike a (fully) oblivious PRF, a partially oblivious PRF causes one portion of an input to be revealed to the server to enable rate limiting and detection of online brute-force attacks. We provided a bilinear-pairing based construction for partially oblivious PRFs that is highly efficient and simple to implement (given a pairings library), and also supports efficient key rotations. A formal proof of security is unobtainable using existing techniques (such as those developed for fully oblivious PRFs). We thus gave new definitions and proof techniques that may be of independent interest.

We implemented Pythia and show that it can be used with a range of applications. Pythia permits fast key rotations, enabling practical reactive and proactive key management. We also explored the use of Pythia to harden brainwallets for cryptocurrencies and detail a threshold security scheme using Pythia.

4 PROTECTING DATA-AT-REST

Clients that choose to store sensitive or valuable information using cloud storage providers may wish to encrypt these data and store the keys separate from the cloud provider's infrastructure. This gives strong security when exactly one of either the cloud provider or the client infrastructure is compromised.

In practice, it is a common requirement to periodically rotate these encryption keys. This restores security in the face of known or unknown compromise of either site. Sending the old and new encryption keys to the cloud provider to update encryption is ill-advised, as it can obviate the security gained by storing keys outside of the cloud provider's infrastructure. Likewise, downloading, re-encrypting, and uploading fresh ciphertext may be costly or impossible. The client may have limited compute power (laptop or mobile phone) and may not have sufficient bandwidth to download and re-upload the entire corpus of ciphertext.

Presently, cloud providers such as Amazon Web Services (see [6]) and Google Cloud Platform (see [48]) accomplish this using a hybrid encryption technique that is practical, but does not accomplish full key rotation. Further, this folklore technique has not been analyzed formally.

The current "gold standard" for encryption is a security notion called authenticated encryption (AE), canonicalized in [88]. Informally, a scheme that provides authenticated encryption guarantees both *confidentiality* (an attacker learns no information from the ciphertext beyond length) and *integrity* (an attacker cannot produce a convincing ciphertext that the encryption key holder will accept). This work establishes formal security notions (definitions) for AE in the updatable encryption setting. Further, we analyze a folklore scheme that is in wide use by practitioners and demonstrate that this scheme falls short of expectations.

Scheme	UP-IND	UP-INT	UP-RE
AE-hybrid†	✗	✗	✗
KSS*	✓	✓	✗
BLMR‡	✗	✗	✗
ReCrypt*	✓	✓	✓

† In-use by practitioners today. * Introduced in this work.
‡ From academic literature.

Table 4.1: Summary of updatable encryption schemes studied and the security notions they fulfill.

Contributions. This chapter provides the following contributions:

- Establishes formal security notions for updatable encryption. Specifically, confidentiality and integrity are established in Section 4.1, and a stronger notion, indistinguishable encryption, is established in Section 4.3.
- Analyzes (and corrects) the existing hybrid updatable encryption scheme in Section 4.2; the existing scheme is known to be in wide use today;
- Introduces *ReCrypt*, a new updatable encryption scheme that meets our strongest notion of updatable encryption security (in Section 4.2); and
- Evaluates the performance of a practical implementation of the ReCrypt scheme in Section 4.4.

4.1 Updatable Authenticated Encryption

We begin by discussing, informally, our assumptions about real-world threats for updatable encryption that motivate our formal models. Then we move to formal treatments: we define updatable AE schemes and notions for the security of these schemes in terms of confidentiality and integrity. Next we investigate the security of updatable schemes that are known to be in production use under these notions.

Threat models. We assume that users of updatable encryption want to store only encrypted information on a remote storage service, keeping the cryptographic keys for these ciphertexts close at-hand. In doing so, we presume that these keys should never be sent to the remote storage service and that the remote service is at risk of compromise. (Otherwise, why bother with encryption in the first place?)

We give attackers wide-abilities in our definitions. We give them: old keys, ciphertext headers and update tokens (which are exchanged during the re-encryption process), and give the attackers read-write abilities on the remote storage service. We presume that some number of old keys have been compromised, new keys generated, and ciphertexts updated. We then ask whether or not an updatable encryption scheme recovers any security and what kind of security is recovered. In particular, we care about the confidentiality and integrity of the updated ciphertexts.

We broadly consider two slightly different settings in terms of security. In the first, attackers are given all those items mentioned in the previous paragraph (keys, headers, tokens, access to updated ciphertexts), but notably, are not given ciphertexts encrypted under the older (compromised) keys. This corresponds, for example, to the following scenario. A client device is lost and it contains keys and ciphertext headers (present from previous updates), but not entire ciphertexts which are stored on the remote storage service. The data owner realizes the device has been lost and uses a copy of the keys and ciphertext headers from another device to perform a key rotation. The attacker gains access to the lost device and later access the remote storage service. This gives the attacker: older keys and ciphertext headers, update tokens (which may be stored on the remote storage service), and updated ciphertexts (headers and bodies) encrypted under fresh keys. In this setting, we ask whether schemes give any confidentiality (UP-IND) or integrity (UP-INT).

In a more challenging setting, we further give the attacker old ciphertexts, those encrypted under compromised keys. Of course, there can be no confidentiality or integrity of these ciphertexts, but after re-encryption, we ask whether updated ciphertexts continue to provide any confidentiality and integrity. In this setting, of course an attacker can read old messages, but she should not be able to read any new messages or updates to these messages, and she should not be able to produce passable forgeries.

Formal Definitions

Now we turn to formal definitions, but first must establish some preliminaries, notably, classic authentication encryption (AE), on which we build updatable AE schemes.

Notation. Where convenient, in this chapter we use subscript notation for the first input to a function, procedure, or oracle query when the first element is a cryptographic key. For example, $E_k(M)$ denotes $E(k, M)$.

Definition 4.1 (Authenticated encryption)**.** *An* authenticated encryption scheme π *is a tuple of algorithms* $(\mathcal{K}, \mathcal{E}, \mathcal{D})$. $\mathcal{K}$ *is a randomized algorithm outputting keys. We denote by* $\mathcal{E}_k(\cdot)$ *the randomized algorithm for encryption by key* k *and by* $\mathcal{D}_k(\cdot)$ *decryption. Decryption is a deterministic algorithm and outputs the distinguished symbol* $\perp$ *to denote a failed decryption.*

It should be noted that we consider randomized AE schemes rather than nonce-based ones[1].

[1]The difference between randomized and nonce-based encryption schemes is subtle. Here we choose to use randomized encryption with inputs only the secret key and message; the encryption routine chooses a random initialization vector which ensures every output is distinct with high probability. By contrast, nonce-based encryption

We use the all-in-one authenticated encryption security definition from [87].

Definition 4.2 (Authenticated Encryption Security). *Let* $\pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ *be an authenticated encryption scheme. Let* Enc, Dec *be oracles whose behaviors depends on hidden values* $\mathsf{b} \in \{0, 1\}$ *and key* $\mathsf{k} \leftarrow\!\$\, \mathcal{K}$. Enc *takes as input a bit string* m *and produces* $\mathcal{E}_\mathsf{k}(\mathsf{m})$ *when* $\mathsf{b} = 0$, *and produces a random string of the same length otherwise.* Dec *takes as input a bit string* C *and produces* $\mathcal{D}_\mathsf{k}(\mathsf{C})$ *when* $\mathsf{b} = 0$, *and produces* $\perp$ *otherwise.*

Let AE-ROR$_\pi^\mathcal{A}$ *be the game in which an adversary* $\mathcal{A}$ *has access to the* Enc *and* Dec *oracles and must output a bit* b'. *The game outputs* true *when* $\mathsf{b} = \mathsf{b}'$. *We require that the adversary not submit outputs from the* Enc *oracle to the* Dec *oracle.*

We associate to an AE scheme π and AE-ROR adversary $\mathcal{A}$ the following advantage measure[2]:

$$\mathbf{Adv}_\pi^{\mathrm{ae}}(\mathcal{A}) = 2 \cdot \Pr\left[\text{AE-ROR}_\pi^\mathcal{A} \Rightarrow \mathsf{true}\right] - 1.$$

Unless otherwise stated, our AE schemes will be length-regular, so that the lengths of ciphertexts depend only on the lengths of plaintexts. This ensures that the above definition also implies a standard "left-or-right" security definition.

Below we establish formal definition for updatable AE. This definition is similar to that of [23], but permits some operations to produce the distinguished error symbol $\perp$ (required for integrity notions). Informally, an updatable AE scheme is one that permits (authenticated) encryption

(introduced by [86]) is deterministic and requires the client to provide a non-repeating value, the nonce. We choose to use randomized schemes to simplify the analysis.

[2]Informally, an advantage measure is the typical metric for the strength of a concrete adversary, in a given game, with concrete parameters. The value ranges $[0, 1]$: 0 indicates the adversary does no better than simple guessing; a value of 1 indicates the adversary wins the game every time. Any value not very close to 0 indicates a security failure with the given adversary and scheme.

and decryption, provides rekey token creation, and re-encryption operations. Re-encryption transforms a ciphertext from one that is valid under an initial key to a ciphertext valid under a new key using a rekey token. Decryption, rekey token creation, and re-encryption may all return the error symbol $\perp$ if the inputs fail integrity verification.

Definition 4.3 (Updatable AE). *An* updatable AE scheme *is a tuple of algorithms* $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{RekeyGen}, \mathsf{ReEnc}, \mathsf{Dec})$ *further defined as follows:*

- $\mathsf{KeyGen}(\lambda) \to k$. *On input security parameter* λ*, outputs secret key* k.
- $\mathsf{Enc}(k, m) \to C$. *On inputs key* k *and message* m *where* m *is a bitstring of arbitrary length, outputs ciphertext* $C = (\widetilde{C}, \overline{C})$*, where* $\widetilde{C}, \overline{C}$ *are the header and body, respectively.*
- $\mathsf{RekeyGen}(k_1, k_2, \widetilde{C}) \to \Delta_{1,2,\widetilde{C}}$. *On inputs two secret keys* (k_1, k_2) *and a ciphertext header* $\widetilde{C}$*, outputs either a rekey token or the error symbol* $\perp$.
- $\mathsf{ReEnc}(\Delta_{1,2,\widetilde{C}}, C_1) \to C_2$. *On inputs a rekey token* Δ *and a ciphertext* C_1*, outputs a new ciphertext or the error symbol* $\perp$.
- $\mathsf{Dec}(k, C) \to m$. *On input a secret key and ciphertext, outputs a message* m *or error* $\perp$.

Each of the algorithms above may be probabilistic except for ReEnc which we require to be deterministic. All algorithms must be efficiently computable.

Informally, an updatable AE scheme is *correct* if the message can always be recovered by decrypting with the proper secret key. Since any number of updates can occur, we use a recursive definition to capture correctness.

Definition 4.4 (Correctness). *Let* m *be any message and* $k_1, ..., k_T$ *be any sequence of keys produced by calls to* KeyGen. *Let* $C_1 = (\widetilde{C}_1, \overline{C}_1) = \mathsf{Enc}(k_1, m)$*, and let* C_{t+1} *be recursively defined as:*

$$C_{t+1} = \mathsf{ReEnc}(\mathsf{RekeyGen}(k_t, k_{t+1}, \widetilde{C}_t), C_t)$$

where $1 \leqslant t \leqslant T$.

Let Π *be an updatable AE scheme.* Π *is* correct *if* $\mathit{Dec}(k_T, C_T) = m$ *with probability 1 for any* T.

Security of Updatable AE

For updatable encryption, we are interested in settings where keys may be compromised. We seek security notions (definitions) that measure whether schemes recover security by generating fresh (uncompromised) keys and updating ciphertexts to these keys.

We establish three security notions for updatable AE schemes: UP-IND, UP-INT, and UP-RE. At a high level, UP-IND and UP-INT capture confidentiality and integrity notions (respectively) in the setting where keys may be compromised but not (old) ciphertexts corresponding to those keys. UP-RE is the strongest notion of security and requires confidentiality and integrity for ciphertexts updated to uncompromised keys when an attacker has access to some keys and ciphertexts (both old and new).

Table 4.1 at the beginning of Chapter 4 shows four schemes (two existing and two novel) studied in this work and which security notions they satisfy.

We define security games in the style of [13]. Figure 4.1 shows the confidentiality and integrity games for UP-IND and UP-INT. In each game there are a sequence of secret keys $k_1 \ldots k_s \ldots k_{s+t}$ generated at the beginning of the game. Keys $k_1 \ldots k_s$ are the compromised keys; these are given to the adversary. Keys $k_{s+1}...sk[s+t]$ are the uncompromised keys. The adversary is given access to a series of oracles (Enc, RekeyGen, ReEnc) and either a left-or-right oracle (UP-IND) or a Try oracle (UP-INT).

Limiting the adversary. Both games require carefully constructed invalidity procedures to make the games useful by preventing trivial wins by the adversary. For instance, in the case of UP-IND, the adversary seeks

```
UP-IND
    $b \leftarrow\$ \{0,1\}$
    $k_1, \ldots, k_{s+t} \leftarrow\$ \mathsf{KeyGen}()$
    $b' \leftarrow\$ \mathcal{A}^{O}(k_1, \ldots, k_s)$
    return $(b' = b)$

Enc$(i, m)$
    return $\mathsf{Enc}(k_i, m)$

ReKeyGen$(i, j, \widetilde{C})$
    if $\mathsf{Invalid}(i, j, \widetilde{C})$ then return $\perp$
    $\Delta_{i,j,\widetilde{C}} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
    return $\Delta_{i,j,\widetilde{C}}$

ReEnc$(i, j, C)$
    $\Delta_{i,j,\widetilde{C}} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
    $C' = (\widetilde{C}', \overline{C}') \leftarrow \mathsf{ReEnc}(\Delta_{i,j,\widetilde{C}}, C)$
    if $\mathsf{Invalid}(i, j, \widetilde{C})$ then return $\widetilde{C}'$
    else return $C'$

LR$(i, m_0, m_1)$
    if $i \leqslant s$ then return $\perp$
    $C \leftarrow\$ \mathsf{Enc}(k_i, m_b)$
    return $C$
```

```
UP-INT
    $\mathsf{win} \leftarrow \mathsf{false}$
    $k_1, \ldots, k_{s+t} \leftarrow\$ \mathsf{KeyGen}()$
    $\mathcal{A}^{O}(k_1, \ldots, k_{s+t})$
    return $\mathsf{win}$

Enc$(i, m)$
    return $\mathsf{Enc}(k_i, m)$

ReKeyGen$(i, j, \widetilde{C})$
    return $\mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$

ReEnc$(i, j, (\widetilde{C}, \overline{C}))$
    $\Delta_{i,j,\widetilde{C}} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
    $C' \leftarrow \mathsf{ReEnc}(\Delta_{i,j,\widetilde{C}}, C)$
    return $C'$

Try$(i, C)$
    if $\mathsf{InvalidCtxt}(i, C)$ then return $\perp$
    $M \leftarrow \mathsf{Dec}(k_i, C)$
    if $M = \perp$ then return $\perp$
    $\mathsf{win} \leftarrow \mathsf{true}$
    return $M$
```

Figure 4.1: Confidentiality (left) and integrity (right) games for updatable encryption security. Note that the procedures Invalid and InvalidCtxt rely on the use of a transcript of oracle calls – this is left implicit for concision.

to discover the value of a hidden bit b. This bit determines whether or not the first or second message is encrypted under an uncompromised key when the adversary queries the LR oracle. For example, imagine an adversary gets a ciphertext C_i from the LR oracle for uncompromised key k_i. Then the adversary requests a rekey token $\Delta_{i,1,\widetilde{C}_1}$. The adversary can now apply the ReEnc routine herself, produce C_i, then recover the underlying message m_b by decrypting with the compromised key k_1 given to the adversary at the start of the game.

To prevent this and other trivial wins, we define the function $\mathsf{Derived}_{\mathsf{LR}}$ and invalidity procedures used in the game UP-IND.

Informally, these invalidity procedures restrict the adversary from calling RekeyGen or ReEnc oracles using a ciphertext that was derived (either directly or eventually) from a ciphertext returned from an LR oracle query.

Definition 4.5 (LR-derived headers)**.** *We recursively define* $\mathit{Derived}_{LR}(i, \widetilde{C})$ *as a function that outputs* true *if:*

- $\widetilde{C}$ *was output in response to some previous query* $LR(i, m_0, m_1)$*; or*
- $\widetilde{C}$ *was output in response to some previous query* $\mathit{ReEnc}(j, i, C')$ *and* $\mathit{Derived}_{LR}(j, C') = \mathsf{true}$*; or*
- $\widetilde{C}$ *was output in response to some previous query* $\mathit{ReEnc}(\Delta_{j,i,\widetilde{C}''}, C')$*, and* $\Delta_{j,i,\widetilde{C}'}$ *was output in response to some query* $\mathit{RekeyGen}(j, i, C')$ *where* $\mathit{Derived}_{LR}(j, \widetilde{C}') = \mathsf{true}$*;*

and outputs false *otherwise.*

With a proper definition for LR-derived headers, we specify our invalidity routine. Informally, we deny the adversary the ability to query the ReEnc and RekeyGen oracles with a header that was derived in some way from a query to the LR oracle.

Definition 4.6 (Invalid oracle query for UP-IND)**.** *Invalid*$(i, j, \widetilde{C})$ *outputs* true *if* $j > s$ *and Derived*$_{LR}(i, \widetilde{C}) =$ true*; and outputs* false *otherwise.*

The full game UP-IND is shown in Figure 4.1. We associate to an UP-IND adversary $\mathcal{A}$ and scheme Π the following advantage measure:

$$\mathbf{Adv}^{\text{up-ind}}_{\Pi,s,t}(\mathcal{A}) = 2 \cdot \Pr\left[\,\text{UP-IND}^{\mathcal{A}}_{\Pi,s,t} \Rightarrow \mathsf{true}\,\right] - 1.$$

Similarly, for the integrity game UP-INT, we define an invalidity procedure that ensures that an adversary cannot submit a ciphertext that is trivially derived from an Enc or ReEnc oracle query (or a key given to the adversary).

Definition 4.7 (Invalid oracle query for UP-INT)**.** *For the security game UP-INT, we define InvalidCtxt*(i, C) *inductively. This procedure outputs* true *if:*

- $i \leqslant s$*: That is,* k_i *is a key given to the adversary; or*
- C *was output in response to some previous query Enc*(i, m)*; or*
- C *was output in response to some previous query ReEnc*(j, i, C') *and InvalidCtxt*$(j, C') =$ true*; or*
- $C = (\widetilde{C}, \overline{C})$ *is the ciphertext that is produced by running the routine ReEnc*$(\Delta_{j,i,\widetilde{C}}, C')$*, where* $C' = (\widetilde{C}', \overline{C}')$ *and* $\Delta_{j,i,\widetilde{C}}$ *was output from some previous query RekeyGen*$(j, i, \widetilde{C}')$ *and InvalidCtxt*$(j, C') =$ true.

InvalidCtxt outputs false *otherwise.*

Real-word settings. These confidentiality and integrity definitions are meant to capture security goals in the setting where attackers have access to some compromised keys, read-and-write access to ciphertexts (encrypted under uncompromised keys), and access to update tokens. Notably, attackers are not given access to ciphertexts produced under compromised keys. We argue briefly that this maps to at least *some* real-world scenarios. (Later we discuss a real-world setting where these definitions are insufficient.)

One such scenario is the case where encryption keys are stored on a portable device (e.g. USB storage drive, laptop, or mobile phone); and ciphertexts are stored with access control at a cloud storage provider. If a user or employee loses the device containing keys (or the device is stolen or detectably compromised), the obvious recourse is for the data owner to generate new keys and execute re-encryption. If re-encryption is accomplished before an attacker gains access to the ciphertexts, then a scheme that meets UP-IND and UP-INT provides confidentiality and integrity guarantees.

4.2 Practical Schemes

We show that the scheme in widespread use today meets neither UP-IND nor UP-INT security. This scheme, which we call *AE-hybrid*, uses a two-part construction. The header is called the key encapsulation message (KEM) and encrypts a per-file encryption key with the data owner's long-term key (which is possibly reused across may files). The ciphertext body (called the data encapsulation message [DEM]) is an encryption of the message proper under the per-file encryption key. The full scheme is detailed in Figure 4.2.

Both header and body use an authenticated encryption primitive (such as AES-GCM). The intuition is that the scheme should inherit the confidentiality and integrity guarantees of an AE scheme when key rotations are performed. We show that, contrary to this intuition, the scheme grants neither if at least one key has been compromised.

Re-encryption is straightforward and performant. The data owner retrieves the headers for ciphertexts she wishes to update. (Retrieving only headers is typically much faster: headers are compact and of fixed length, whereas ciphertext bodies may be of arbitrary length.) The data owner decrypts the headers under the old key, re-encrypts them under

```
$\mathsf{Enc}(k, m)$
    $x \leftarrow\!\$\ \mathcal{K}$
    $\widetilde{C} \leftarrow\!\$\ \mathcal{E}(k, x)$
    $\overline{C} \leftarrow\!\$\ \mathcal{E}(x, m)$
    return $(\widetilde{C}, \overline{C})$

$\mathsf{RekeyGen}(k_1, k_2, \widetilde{C})$
    $x = \mathcal{D}(k_1, \widetilde{C})$
    if $x = \perp$ return $\perp$
    $\Delta_{1,2,\widetilde{C}} \leftarrow\!\$\ \mathcal{E}(k_2, x)$
    return $\Delta_{1,2,\widetilde{C}}$

$\mathsf{Dec}(k, (\widetilde{C}, \overline{C}))$
    $x = \mathcal{D}(k, \widetilde{C})$
    if $x = \perp$ return $\perp$
    $m = \mathcal{D}(x, \overline{C})$
    return $m$

$\mathsf{KeyGen}$ : return $k \leftarrow\!\$\ \mathcal{K}$
$\mathsf{ReEnc}(\Delta_{1,2,\widetilde{C}}, (\widetilde{C}, \overline{C}))$ : return $C' = (\Delta_{1,2,\widetilde{C}}, \overline{C})$
```

Figure 4.2: Algorithms for the AE-hybrid updatable AE scheme.

the new key, and uploads the new headers to the storage provider (the new headers become the rekey tokens Δ in our notation).

We note that this scheme is presently in widespread deployment. Both Google Cloud Platform (see [48]) and Amazon Web Services (see [6]) are known to use AE-hybrid in production.

Insecurity of AE-Hybrid

We first show that AE-hybrid fails to provide confidentiality under our definition UP-IND. The intuition here is straightforward: the adversary needs only one compromised key and a rekey token associated with that key. Rekey tokens include the per-message encryption key and this encryption key is never properly rotated in the AE-hybrid scheme. Once the adversary recovers the underlying encryption for the ciphertext body, she can always recover the ciphertext body no matter how many re-encryptions are performed. Now we demonstrate the same formally, first the limitation with confidentiality, and then with integrity.

Theorem 4.8 (AE-hybrid does not recover confidentiality (UP-IND) with re-encryption)**.** *If Π is the AE-hybrid updatable encryption scheme then there* *exists an adversary $\mathcal{A}$ for the UP-IND security game, that makes 2 queries where:*

$$\mathbf{Adv}^{\textit{up-ind}}_{\Pi,s,t}(\mathcal{A}) = 1$$

for all $t \geqslant 1$ *(at least one compromised key) and* $s \geqslant 1$ *(no matter the number re-encryptions).*

Proof. Our proof is by construction of $\mathcal{A}$. To begin, the adversary $\mathcal{A}$ makes an initial query $\mathsf{LR}(1, m_0, m_1)$ for distinct messages $m_0 \neq m_1$ and receives challenge ciphertext $C^* = (\mathcal{E}_{k_1}(x), \mathcal{E}_x(m_b))$. $\mathcal{A}$ makes a second query $\mathsf{RekeyGen}(1, t+1, C^*)$. k_{t+1} is a compromised key and so $\mathsf{Invalid}$ outputs false; $\mathcal{A}$ receives the re-encrypted ciphertext header $\widetilde{C}' = \mathcal{E}_{k_{t+1}}(x)$. $\mathcal{A}$ recovers $x = \mathcal{D}_{k_1}(\widetilde{C}')$, computes $m_b = \mathcal{D}_x(\overline{C}^*)$, checks whether $m_b = m_0$ or $m_b = m_1$, and outputs the correct value of b. □

Similarly, AE-hybrid fails to provide integrity in the UP-INT game. A similar attack applies, once an attacker has recovered the key used to encrypt the ciphertext body, she can create arbitrary forgeries without needing to modify the header. Formal treatment of integrity follows.

Theorem 4.9 (AE-hybrid does not recover integrity (UP-INT) with re-encryption)**.** *If* Π *is the updatable AE scheme AE-hybrid there exists an adversary* $\mathcal{A}$ *making 2 queries and one* Try *query such that* $\mathbf{Adv}^{\textit{up-int}}_{\Pi,s,t}(\mathcal{A}) = 1$ *for all* $s \geqslant 1$ *and* $t \geqslant 1$.

Proof. Again, our proof is by construction of $\mathcal{A}$. $\mathcal{A}$ first queries $\text{Enc}(1, m)$ to obtain an encryption $C = (\mathcal{E}(k_1, x), \mathcal{E}(x, m))$, and subsequently queries $\text{ReEnc}(1, t+1, C)$, receiving the re-encryption $C' = (\mathcal{E}(k_{t+1}, x), \mathcal{E}(x, m))$. Since $\mathcal{A}$ has key k_{t+1}, $\mathcal{A}$ recovers $x = \mathcal{D}(k_{t+1}, \widetilde{C}')$ by performing the decryption locally.

Finally, $\mathcal{A}$ constructs the ciphertext $C^* = (\widetilde{C}, \mathcal{E}(x, m'))$ for some $m' \neq m$ and queries $\text{Try}(1, C^*)$. Since C^* is not derived from C and k_1 is not compromised, UP-INT outputs true. □

Note in the theorems above, at least one key compromise and one key rotation is required. This is, of course, the point of using an updatable AE scheme. If no keys are corrupted, AE-hybrid does provide UP-IND and UP-INT security. That is, use of AE-hybrid doesn't weaken security in the situation where no keys are compromised. Formal proof of this can be found in [44].

Fixing AE-hybrid

We demonstrate a straightforward modification to the AE-hybrid scheme that gives both confidentiality and integrity (UP-IND, UP-INT). The scheme is called KSS (KEM/DEM with secret sharing) and is shown in Figure 4.3. The essentials are this: the ciphertext body is augmented with a share of the secret key. The remaining share is included in the header and stored alongside a (keyed) hash of the contents of the ciphertext body. The intuition is that using a keyed hashed of the ciphertext body binds the header and body together (so that simple swaps are prevented, i.e. rollback attacks). Further, using a secret share means that acquiring a rekey token (header) related to a compromised key is insufficient to recover the key used in the data encapsulation message (DEM). The DEM key is never changed, but the secret shares are updated during re-encryption so that new ciphertext bodies leak no information about old secret shares.

We note that the performance of KSS is identical to that of AE-hybrid.

Now we demonstrate formally that KSS provides both UP-IND and UP-INT security.

Theorem 4.10 (Confidentiality (UP-IND) of KSS)**.** *Let* $\pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ *be a symmetric encryption scheme and* Π *be the updatable AE scheme KSS using* π *as defined in Figure 4.3. Then for any adversary* $\mathcal{A}$ *for the game UP-IND, making at most* q *queries to the* LR *oracle, there exists an adversary* $\mathcal{B}$ *for the AE security*

```
$\mathsf{Enc}(k, m)$
    $x, y \leftarrow\!\$\, \mathcal{K}$
    $\chi = x \oplus y$
    $\beta \leftarrow\!\$\, \mathcal{E}_x(m)$
    $\widetilde{C} \leftarrow\!\$\, \mathcal{E}_k(\chi \parallel h(\beta))$
    $\overline{C} = (y, \beta)$
    return $(\widetilde{C}, \overline{C})$

$\mathsf{RekeyGen}(k_1, k_2, \widetilde{C})$
    $\chi \parallel \tau = \mathcal{D}_{k_1}(\widetilde{C})$
    $y' \leftarrow\!\$\, \mathcal{K}$
    $\nu = \mathcal{E}_{k_2}((\chi \oplus y') \parallel \tau)$
    return $(y', \nu)$

$\mathsf{Dec}(k, (\widetilde{C}, \overline{C}))$
    $\chi \parallel \tau = \mathcal{D}_k(\widetilde{C})$
    $y, \beta = \overline{C}$
    if $\tau \neq h(\beta)$ then
        return $\perp$
    endif
    $x = \chi \oplus y$
    $m = \mathcal{D}_x(\beta)$
    return $m$

$\mathsf{ReEnc}(\Delta_{1,2,\widetilde{C}}, (\widetilde{C}, \overline{C}))$
    $y', \nu = \Delta_{1,2,\widetilde{C}}$
    $y, \beta = \overline{C}$
    $\overline{C}' = (y \oplus y', \beta)$
    return $(\nu, \overline{C}')$

$\mathsf{KeyGen}$: return $k \leftarrow\!\$\, \mathcal{K}$
```

Figure 4.3: The KSS updatable AE scheme.

game where:

$$\mathbf{Adv}^{\textit{up-ind}}_{\Pi,s,t}(\mathcal{A}) \;\leqslant\; 2(t+q)\cdot \mathbf{Adv}^{\textit{ae}}_{\pi}(\mathcal{B})$$

for all $s \geqslant 0, t \geqslant 1$.

Proof. We argue using a series of games that the advantage of any adversary in the UP-IND game for KSS is bounded by the advantage in the AE security game for the underlying AE scheme π. The first game, G_0, is the UP-IND game for KSS using the underlying scheme π. For $1 \leqslant i \leqslant t$, in game G_i we replace all ciphertext headers encrypted under key k_i and instead return a random string of the same length used to answer the query. The value of the returned ciphertext header $\widetilde{C}$ is stored along with the encrypted value $(\chi \parallel \tau)$ in order to simulate later calls to ReEnc and ReKeyGen.

Let S_i be the event that G_i outputs true. We claim that for $0 < i \leqslant t$, there exists $\mathcal{B}$ such that:

$$|\Pr[S_{i-1}] - \Pr[S_i]| \leqslant \mathbf{Adv}^{\mathrm{ae}}_{\pi}(\mathcal{B}).$$

We construct $\mathcal{B}$ by using the Enc oracle in the AE security game for π to encrypt ciphertext headers. When the hidden bit $\widehat{b}$ in the AE game is 0, then $\mathcal{B}$ perfectly simulates game G_{i-1} for key k_i and when $\widehat{b} = 1$, then $\mathcal{B}$ perfectly simulates game G_i. $\mathcal{B}$ simply returns as a guess $\widehat{b} = 1$ if the adversary is correct. Any difference between the success probabilities in G_{i-1} and G_i results in an advantage for the adversary.

Now, we observe that the adversary in game G_t cannot learn anything about the DEM key x used to encrypt a challenge. Even through a re-encryption to a corrupted key, the most the adversary can learn is the value $\chi' = x \oplus y'$, where y' is in the ciphertext body and unobtainable by the adversary.

Hence consider another set of hybrids G_j for $t < j \leqslant t + q$, where the adversary makes at most q queries to the left-or-right oracle LR. In the same spirit as the previous hybrids, we construct an AE adversary such that $|\Pr[S_{j-1}] - \Pr[S_j]| \leqslant \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B})$, this time replacing the encryption of the DEM during a challenge query with the output of the AE encryption oracle.

Finally, observe that in game G_{t+q}, all outputs from LR oracle are of the form (r, χ, z) where r, z are random strings. Since the outputs are unrelated to message inputs, and indeed the hidden bit b, the adversary can learn nothing from oracle queries and so $\Pr[S_{t+q}] = \frac{1}{2}$ and we conclude:

$$\begin{aligned}
\mathbf{Adv}^{\text{up-ind}}_{\Pi,s,t}(\mathcal{A}) &= 2 \cdot \Pr[S_0] - 1 \\
&= 2 \cdot (\Pr[S_0] - \Pr[S_1] + \Pr[S_1] + \dots \\
&\qquad - \Pr[S_{t+q}] + \Pr[S_{t+q}]) - 1 \\
&\leqslant 2(t + q) \cdot \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B}) \ . \qquad \square
\end{aligned}$$

Our modification to include an encrypted hash of the ciphertext provides necessary integrity protection. As we will see in the following theorem, collision resistance of the hash function is sufficient to provide UP-INT security, since the hash is integrity-protected by the AE encryption of the KEM. The hash is encrypted to avoid compromise of the ciphertext header being sufficient to distinguish messages.

We achieve collision resistance by assuming h to be a random oracle. However, this assumption could be avoided by re-using the DEM key x to additionally key the hash function.

Theorem 4.11 (Integrity (UP-INT) of KSS). *Let $\pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ be a symmetric encryption scheme, h be a cryptographic hash function modelled as a random oracle with output length ℓ_h, and Π be the updatable AE scheme KSS using π and h as defined in Figure 4.3. Then for any adversary $\mathcal{A}$, making at most q_h queries to the random oracle h, there exists an adversary $\mathcal{B}$ for the AE security game where:*

$$\mathbf{Adv}^{\textit{up-int}}_{\Pi,s,t}(\mathcal{A}) \leqslant t \cdot \mathbf{Adv}^{ae}_{\pi}(\mathcal{B}) + \frac{q_h^2}{2^{\ell_h}}$$

for all $s \geqslant 0, t > 0$.

Proof. We use the same sequence of games as in the previous proof, leveraging the integrity properties of authenticated encryption.

In game G_i for $1 \leqslant i \leqslant t$, for any query (j, C) to the Try oracle where $j \leqslant i$, the decryption of the ciphertext header $\widetilde{C}$ will always decrypt to $\perp$, unless $\widetilde{C}$ was a previously generated random string output by the encryption oracle.

Hence in game G_t, the adversary can only win by re-using a previously seen header $\widetilde{C}$.

Therefore, the adversary can only win by submitting a ciphertext of the form: $(\widetilde{C}, \overline{C}')$, where $\overline{C}' \neq \overline{C}$, the ciphertext body returned in the same query as $\widetilde{C}$.

Let $\overline{C} = (y, \overline{C}^1)$ and thus we require $\overline{C}' = (y', \mathcal{E}(\chi \oplus y', m'))$ for either $y' \neq y$ or $\mathcal{E}(\chi \oplus y', m') \neq \overline{C}^1$. Note that the adversary can learn the value of x, y used in $\overline{C}$ by querying ReKeyGen$(i, t+1, \widetilde{C})$ and receiving $\mathcal{E}(k_{t+1}, \chi \parallel h(\overline{C}^1)), y$. From this, they can simply decrypt and learn $x = \chi \oplus y$

To succeed, the adversary needs to find y', m' such that $h(m') = \mathcal{D}(\chi \oplus y', \mathcal{E}(x, h(m)))$. Modelling h as a random oracle, for any value of $y' \neq y$, the probability that the adversary finds an input z to h, such that $h(z) = \mathcal{D}(\chi \oplus y', \mathcal{E}(x, h(m)))$ is $\frac{1}{2^{\ell_h}}$. If such a value z is found, the adversary can compute $m' = \mathcal{D}(\chi \oplus y, z)$. However, the probability that the adversary, making at most q_h oracle queries to h finds such a value is given by $\frac{q_h}{2^{\ell_h}}$.

Alternatively, for $y = y'$, the adversary needs to find a message $m' \neq m$ such that $h(m') = h(m)$. Again, modelling h as a random oracle, the probability that the adversary, making at most q_h oracle queries to h, finds a collision is given by $\frac{q_h^2}{2^{\ell_h}}$.

Since this probability is greater than the probability of finding a pre-image when $y' \neq y$, the probability that the adversary wins in G_{t+1} is bounded by this probability of finding a collision, and we conclude that:

$$\mathbf{Adv}^{\text{up-int}}_{\Pi,s,t}(\mathcal{A}) \leqslant t \cdot \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B}) + \frac{q_h^2}{2^{\ell_h}}. \qquad \square$$

Beyond UP-IND and UP-INT. The careful reader will notice that the UP-IND and UP-INT definitions include a severe limit to adversaries: the definitions assume that an adversary never has simultaneous access to a ciphertext and key from the same key rotation era. Further, careful inspection of KSS (and AE-hybrid) show that the DEM key is never refreshed. The motivating, real-world scenario where these assumptions breaks down is called an exfiltration attack which we describe briefly.

If an attacker has simultaneous (even limited) access to both keys and ciphertexts, she can decrypt message headers, recover the DEM keys and exfiltrate them for later exploitation. The obvious scenario is an authorized user with the foresight to recover and exfiltrate DEM keys. If this user is later de-authorized (say, leaves the company), it is common practice to perform re-encryption to invalidate existing keys. However, in the case of KSS (and AE-hybrid), re-encryption is of limited value if the user still

```
$\mathsf{Enc}(k, m)$
$x, y \leftarrow\$ \mathcal{K}$
$\chi = x + y$
$\tau = h(m) + F_x(0)$
for $1 \leqslant i \leqslant \ell$
    $\overline{C}_i = m_i + F_x(i)$
$\widetilde{C} \leftarrow\$ \mathcal{E}_k(\chi \,\|\, \tau)$
$\overline{C} = (y, \overline{C}_1, \ldots, \overline{C}_\ell)$
return $(\widetilde{C}, \overline{C})$
```

```
$\mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
$\chi \,\|\, \tau = \mathcal{D}_{k_i}(\widetilde{C})$
if $\chi \| \tau = \perp$ then
    return $\perp$
$x', y' \leftarrow\$ \mathcal{K}$
$\chi' = \chi + x' + y'$
$\tau' = \tau + F_{x'}(0)$
$\widetilde{C}' \leftarrow\$ \mathcal{E}_{k_j}(\chi' \,\|\, \tau')$
$\Delta_{i,j,\widetilde{C}} = (\widetilde{C}', x', y')$
return $\Delta_{i,j,\widetilde{C}}$
```

```
$\mathsf{ReEnc}(\Delta_{i,j,\widetilde{C}}, (\widetilde{C}, \overline{C}))$
$(\widetilde{C}', x', y') = \Delta_{i,j,\widetilde{C}}$
$y = \overline{C}_0$
for $1 \leqslant i \leqslant \ell$
    $\overline{C}'_i = \overline{C}_i + F_{x'}(i)$
$\overline{C}' = (y + y', \overline{C}'_1, \ldots, \overline{C}'_\ell)$
return $(\widetilde{C}', \overline{C}')$
```

```
$\mathsf{Dec}(k, (\widetilde{C}, \overline{C}))$
$\chi \| \tau = \mathcal{D}_k(\widetilde{C})$
if $(\chi \| \tau) = \perp$ then
    return $\perp$
$y = \overline{C}_0$
$x = \chi - y$
for $1 \leqslant i \leqslant \ell$
    $m_i = \overline{C}_i - F_x(i)$
$m = (m_0, \ldots, m_\ell)$
$\hat{\tau} = h(m) + F_x(0)$
if $\hat{\tau} = \tau$ then
    return $m$
else return $\perp$
```

```
$\mathsf{KeyGen}$ : return $k \leftarrow\$ \mathcal{K}g()$
```

Figure 4.4: The ReCrypt updatable, authenticated encryption scheme.

holds the DEM keys.

We now establish a stronger security notion that removes these limits from the adversary and demands that the entire ciphertext is refreshed during re-encryption. We also introduce a new encryption scheme that meets this level of security and report on a prototype implementation.

4.3 Indistinguishable re-encryption

We formalize our strong notion of updatable authenticated encryption in Figure 4.5. We call this notion UP-RE. It is conceptually similar to UP-IND except that the adversary is given a left-or-right *re-encryption* oracle that operates on ciphertexts rather than on messages (plaintexts).

Similar to UP-IND, an $\mathsf{Invalid}_{\mathsf{RE}}$ procedure is required to prevent trivial wins and retain the definition's utility. For UP-RE the procedures $\mathsf{Derived}_{\mathsf{LR}}$ and $\mathsf{Invalid}$ are nearly identical to those specified in the UP-IND game, except the LR oracle is replaced with a ReLR oracle; the formal definitions follow.

Definition 4.12 (ReLR-derived headers). *We recursively define the function* $\mathsf{Derived}_{\mathsf{ReLR}}(i, \widetilde{C})$ *to output* true *iff any of the following conditions hold:*

- $\widetilde{C}$ *was the ciphertext header output in response to a query* $\text{ReLR}(i, C_0, C_1)$.
- $\widetilde{C}$ *was the ciphertext header output in response to a query* $\text{ReEnc}(j, i, C')$ *and* $\mathsf{Derived}_{\mathsf{ReLR}}(j, \widetilde{C}') = \mathsf{true}$.
- $\widetilde{C}$ *is the ciphertext header output by running* $\mathsf{ReEnc}(\Delta_{j,i,\widetilde{C}'}, C')$ *where* $\Delta_{j,i,\widetilde{C}'}$ *is the result of a query* $\text{ReKeyGen}(j, i, C')$ *for which* $\mathsf{Derived}_{\mathsf{ReLR}}(j, \widetilde{C}') = \mathsf{true}$.

The procedure $\mathsf{Invalid}_{\mathsf{RE}}$ output true if $\mathsf{Derived}_{\mathsf{ReLR}}(i, \widetilde{C})$ outputs true and $j \leqslant s$.

We associate to an updatable encryption scheme Π, UP-RE adversary $\mathcal{A}$, and parameters s, t the advantage measure:

$$\mathbf{Adv}^{\text{up-re}}_{\Pi,s,t}(\mathcal{A}) = 2 \cdot \Pr\left[\text{UP-RE}^{\mathcal{A}}_{\Pi,s,t} \Rightarrow \mathsf{true}\right] - 1 .$$

Informally, an updatable encryption scheme is UP-RE secure if no adversary can achieve advantage far from zero given reasonable resources (run time, queries, and number of keys).

The ReCrypt scheme. We introduce an updatable authenticated encryption scheme *ReCrypt* that provides UP-RE security. This scheme builds on the updatable encryption scheme of [23] which we refer to as *BLMR*. BLMR makes use of a key homomorphic pseudorandom function (PRF) to provide full refresh of a ciphertext without requiring decryption. The BLMR scheme targets strictly weaker security guarantees (notably, ignored integrity). Further, BLMR targets an even weaker definition of confidentiality and does not meet our own UP-IND notion. We omit the proof of this claim here, but it can be found in [44].

```
UP-RE
    $b \leftarrow\$ \{0,1\}$
    $k_1, \ldots, k_{s+t} \leftarrow\$ \mathsf{KeyGen}()$
    $b' \leftarrow\$ \mathcal{A}^{O}(k_1, \ldots, k_s)$
    return $(b' = b)$

Enc$(i, m)$
    return $\mathsf{Enc}(k_i, m)$

ReKeyGen$(i, j, \widetilde{C})$
    if $\mathsf{Invalid}_{\mathsf{RE}}(i, j, \widetilde{C})$ then return $\perp$
    $\Delta_{i,j,\widetilde{C}} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
    return $\Delta_{i,j,\widetilde{C}}$

ReEnc$(i, j, C)$
    $\Delta_{i,j,\widetilde{C}} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C})$
    $C' = (\widetilde{C}', \overline{C}') \leftarrow \mathsf{ReEnc}(\Delta_{i,j,\widetilde{C}}, C)$
    if $\mathsf{Invalid}_{\mathsf{RE}}(i, j, \widetilde{C})$ then return $\widetilde{C}'$
    else return $C'$

ReLR$(i, j, C_0, C_1)$
    if $j \leqslant t$ then return $\perp$
    for $\beta \in \{0,1\}$ do
        $\Delta_{i,j,\widetilde{C}_\beta} \leftarrow\$ \mathsf{RekeyGen}(k_i, k_j, \widetilde{C}_\beta)$
        $C'_\beta \leftarrow \mathsf{ReEnc}(\Delta_{i,j,\widetilde{C}_\beta}, C_\beta)$
        if $C'_\beta = \perp$ then return $\perp$
    return $C'_b$
```

Figure 4.5: The game used to define re-encryption indistinguishability, UP-RE.

ReCrypt requires a key homomorphic PRF. We use the definition from [23] below.

Definition 4.13 (Key-homomorphic PRF). *Let $(\mathcal{K}, \oplus)$ and $(\mathcal{X}, \otimes)$ each be groups; let F be a function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$. We say F is a* key-homomorphic PRF *if $\forall k_1, k_2 \in \mathcal{K}$, $\forall x \in \mathcal{X}$:*

$$F_{k_1}(x) \otimes F_{k_2}(x) = F_{k_1 \oplus k_2}(x).$$

To define security for the key-homomorphic PRF, let game $\mathrm{PRF1}_F^{\mathcal{A}}$ be the game that selects a key $k \leftarrow\$ \mathcal{K}$ and then runs an adversary $\mathcal{A}$ that can adaptively query to an oracle that returns F_k applied to the queried message. The adversary outputs a bit. Let game $\mathrm{PRF0}_F^{\mathcal{A}}$ be the game in which an adversary $\mathcal{A}$ can adaptively query an oracle that returns a random draw from $\mathcal{Y}$. The adversary outputs a bit. We assume that $\mathcal{A}$, in either game, never queries the same value twice to its oracle. We define the PRF advantage of $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{A}) = \left|\Pr\left[\mathrm{PRF1}^{\mathcal{A}}_{\mathsf{F}} \Rightarrow 1\right] - \Pr\left[\mathrm{PRF0}^{\mathcal{A}}_{\mathsf{F}} \Rightarrow 1\right]\right| \ .$$

A simple example of a secure key-homomorphic PRF in the random oracle model (ROM) is the function $\mathsf{F}_k(x) = k \cdot \mathsf{H}(x)$ where $\mathcal{Y} = \mathbb{G}$ is an additive group in which the decisional Diffie–Hellman assumption holds. This construction is from [78].

With preliminaries established, we specify ReCrypt in Figure 4.4. The scheme uses as primitives: a key homomorphic PRF $\mathsf{F} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, an authenticated encryption scheme $\pi = (\mathcal{K}g, \mathcal{E}, \mathcal{D})$, and a hash function $h : \{0,1\}^* \to \mathcal{Y}$ (that we model as a random oracle).

ReCrypt security. ReCrypt gives UP-IND, UP-INT, and UP-RE security. In each case, we reduce the security of ReCrypt to the security of the underlying primitives (π, F). That is, any efficient attack on ReCrypt can be translated into an efficient attack on one of these two primitives. Now we turn to formal statements of security and proofs of each. We begin with the confidentiality notion UP-IND.

Theorem 4.14 (ReCrypt is UP-IND secure)**.** *Let $\pi = (\mathcal{K}g, \mathcal{E}, \mathcal{D})$ be an authenticated encryption scheme, $\mathsf{F} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ a key homomorphic PRF, $h : \{0,1\}^* \to \mathcal{Y}$ a hash function, and Π the ReCrypt updatable authenticated encryption scheme defined in Figure 4.4. If $\mathcal{A}$ is an UP-IND adversary, then there exist adversaries $\mathcal{B}$, $\mathcal{C}$ for the AE and PRF security games (respectively) such that:*

$$\mathbf{Adv}^{\textit{up-ind}}_{\Pi,s,t}(\mathcal{A}) \leqslant 2t \cdot \mathbf{Adv}^{\textit{ae}}_{\pi}(\mathcal{B}) + 2 \cdot \mathbf{Adv}^{\textit{prf}}_{\mathsf{F}}(\mathcal{C})$$

for $s \geqslant 0$, $t > 0$.

The goal of this proof (and each of the proofs below) is to reduce attacks on the ReCrypt scheme to attacks on the underlying primitives (π, F, h). We use a "game-hopping" technique [36] that is quite common and reasonable to follow. We define a series of games, each with a minor change from the prior game; then we bound the advantage difference between these two games by showing that an UP-RE adversary that can do much better in the modified game permits concrete construction of another adversary that can win in either the AE-ROR or PRF game.

Proofs proceed in two parts: first replacing calls to the AE scheme π, then replacing calls to the PRF F. In the first part, we start by replacing re-encryption ciphertexts with random strings. We show that an adversary that can distinguish between valid ciphertexts and random strings permits construction of an adversary ($\mathcal{B}$) that can win the AE security game. This requires s "game-hops", one for each rekey.

In the second part, we replace calls to the PRF F with calls to an oracle in the PRF security game. An adversary that can win this game permits construction of an adversary $\mathcal{C}$ that can win in the PRF security game. With all calls to the underlying schemes reduced to either random strings or calls to other oracles, our proof concludes and we have the stated result. Now we turn to the formal proof.

Proof. As mentioned, the first part of the proof uses the AE security of π to show that the value of x used to key the key-homomorphic PRF is hidden from the adversary.

Let G_0 be the original UP-IND game. For $1 \leqslant i \leqslant t$, Game G_i is the same as G_{i-1}, except we replace the encryption $\mathcal{E}(k_i, (\chi, h(m) + F(x, 0)))$ with a random string r and store the tuple (x, y, m) as a lookup in the table $\mathbb{C}$ indexed by the random string r. For queries to $\mathsf{RekeyGen}(i, j, \widetilde{C})$, if $\widetilde{C} = r$ for some previously returned r, then compute x', y' as usual and either:

- if $j \leqslant i$, return (r', x', y') for some random r', storing the value $(x + x', y + y', m)$, or
- if $j > i$, return $(\mathcal{E}(k_j, (\chi', h(m) + F(x + x', 0))), x', y')$.

If $\widetilde{C}$ has not been previously seen, return $\perp$. These modifications are shown in Figure 4.6.

Let S_i be the event that $\mathcal{A}$ outputs the correct bit in game G_i. Then we can construct an adversary $\mathcal{B}$ such that $|\Pr[S_i] - \Pr[S_{i-1}]| \leqslant \mathbf{Adv}^{\mathrm{ae}}_{\pi}(\mathcal{B})$. To see this, notice that by replacing the $\mathcal{E}(k_i, \cdot)$ and $\mathcal{D}(k_i, \cdot)$ functions with calls to an instance of the AE game for Π, in the real world, we get G_{i-1}, and the random world corresponds to G_i.

In game G_t, suppose the attacker queries the LR oracle to receive a challenge ciphertext C. Then the header $\widetilde{C}$ will be equal to some random string r, while the body $\overline{C}$ consists of the key mask y and the message encrypted by the PRF F keyed by some value x. Note that the use of AE forces the adversary to submit only previously seen ciphertext headers to oracles with $i \leqslant t$.

From re-keygen queries, the attacker can learn the fresh values of x', y', and r'. However, these are insufficient to learn anything about the value of x used to encrypt the message.

Similarly, re-encryption queries where the target key j is uncompromised, i.e., $j \leqslant t$, the adversary receives a fresh ciphertext encrypted under x' with no way of learning x'.

On the other hand, if $j > s$, the invalidity condition $\mathsf{Invalid}_{\mathsf{RE}}$ will be true, so the adversary learns the ciphertext header. This is of the form $\mathcal{E}(k_j, (x + y + x' + y', h(m) + F(x + x', 0)))$. Since the adversary possesses k_j, they may decrypt the ciphertext header. However, x is still masked by y.

Now for the second part of our proof: it uses the PRF property of F to show that indistinguishability holds, given that the adversary cannot learn x. We note this technique is similar to that used in [24]. We construct a game G_{t+1} such that $|\Pr[S_{t+1}] - \Pr[S_t]| \leqslant \mathbf{Adv}^{\mathrm{prf}}_{F}(\mathcal{C})$.

When $i \leqslant t$, we replace the usage of $F(x, l)$ in the LR oracle by using the PRF challenge oracle, denoted $f(l)$. Suppose K is the key used by the PRF challenger. When the PRF challenge is from the real world, the computed encryption is of the form $m_l + F(x, l) + f(l) = m_l + F(K + x, l)$. While the ciphertext header includes the value $\tau = h(m) + F(K + x, 0)$.

Since we showed that the value of x is unknown to the adversary for challenge ciphertexts, this change perfectly models the game G_t.

Furthermore, the computation of re-keying tokens and re-encryptions does not remove the PRF mask f from the ciphertext: if j is uncorrupted, then we compute $\overline{C}_l + F(x', l) = m_l + F(x + x' + K, l)$; otherwise, $\mathsf{Invalid}_{\mathsf{RE}}$ will be false, and the oracle returns $\perp$ for the ciphertext body. The adversary can also obtain the ciphertext header containing an encryption of $(x + x', \tau = h(m) + F(K + x + x', 0))$.

Now, let G_{t+1} correspond to the event that the PRF challenge returns a random value. Then the message is always perfectly masked by the output of $f(l)$ – we showed earlier that this value is present even after a re-encryption is computed. Therefore both the message and the hash are perfectly masked by the PRF values, and the adversary cannot win with a probability greater than $\frac{1}{2}$. Hence, we conclude that:

$$\begin{aligned} \mathbf{Adv}^{\text{up-ind}}_{\Pi,s,t}(\mathcal{A}) &= |2 \cdot \Pr[S_0] - 1| \\ &\leqslant 2t \cdot \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B}) + 2 \cdot \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{C}) \end{aligned}$$

for all $s \geqslant 0, t > 1$. □

Next we show that ReCrypt gives integrity in the sense of UP-INT. The proof uses the same game transformations and the same two-part structure; relying on the AE security of π but also requires that we model the hash function h as random oracle.

Theorem 4.15 (ReCrypt is UP-INT secure)**.** *Let $\pi = (\mathcal{K}g, \mathcal{E}, \mathcal{D})$ be an AE scheme, $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ a key homomorphic PRF, h a cryptographic hash function modelled as a random oracle with outputs in $\mathcal{Y}$, and $\mathcal{A}$ an adversary in the UP-INT security game. If Π is the ReCrypt scheme then there exist an adver-*

```
$\mathsf{Enc}(i, m)$
$x, y \leftarrow\$ \mathcal{K}$
$r \leftarrow\$ \{0,1\}^{|\widetilde{C}|}$
$\mathbb{C}_i[r] = (x, y, m)$
$\overline{C}_0 = y$
for $1 \leqslant l \leqslant \ell$
    $\overline{C}_l = m_l + F(x, l)$
return $(\widetilde{C} = r, \overline{C})$
```

```
$\mathsf{RekeyGen}(i, j, \widetilde{C})$
$(x, y, m) \leftarrow \mathbb{C}[\widetilde{C}]$
if $(x, y, m) = \perp$ return $\perp$
$x', y' \leftarrow\$ \mathcal{K}$
$\chi' = x + y + x' + y'$
if $j \leqslant i$ then
    $r \leftarrow\$ \{0,1\}^{|\widetilde{C}|}$
    $\mathbb{C}_j[r] = (x + x', y + y', m)$
    $\widetilde{C}' = r$
else
    $\tau = h(m) + F(x + x', 0)$
    $\widetilde{C}' \leftarrow\$ \mathcal{E}(k_j, (\chi', \tau))$
return $\Delta_{i,j,\widetilde{C}} = (\widetilde{C}', x', y')$
```

```
$\mathsf{Dec}(i, (\widetilde{C}, \overline{C}))$
$(x, y, m) \leftarrow \mathbb{C}_i[\widetilde{C}]$
if $(x, y, m) = \perp$ return $\perp$
$y' = \overline{C}_0$
$x' = x + y - y'$
for $1 \leqslant l \leqslant \ell$
    $m'_l = \overline{C}_l - F(x', l)$
$\tau = h(m) + F(x, 0)$
if $\tau - F(x', 0) = h(m')$ then
    return $m = (m_1, \ldots, m_\ell)$
else
    return $\perp$
```

Figure 4.6: The replacement algorithms used in game G_i for the proofs of security for the ReCrypt construction. For the i-th key, encryption/decryption by Π is replaced by random strings r of the same length and a lookup.

sary $\mathcal{B}$ for the AE security game that makes q_h queries to h and q oracle queries where:

$$\mathbf{Adv}^{\textit{up-int}}_{\Pi,s,t}(\mathcal{A}) \leqslant 2t \cdot \mathbf{Adv}^{\textit{ae}}_{\pi}(\mathcal{B}) + \frac{q^2 + q_h^2}{|\mathcal{Y}|} + \frac{q^2}{|\mathcal{X}| \cdot |\mathcal{Y}|}$$

for $s \geqslant 0, t > 0$.

Proof. We begin in the same manner as the previous proof: apply t transformations as shown in Figure 4.6, substituting encryption by π with random strings for uncorrupted keys. Decryption is then simulated by using a lookup of previously returned values. Let S_i be the event that $\mathcal{A}$ outputs the correct bit in game G_i.

In game G_{t+1}, we replace the Dec verification check $\tau - F(x', 0) = h(m')$ and instead directly verify that $m' = m$. These two games are identical, unless the adversary submits for verification a ciphertext $C = (\widetilde{C}, \overline{C})$ such that $\widetilde{C}$ has previously been generated for the tuple (x, y, m) and $\overline{C}$, decrypts using $\chi - y$ to a message m' such that $m' \neq m$ but $h(m) + F(x, 0) - F(x + y - y') = h(m')$. That is, $h(m) + F(y - y', 0) = h(m')$. For all adversarially chosen y', the adversary needs to construct a ciphertext $\overline{C}'$ such that $h(\overline{C}'_1 - F(x + y - y', 1), \ldots, \overline{C}'_\ell - F(x + y - y', \ell))$ is equal to some fixed value $h(m) - F(y - y', 0)$. This is equivalent to the pre-image resistance of h. Alternatively, if the adversary does not modify y, then this reduces to finding a value of m' such that $h(m) = h(m')$, which is just the collision resistance of h.

By modelling h as a random oracle, we can bound the probability that the adversary making q_h queries to the random oracle, succeeds in finding suitable values of y', m, and m' by $q_h^2/|\mathcal{Y}|$. Therefore $|\Pr[S_{t+1}] - \Pr[S_t]| \leqslant q_h^2/|\mathcal{Y}|$.

Let (i, C) be a tuple submitted by the adversary to the Try oracle. We show that with high probability, one of the following must be true: either $\mathsf{InvalidCTXT}(i, C) = 1$ or $\mathsf{Dec}(k_i, C) = \perp$. By the first t game hops, we know that $\mathcal{D}(k_i, \widetilde{C}) = \perp$ unless $\widetilde{C}$ was previously output by an oracle query (whether Enc or ReKeyGen). Furthermore, for all $\widetilde{C}$, with high probability, there exists precisely one $\overline{C}$. For $1 \leqslant l \leqslant \ell$, decryption is computed as $\overline{C}_l + F(x, l)$. Therefore, any modification to a ciphertext block will result in a distinct message, which will result in a failed decryption due to the equality check. The exception to this is when the same random value $\widetilde{C}$ was generated for distinct messages. If the adversary makes a total of q queries, this happens with probability:

$$q^2/2^{|\widetilde{C}|} \leqslant q^2/2^{|h(x)|+|x|} \leqslant q^2/2^{\log|\mathcal{X}|+\log|\mathcal{Y}|} \leqslant q^2/(|\mathcal{X}| \cdot |\mathcal{Y}|).$$

This shows that for each $\widetilde{C}$, there can only be one $\overline{C}$. But what about the converse? For a given ciphertext body $\overline{C}$, suppose there exists (x', m') such that $\overline{C}_l = m_l + F(x, l) = m'_l + F(x', l)$. Therefore $F(x' - x, l) = m_l - m'_l$. However, m' was submitted to the (re-)encryption oracle before x' was generated. Therefore, for any pair $(x, m), (x', m')$ the probability that

$\mathsf{F}(x' - x, l) = m_l - m'_l$ is $1/|\mathcal{Y}|$, and thus if the adversary makes q queries, the probability is bounded by $q^2/|\mathcal{Y}|$.

Together with the probability that two $\widetilde{C}$ are equal, this becomes the probability the adversary wins in game G_{t+1}, otherwise we have established that all valid ciphertext headers have precisely one matching ciphertext body, in which case $\mathsf{InvalidCTXT}(i, C)$ is true.

Therefore, the advantage of $\mathcal{A}$ is:

$$\mathbf{Adv}^{\text{up-int}}_{\Pi,s,t}(\mathcal{A}) \leqslant t \cdot \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B}) + \frac{q^2 + q_h^2}{|\mathcal{Y}|} + \frac{q^2}{|\mathcal{X}| \cdot |\mathcal{Y}|} . \qquad \square$$

Finally, we prove that ReCrypt meets our re-encryption indistinguishability notion UP-RE.

Theorem 4.16 (ReCrypt is UP-RE secure)**.** *Let* $\pi = (\mathcal{K}g, \mathcal{E}, \mathcal{D})$ *be an AE scheme,* $\mathsf{F} L\mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ *a key homomorphic PRF, and* $\mathcal{A}$ *an adversary in the UP-RE security game. If* Π *is the ReCrypt scheme then there exist adversaries* $\mathcal{B}, \mathcal{C}$ *such that:*

$$\mathbf{Adv}^{\textit{up-re}}_{\Pi,s,t}(\mathcal{A}) \leqslant 2t \cdot \mathbf{Adv}^{\textit{ae}}_{\pi}(\mathcal{B}) + 2 \cdot \mathbf{Adv}^{\textit{prf}}_{\mathsf{F}}(\mathcal{C})$$

for $s \geqslant 0, t > 0$.

Proof. Again we begin by applying the same t steps as in the previous proof, shown in Figure 4.6. Then we have the ciphertext headers replaced with random strings, and the re-keygen process is simulated by recording inputs previously seen.

Hence, in game G_t, the adversary cannot learn anything about the PRF key x used in the challenge, nor does the ciphertext header reveal anything for compromised keys, since all values are masked by values stored in the ciphertext body.

As in the UP-IND proof, we construct a PRF adversary $\mathcal{C}$ in the second stage of our proof. In game G_t we replace the re-encryption routine from challenges to return: $(r, \overline{C}_b + f(l) + F(x' + y', l))$. That is, the ciphertext header $\widetilde{C}$ is a random string, by the previous hybrids, and we re-encrypt the ciphertext body $\overline{C}_b$ to use (implicitly) key $\chi + K + x'$, where K is the key used by the PRF challenger. The re-keying token is now $(r, K + x', y')$. However, since we do not return this to the adversary, we do not need to compute it, nor does the adversary learn $K + x'$. Any further re-encryption or re-keygen queries by the adversary preserves the usage of $f(l)$ as a mask.

Now we let game G_{t+1} be the scenario in which the PRF challenger returns random values, and we conclude that the adversary cannot learn any information about the bit b.

The same computation as before results in:

$$\mathbf{Adv}^{\text{up-re}}_{\Pi,s,t}(\mathcal{A}) \leqslant 2t \cdot \mathbf{Adv}^{\text{ae}}_{\pi}(\mathcal{B}) + 2 \cdot \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{C}) \ . \qquad \square$$

Relationship between notions. An interesting open question is whether UP-RE security implies UP-IND and UP-INT security. In the case of (non-updatable) authenticated encryption, the strongest property, AE, implies the individual confidentiality and integrity notions. The same property would be useful here and would short-cut analysis of new schemes. It seems plausible that UP-RE automatically implies UP-IND and UP-INT; and if not, understanding the separating circumstances might be critical to improving those definitions and understanding the gap between UP-RE and the weaker notions. We leave this interesting question to future work.

4.4 ReCrypt Implementation and Evaluation

In this section we detail a concrete instantiation of ReCrypt and report on the performance of a prototype implementation.

ReCrypt Operation	Time per CPU 1 block	1 KB	1 MB	1 GB	cycles/byte
Encrypt	353 µs	8.5 ms	8.9 s	2.5 hours	30.7 K
ReEnc	239 µs	7.3 ms	7.2 s	2.0 hours	26.3 K
Decrypt	328 µs	7.7 ms	7.5 s	2.2 hours	27.9 K
ReKeyGen (total)	178 µs				2.3 M

Table 4.2: Processing times for ReCrypt operations. 1 block represents any plaintext not larger than 31 bytes.

We use the random oracle model (ROM) [79] to establish our key-homomorphic PRF for ReCrypt. Let $\mathsf{H} : \mathcal{X} \to \mathbb{G}$ be a hash function (modeled as a random oracle) where $(\mathbb{G}, +)$ is a group in which the decisional Diffie-Hellman (DDH) assumption holds. $\mathbb{F}_p$

We select $\mathsf{F}(k, x) = k \cdot \mathsf{H}(x)$ as our key-homomorphic PRF and choose $\mathbb{G} = \mathsf{E}(\mathbb{F}_p)$, an elliptic curve over a prime order finite field as our group.

For the AE scheme π, the prototype uses AES in Galois Counter Mode (AES-GCM) with 128 bit keys and SHA256 for the hash function h. It uses the elliptic curve Curve25519 [15], which has $p = 2^{255} - 19$. For the hash function H which hashes bitstrings to elliptic curve elements, the process is more complicated. Bitstrings are first hashed with h (SHA256) and then encoded with the Elligator encoding function [17] to produce elliptic curve points (elements in group $\mathbb{G}$). More details about the complexity and security of this encoding is described in [44].

Microbenchmarks. Our implementation is single-threaded and we measured performance on an Intel CPU (Haswell) running at 3.8GHz. We used the following number of iterations for each measurement: 1000 (for 1 block, 1 KB), 100 (for 1 MB) and 1 (for 1 GB). Cycles per byte was computed with 1 MB ciphertexts.

Table 4.2 shows wall clock times for ReCrypt operations over various plaintext sizes. ReCrypt operations are essentially public-key operations and so are not competitive in terms of performance compared to traditional (symmetric) authenticated encryption operations such as AES-GCM. For comparison, AES-GCM on the same evaluation platform performs encryption as follows: 1 block (15 µs), 1 KB (24 µs), 1 MB (9 ms), 1 GB (11 s). The weaker scheme, KSS, has performance that matches AES-GCM, whereas ReCrypt scheme is determined by scalar multiplications required to evaluate the PRF: giving a $1000\times$ performance cost to achieve our strongest notion of security for updatable encryption.

Applications. Given the performance difference, ReCrypt is best suited to very small or very valuable plaintexts (ideally, both). Given this, we briefly consider two application settings for updatable AE.

For privacy and security reasons, regulation mandates that credit card numbers must be stored in encrypted form [83]. These include the recommendation that a mechanism must be in place to rotate keys on a regular basis and in the face of known or suspected compromise. Given the sensitive nature of payment information, the naive solution of decrypting and encrypting the data to rotate keys exposes it to some risk, since it makes the data available in plaintext form for at least a period of time. Similarly, NIST guidelines [9] recommend balancing the risk introduced by the re-encryption process with the benefits offered by key rotation. On the other hand, our updatable AE schemes enable secure rotation of keys using an untrusted storage service.

Consider a payment system with, say, 1 billion credit card entries. The storage required for encrypted credit card numbers using ReCrypt is about 30 GB (assuming one block per entry). Projecting from performance measurements in Table 4.2, a full key rotation across the this dataset requires 60 CPU-hours: a significant amount of computation, but potentially not prohibitive given infrequent rotations and many available CPUs. The time to decrypt a single entry is sub-millisecond; this is small compared to processing time for a credit card transaction.

As a second example we consider long-term storage of static data, commonly known as “deep” or “cold” storage. Such data is accessed infrequently, yet data owners may still desire (or be required to) periodically rotate the encryption keys used for protecting the data. In such cases it may be more convenient for the data owner to allow the storage provider to rotate the encryption keys using a system local to the data, as opposed to the data owner retrieving the data and performing the re-encryption.

For a rough estimation of costs, we compute the cost of performing re-encryption using ReCrypt on Amazon Web Service’s Elastic Compute Cloud (AWS EC2). Using the price per CPU-hour of an AWS EC2 instance of 0.05 $/hour (USD)[3], we compute the cost to perform updates using ReCrypt as 0.10 $/GB. For small- to medium-sized data sets or for data sets that are particularly valuable (e.g. financial information), this cost may be justified. However, for moderately-sized to large data sets, the cost may be prohibitive and clients may favor basic security schemes like KSS.

4.5 Related Work

We have mentioned related work elsewhere in this chapter, but updatable AE’s relationship to proxy re-encryption deserves further discussion.

Updatable AE appears similar in definition to proxy re-encryption but critical distinctions remain: first definitional differences, then security differences. In proxy re-encryption, a proxy transforms a ciphertext created under an initial key into a ciphertext that can be decrypted by another key. Similar to updatable AE, this is accomplished with the use of an update token derived from the two keys. In the existing definitions, proxy re-encryption schemes are *ciphertext independent*; that is, update tokens

[3]This rate is based on the lowest per-CPU cost on AWS as of June 2017. Namely, an m4.16xlarge instance available in the AWS US-East (Ohio) region which provides 64 CPUs and is available on-demand for 3.20 $/hour.

work for any ciphertexts whereas our updatable AE definition permits update tokens to be bound to a single ciphertext.

Further, proxy re-encryption targets a distinct security goal. Proxy re-encryption seeks no information leakage of the plaintext to the proxy (confidentiality) and preserving the integrity of the messages through decryption. However, all keys are expected to remain secret. In proxy re-encryption there is no expectation that the encryption key for initial ciphertext will be given to the adversary and so existing proxy re-encryption schemes target a distinct security notion. Indeed, the symmetric schemes of [54, 35] use a "double-encryption" in which an outer layer of encryption is removed and replaced with a new outer layer of encryption by the proxy while the inner-most encryption remains unmodified. From the world of public key cryptography, [32] provide a proxy re-encryption scheme that meets chosen-ciphertext security, but fails to meet our own UP-RE security.

4.6 Discussion

This chapter systematically studies updatable AE in the presence of compromised keys. It provides a sequence of security notions meeting different real-world security requirements and schemes that satisfy them efficiently. Along the way, we demonstrate the limitations of current approaches, as represented by AE-hybrid, improve it at low cost to obtain the KSS scheme meeting our confidentiality (UP-IND) and integrity (UP-INT) notions. We also give a stronger updatable encryption scheme, ReCrypt, that is secure under all of our security notions (UP-IND, UP-INT, and UP-RE, indistinguishable re-encryption). We implemented ReCrypt and present a basic performance evaluation for our prototype. The scheme is slower than the hybrid approaches but offers complete key rotation.

This work puts updatable AE on a firm theoretical foundation and brings schemes with improved security closer to industrial application. While there is a rich array of different security models for practitioners to chose from, it is clear that achieving strong security (currently) comes at a substantial price. Meanwhile, weaker but still useful security notions can be achieved at no performance penalty over conventional AE. It is an important challenge to find constructions which lower the cost compared to ReCrypt without reducing security. But it seems that fundamentally new ideas are needed: ReCrypt uses essentially public key operations to achieve an updatable, symmetric encryption.

From a theoretical perspective, it is of interest to study the exact relations between our security notions, in particular whether UP-RE is strong enough to imply UP-IND and UP-INT.

5 CONCLUSION

This work examined multiple security-critical mechanisms required for internet-scale services. We find that in many cases service developers have brought forward existing security mechanisms without sufficient care. In the context of internet-scale services, existing assumptions break and the security of these techniques no longer holds. In the case of operating system random number generators (RNGs), snapshot resumption breaks fundamental assumptions related the the RNG design and this results in reset vulnerabilities on multiple major platforms (FreeBSD, Windows, and Linux). For password-based authentication, the state-of-the-art techniques for hardening passwords are at a dead-end. Increasing the computational cost of hardening functions only slows down, but cannot prevent, offline dictionary attacks, and so we propose a new direction for securing password-based credentials. Encryption of data-at-rest in the world of third-party storage must account for key rotation and the current technique, authenticated encryption (AE) hybrid, is shown to have severe limitations–meeting none of our formal security definitions.

In each of these cases we have conceived, implemented, and evaluated valuable replacements. In the case of encrypting data-at-rest, we have proven the benefits formally. To fix random numbers we have given a new RNG design, Whirlwind. It prevents reset vulnerabilities, is amenable to formal analysis, and is built from principled design. For password-based authentication, we created Pythia, a new authentication scheme provided as a modern, cloud-based service. It represents an enormous increase in security, providing rate-limiting, encrypted credentials, and key rotation. Finally, we created the ReCrypt updatable encryption scheme that meets our strongest definitions of security for this setting.

Directions for future work. For operating system RNGs we have put forth a strong contender for improvement with Whirlwind. Whirlwind will benefit from formal evaluation under the definition of a PRNG with input. It was created with this mind and we sought to streamline the design to simplify formal analysis. We have evaluated it in the context of data center servers, but the Linux operating system serves a wide array of platforms. Performance profiling on these platforms will determine if it's a good contender to replace the legacy RNG, especially on embedded platforms where there may be considerable differences in performance and entropy between RNG designs.

With Pythia, we have provided detailed formal analysis on the password hardening protocol. Critically, Pythia also provides a key rotation procedure, but this has not yet received formal examination. This warrants careful analysis, insecurities are problematic. In particular, Pythia provides a trust-on-first-use when the client stores the Pythia server's public key and zero-knowledge proof. It is an open question whether or not a key rotation carries forward the same trust or if the client must begin again, resetting its trust after each key rotation.

We have established initial definitions for updatable authenticated encryption. We had sought to match them as well as we could to reasonable scenarios. To validate or improve these definitions requires a better understanding of real-world configurations and real-world threats. If our assumptions are too strong (or too weak), then they should be adjusted, which necessarily leads to new definitions and then new updatable encryption schemes.

Of particular interest is the gap between UP-RE and the combination of UP-IND and UP-INT. Does UP-RE imply both UP-IND and UP-INT? Is the relationship an equivalence? If so, the result is interesting; if not, the separating examples likely point to definitions that lie within the gap. And such a definition may lead to schemes that provide a hybrid level of security and performance. This would be enormously valuable. Currently, KSS provides excellent performance, but mediocre security; ReCrypt gives excellent security, but performance that is unacceptable for many uses.

Also of interest is whether or not hardware-acceleration can substantially improve elliptic curve cryptography (on which we build ReCrypt) and by how much. Each order of magnitude of performance improvement brings UP-RE security to a very large class of uses.

Next we consider future directions for internet-scale security in a broader context.

Lessons learned. In each setting, we have discovered some mismatch between the original assumptions of security mechanisms (e.g. no VM snapshots; local password hardening only) and the new context of internet-scale services (VM reset vulnerabilities; easily accessible external services). This applies to many more security mechanisms than those we have analyzed. Further, the context of internet-scale services offers new primitives for updating and building security mechanisms, including: fast networks, distributed processing power, and multiple security domains. We view this context as a rich design space for conceiving new, improved means of enhancing security and privacy.

It should also be noted that these powers are not limited only to defenders. Attackers also benefit from on-demand computational power, fast networks, and the increased connectivity of information systems. The threat landscape is evolving at the same pace as the defender's new abilities. Attackers won't sit idle, neither should the defenders.

REFERENCES

[1] Haveged entropy gatherer. `http://www.issihosts.com/haveged/`.

[2] Abe, Masayuki, and Tatsuaki Okamoto. 2000. Provably secure partially blind signatures. In *Advances in cryptology–crypto*. Springer.

[3] Akhawe, Devdatta. 2016. How dropbox securely stores your passwords. *Dropbox Tech Blog*.

[4] Aranha, D. F., and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[5] Arkin, Brad, Frank Hill, Scott Marks, Matt Schmid, Thomas John Walls, and Gary Mc-Graw. 1999. How we learned to cheat at online poker: A study in software security. The developer.com Journal.

[6] AWS. Protecting data using client-side encryption. `http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingClientSideEncryption.html`.

[7] Bagherzandi, Ali, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. 2011. Password-protected secret sharing. In *Computer and communications security*. ACM.

[8] Barak, Boaz, and Shai Halevi. 2005. A model and architecture for pseudo-random generation with applications to /dev/random. In *Computer and communications security — ccs*, 203–212. ACM.

[9] Barker, Elaine. 2016. Recommendation for key management Part 1: General. In *Nist special publication 800-57 part 1 revision 4*.

[10] Barreto, Paulo SLM, and Michael Naehrig. 2006. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*. Springer.

[11] Barroso, Luiz André, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8(3):1–154.

[12] Bellare, Mihir, Sriram Keelveedhi, and Thomas Ristenpart. 2013. Dupless: server-aided encryption for deduplicated storage. In *Usenix security*. USENIX.

[13] Bellare, Mihir, and Phillip Rogaway. 2006. The security of triple encryption and a framework for code-based game-playing proofs. In *Annual international conference on the theory and applications of cryptographic techniques*, 409–426. Springer.

[14] Bendel, Mike. Hackers describe PS3 security as epic fail, gain unrestricted access. `http://www.exophase.com/20540/hackers-describe-ps3-security-as-epic-fail-gain-unrestricted-access/`.

[15] Bernstein, Daniel J. Curve25519: New Diffie-Hellman speed records. 207–228.

[16] Bernstein, Daniel J, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. 2013. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *Advances in cryptology — asiacrypt 2013*, 341–360. Springer.

[17] Bernstein, Daniel J., Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. 967–980.

[18] Bitcoin addresses. Technical background of version 1 Bitcoin addresses. `https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses`.

[19] Bitcoin Weaknesses. Bitcoin wiki, “weaknesses”. `https://en.bitcoin.it/wiki/Weaknesses`.

[20] Blaze, Matt, Gerrit Bleumer, and Martin Strauss. 1998. Divertible protocols and atomic proxy cryptography. In *Advances in cryptology–eurocrypt*. Springer.

[21] Boldyreva, Alexandra. 2002. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *Public key cryptography*. Springer.

[22] Boneh, Dan, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. 2013. Key homomorphic PRFs and their applications. In *Advances in cryptology–crypto*. Springer.

[23] ———. 2013. Key homomorphic prfs and their applications. In *Advances in cryptology–crypto 2013*, 410–428. Springer.

[24] ———. 2015. Key homomorphic PRFs and their applications. Cryptology ePrint Archive, Report 2015/220. `http://eprint.iacr.org/2015/220`.

[25] Boneh, Dan, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *Advances in cryptology–asiacrypt*. Springer Berlin Heidelberg.

[26] Boneh, Dan, and Brent Waters. 2013. Constrained pseudorandom functions and their applications. In *Advances in cryptology-asiacrypt*. Springer.

[27] Brainwallet. Brainwallet. `https://en.bitcoin.it/wiki/Brainwallet`.

[28] Brodkin, Jon. 2016. Netflix finishes its massive migration to the amazon cloud. *Ars Technica*.

[29] Camenisch, Jan, Anna Lysyanskaya, and Gregory Neven. 2012. Practical yet universally composable two-server password-authenticated secret sharing. In *Computer and communications security*. ACM.

[30] Camenisch, Jan, Gregory Neven, and abhi shelat. 2007. Simulatable adaptive oblivious transfer. In *Advances in cryptology–eurocrypt*. Springer Berlin Heidelberg.

[31] Camenisch, Jan, and Markus Stadler. 1997. Proof systems for general statements about discrete logarithms. Technical Report No. 260, Dept. of Computer Science, ETH Zurich.

[32] Canetti, Ran, and Susan Hohenberger. 2007. Chosen-ciphertext secure proxy re-encryption. In *Acm ccs 07*, ed. Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, 185–194. ACM Press.

[33] Chaum, David. 1983. Blind signatures for untraceable payments. In *Advances in cryptology*. Springer.

[34] Chaum, David, and Torben Pryds Pedersen. 1993. Wallet databases with observers. In *Advances in cryptology–crypto*. Springer.

[35] Cool, DL, and Angelos D Keromytis. 2005. Conversion and proxy functions for symmetric key ciphers. In *Information technology: Coding and computing, 2005. itcc 2005. international conference on*, vol. 1, 662–667. IEEE.

[36] Dent, Alexander W. 2006. A note on game-hopping proofs. *IACR Cryptology ePrint Archive* 2006:260.

[37] Di Raimondo, Mario, and Rosario Gennaro. 2003. Provably secure threshold password-authenticated key exchange. In *Advances in cryptology–eurocrypt*. Springer.

[38] Dodis, Yevgeniy. 2002. Efficient construction of (distributed) verifiable random functions. In *Public key cryptography*. Springer.

[39] Dodis, Yevgeniy, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. 2013. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 acm sigsac conference on computer & communications security*, 647–658. ACM.

[40] Dodis, Yevgeniy, and Aleksandr Yampolskiy. 2005. A verifiable random function with short proofs and keys. In *Public key cryptography*. Springer.

[41] Dorrendorf, Leo, Zvi Gutterman, and Benny Pinkas. 2009. Cryptanalysis of the random number generator of the Windows operating system. *ACM Transactions on Information and System Security (TISSEC)* 13(1):10.

[42] ElGamal, Taher. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology–crypto*. Springer.

[43] Everspaugh, Adam, Rahul Chatterjee, Samuel Scott, Ari Juels, Thomas Ristenpart, and Cornell Tech. 2015. The pythia prf service. *IACR Cryptology ePrint Archive* 2015:644.

[44] Everspaugh, Adam, Kenneth Paterson, Thomas Ristenpart, and Sam Scott. 2017. Key rotation for authenticated encryption. Cryptology ePrint Archive, Report 2017/527. `http://eprint.iacr.org/2017/527`.

[45] Ford, Warwick, and Burton S. Kaliski, Jr. 2000. Server-assisted generation of a strong secret from a password. In *International workshops on enabling technologies: Infrastructure for collaborative enterprises*. IEEE.

[46] Freedman, Michael J, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword search and oblivious pseudorandom functions. In *Theory of cryptography*. Springer.

[47] Garfinkel, Tal, and Mendel Rosenblum. 2005. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Workshop on Hot Topics in Operating Systems — HotOS-X*.

[48] GCP. Managing data encryption. `https://cloud.google.com/storage/docs/encryption`.

[49] Goichon, François, Cédric Lauradoux, Guillaume Salagnac, and Thibaut Vuillemin. 2012. Entropy transfers in the Linux random number generator. Research Report RR-8060, INRIA.

[50] Goldberg, Ian, and David Wagner. 1996. Randomness and the Netscape browser. *Dr Dobb's Journal* 66–71.

[51] Gupta, Akhil. 2016. Scaling to exabytes and beyond. *Dropbox Tech Blog*.

[52] Gutterman, Zvi, Benny Pinkas, and Tzachy Reinman. 2006. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy*, 371–385. IEEE.

[53] Heninger, Nadia, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Usenix security*, 205–220. USENIX.

[54] Ivan, Anca, and Yevgeniy Dodis. 2003. Proxy cryptography revisited. In *Ndss 2003*. The Internet Society.

[55] Jarecki, Stanislaw, Aggelos Kiayias, and Hugo Krawczyk. 2014. Round-optimal password-protected secret sharing and t-PAKE in the password-only model. In *Advances in cryptology–asiacrypt*. Springer.

[56] Kelsey, John, Bruce Schneier, and Niels Ferguson. 2000. Yarrow-160: Notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *Selected areas in cryptography*, 13–33. Springer.

[57] Kerrigan, Brendan, and Yu Chen. 2012. A study of entropy sources in cloud computers: Random number generation on cloud hosts. In *Computer network security*, 286–298. Springer.

[58] Kerrisk, Michael. 2012. LCE: Don't play dice with random numbers. `https://lwn.net/Articles/525459/`.

[59] Kim, Soo Hyeon, Daewan Han, and Dong Hoon Lee. 2013. Predictability of Android OpenSSL's pseudo random number generator. In *Computer and communications security — ccs*, 659–668. ACM.

[60] Krawczyk, Hugo, Ran Canetti, and Mihir Bellare. 1997. Hmac: Keyed-hashing for message authentication.

[61] Krohn, Max, and Chris Coyne. Wrap Wallet. `https://keybase.io/warp`.

[62] Lacharme, Patrick, Andrea RÃ¶ck, Vincent Strubel, and Marion Videau. 2012. The Linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251. `http://eprint.iacr.org/`.

[63] Liskov, Moses, Ronald L Rivest, and David Wagner. 2002. Tweakable block ciphers. In *Advances in cryptology–crypto*. Springer.

[64] Lysyanskaya, Anna. 2002. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in cryptology–crypto*. Springer.

[65] MacKenzie, Philip, and Michael K Reiter. 2003. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing* 16(4).

[66] ———. 2003. Networked cryptographic devices resilient to capture. *International Journal of Information Security* 2(1).

[67] MacKenzie, Philip, Thomas Shrimpton, and Markus Jakobsson. 2002. Threshold password-authenticated key exchange. In *Advances in cryptology–crypto*. Springer.

[68] Mc Guire, Nicholas, Peter Odhiambo Okech, and Qingguo Zhou. 2009. Analysis of inherent randomness of the Linux kernel. In *Real time linux workshop*.

[69] McEvoy, Robert, James Curran, Paul Cotter, and Colin Murphy. 2006. Fortuna: Cryptographically secure pseudo-random number generation in software and hardware. In *Irish signals and systems conference*, 457–462. IET.

[70] McMillan, R. 2014. The inside story of Mt. Gox, bitcoin's $460 million disaster. *Wired*.

[71] McVoy, Larry W, and Carl Staelin. 1996. lmbench: Portable tools for performance analysis. In *Usenix annual technical conference*, 279–294. San Diego, CA, USA.

[72] Micali, Silvio, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *Foundations of computer science*. IEEE.

[73] Micali, Silvio, and Ray Sidney. 1995. A simple method for generating and sharing pseudo-random functions, with applications to clipper-like key escrow systems. In *Advances in cryptology–crypto*. Springer.

[74] Mowery, Keaton, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson. 2013. Welcome to the Entropics: Boot-time entropy in embedded devices. In *IEEE Symposium on Security and Privacy*, 589–603. IEEE.

[75] Muffet, Alec. 2015. Facebook: Password hashing & authentication. Presentation at Real World Crypto.

[76] Müller, Stephen. 2013. CPU time jitter based non-physical true random number generator.

[77] Naor, Moni, Benny Pinkas, and Omer Reingold. 1999. Distributed pseudo-random functions and KDCs. In *Advances in cryptology–eurocrypt*. Springer.

[78] ———. 1999. Distributed pseudo-random functions and KDCs. In *Eurocrypt'99*, ed. Jacques Stern, vol. 1592 of *LNCS*, 327–346. Springer, Heidelberg.

[79] ———. 1999. Distributed pseudo-random functions and kdcs. In *International conference on the theory and applications of cryptographic techniques*, 327–346. Springer.

[80] National Institute of Standards and Technology. 2002. Federal information processing standards publication 180-2: Secure hash standard. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.

[81] Pagliery, Jose. 2016. Hackers selling 117 million linkedin passwords. *CNN*.

[82] Paterson, Kenneth G, and Sriramkrishnan Srinivasan. 2009. On the relations between non-interactive key distribution, identity-based encryption and trapdoor discrete log groups. *Designs, Codes and Cryptography* 52(2).

[83] PCI Security Standards Council. 2016. Requirements and security assessment procedures. In *PCI DSS v3.2*.

[84] Pointcheval, David, and Jacques Stern. 1996. Provably secure blind signature schemes. In *Advances in cryptology–asiacrypt*. Springer.

[85] Ristenpart, Thomas, and Scott Yilek. 2010. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Network and distributed systems security — ndss*. ISOC.

[86] Rogaway, Phillip. 2004. Nonce-based symmetric encryption. In *Fast software encryption*, 348–358. Springer.

[87] Rogaway, Phillip, and Thomas Shrimpton. 2006. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. Cryptology ePrint Archive, Report 2006/221. `http://eprint.iacr.org/2006/221`.

[88] ———. 2006. A provable-security treatment of the key-wrap problem. In *Annual international conference on the theory and applications of cryptographic techniques*, 373–390. Springer.

[89] Roskind, Jim. 2013. QUIC: Multiplexed stream transport over UDP. *Google working design document*.

[90] Sakai, R., K. Ohgishi, and M. Kasahara. 2000. Cryptosystems based on pairing. In *Cryptography and information security*.

[91] Shah, Amit. 2013. About random numbers and virtual machines. `http://log.amitshah.net/2013/01/about-random-numbers-and-virtual-machines/`.

[92] Shamir, Adi. 1979. How to share a secret. *Communications of the ACM* 22(11):612–613.

[93] Stamos, Alex, Andrew Becherer, and Nathan Wilcox. 2009. Cloud computing models and vulnerabilities: Raining on the trendy new parade. *BlackHat USA*.

[94] Stephens, Michael A. 1970. Use of the Kolmogorov-Smirnov, Cramér-Von Mises and related statistics without extensive tables. *Journal of the Royal Statistical Society. Series B (Methodological)* 115–122.

[95] Thielman, Sam. 2016. Yahoo hack: 1bn accounts compromised by biggest data breach in history. *The Guardian*.

[96] Thompson, Christopher J, Ian J De Silva, Marie D Manner, Michael T Foley, and Paul E Baxter. 2011. Randomness exposed — an attack on hosted virtual machines.

[97] Winternitz, Robert S. 1984. A secure one-way hash function built from DES. In *IEEE Symposium on Security and Privacy*, 88–90.

[98] Yilek, Scott, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Sigcomm conference on internet measurement*, 15–27. ACM.