# When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography

Thomas Ristenpart Scott Yilek

Dept. of Computer Science and Engineering
University of California, San Diego, USA
{tristenp,syilek}@cs.ucsd.edu

## Abstract

*Random number generators (RNGs) are consistently a weak link in the secure use of cryptography. Routine cryptographic operations such as encryption and signing can fail spectacularly given predictable or repeated randomness, even when using good long-lived key material. This has proved problematic in prior settings when RNG implementation bugs, poor design, or low-entropy sources have resulted in predictable randomness. We investigate a new way in which RNGs fail due to reuse of virtual machine (VM) snapshots. We exhibit such VM reset vulnerabilities in widely-used TLS clients and servers: the attacker takes advantage of (or forces) snapshot replay to compromise sessions or even expose a server's DSA signing key. Our next contribution is a backwards-compatible framework for hedging routine cryptographic operations against bad randomness, thereby mitigating the damage due to randomness failures. We apply our framework to the OpenSSL library and experimentally confirm that it has little overhead.*

## 1. Introduction

The security of routine cryptographic operations such as encryption, key exchange, and randomized signing rely on access to good (unpredictable, fresh) randomness. Unfortunately, the random number generators (RNGs) used in practice frequently fail. Examples stem from poorly designed RNGs [28, 32, 35], implementation bugs [13, 51], untimely exposure of randomness [17], and even the inability to find sufficient entropy in a system's environment [36]. Since deployed cryptographic routines provide no security given bad randomness (even when using good long-lived keys), the attacks that result from RNG failure are spectacular [13, 16, 32, 35, 53].

In this work, we first show a new way in which deployed RNGs fail, due to virtual machine (VM) resets. Beyond their relevance to RNG failures, the vulnerabilities are also interesting because they introduce a potentially-widespread class of practical problems due to virtualization technologies. Our second contribution is a general and backwards-compatible framework to hedge against the threat of randomness failures in deployed cryptographic algorithms. We discuss each contribution more in turn.

**VM reset vulnerabilities.** Virtualization technologies enable significant flexibility in handling the state of guest systems (an operating system, user applications, and data). In particular, virtual machine (VM) snapshots, i.e. copies of the state of the guest, can be used to replicate, backup, transfer (to another physical system), or reset (to a prior state) the guest. Snapshots are one reason virtualization is transforming numerous areas of computing. However, Garfinkel and Rosenblum [31] suggest that, in theory, snapshots might lead to security problems due to reuse of security-critical state. Namely, reusing a VM snapshot might lead to (what we call) VM reset vulnerabilities. But no insecurities have been reported for real systems, leaving open the question of whether reset vulnerabilities are a practical problem.

We answer this question by revealing exploitable VM reset vulnerabilities within popular software. Our attacks are against TLS [8] implementations used for secure web browsing and work when a victim VM runs twice from the same snapshot. We investigate both TLS clients and servers, presenting session-compromise attacks against clients in the Firefox and Chrome web browsers and secret-key recovery attacks against the Apache HTTPS server. The latter is particularly damaging — an attacker can remotely extract a server's DSA secret key. With this key, the attacker can impersonate the server. We exhibit exploits when the victim is run within either VMWare [12] or VirtualBox [10], two popular VM managers.

The attacks work because the VM resets lead to cryptographic operations (here, key exchange and signing) using the same randomness more than once. These cryptographic operations, in turn, fail to provide any security given repeat randomness. One conceptually simple solution, then, is to ensure that applications sample sufficiently fresh randomness immediately before use. Unfortunately, there are lurking complexities to overcome. Besides the difficulty of ensuring every RNG-using application is updated, there is the more subtle problem of where to find good randomness after VM resets. For example, the state of traditional RNGs (e.g., Linux's /dev/random) is also reset with the rest of the guest. We provide more discussion of systems solutions in the body, but leave the bulk of this task to future work.

Finally, we suspect that the VM reset vulnerabilities we show are indicative of further issues. An important open question is whether other practical insecurities arise due to VM resets.

**Hedging deployed cryptography.** The attacks above are symptomatic of the widespread fragility of cryptographic operations to repeated or predictable randomness. Because many cryptographic operations fundamentally rely on good randomness to achieve the desired security level, repairing RNGs is the only full solution. However, the complexity of RNG design, the frequency with which RNG failures occur, and the significant damage that results all suggest that cryptography should be designed so that bad randomness has as little ill effect as possible.

One potential approach is to implement some form of hedged cryptography, considered in various ways in [19, 37, 46, 48]. The general idea is that routine cryptographic operations should be designed so that, given good randomness, they provably meet traditional security goals and, given bad randomness, they nevertheless provably achieve some meaningful security level. While not everywhere eliminating the need for good randomness, hedging against bad randomness can significantly mitigate the threat of RNG failure. Unfortunately, existing approaches only treat specific primitives such as public-key encryption [19, 52] or symmetric encryption [37, 46, 48] and only treat specific, disparate kinds of RNG failures. In this work we seek an approach that is fast, conserves existing security when randomness is good, works with arbitrary deployed algorithms, and can boost security in the face of arbitrary types of RNG failures.

We give a framework for hedging cryptographic operations that achieves these goals. Our starting point is techniques from [19, 52], which we show can be straightforwardly combined to be applicable to arbitrary cryptographic primitives. Briefly, an operation is replaced by a hedged version with equivalent functionality. The hedged version preprocesses RNG-derived randomness together with other inputs (messages, keys, etc.) with HMAC to provide (pseudo)randomness for the cryptographic operation. The modifications are simple. Even so, by changing the cryptography we can prove that important schemes are more secure in the face of various kinds of randomness failure.

We apply our framework to the latest OpenSSL code base, which doubles as a cryptographic library used by many applications and as a widely used implementation of the TLS protocol. Benchmarking the hedged version of the library indicates that overhead is very low. Because hedging does not impact functionality, our library interoperates transparently with existing TLS implementations. We are currently in the process of preparing our implementation for public release, which will allow immediate deployment with the corresponding security benefits.

## 2. Random Number Generation and Threat Models

There are many methods for generating cryptographically strong random numbers. We do not go into significant detail regarding particular implementations. See [36, 43] for details regarding some platform-specific random number generators. Instead we give an abstract model of random number generation which already suffices for discussing threats and attacks. We will fill in further details when necessary.

The task of a cryptographic random number generator (RNG) is to provide uniform, private bits to applications. We find it convenient to view an RNG as a *stack*. The first layer is the *entropy layer* where entropy is generated. Here sources of (hopefully) unpredictable events occur in a manner that can be sampled. Examples include temperature variations, clock drift, interrupt timings, mouse movements or keyboard clicks, and network packet arrivals. We view the entropy layer not as an actual software or hardware system, but rather as an encapsulation of the physical processes from which entropy is harvested. The second layer is the *sampling layer* which samples from the entropy layer to measure events and generate digital descriptions of them. This layer also attempts to extract uniform random bits from the unpredictable data and maintains a pool of such bits. The uniform bits are then provided to applications requiring randomness in the *consuming layer*.

We note that, crucially, the consuming layer may itself consist of further sequences of RNG-related systems. For example, in typical operating systems the kernel implements the sampling layer, a cryptography library consumes kernel-supplied randomness and also provides it to further applications. Note that every layer above the first potentially stores randomness internally in state. Looking ahead, this aspect of RNG stacks will be important in the context of reset attacks, which abuse reused state. Security requires that each layer ensures distinct requests (even ones made in parallel) are answered with fresh randomness.

**Threat models.** There exist numerous ways in which an RNG stack might fail or be tampered with by a dedicated attacker. For the purposes of threat modeling, however, we can loosely categorize randomness failures by the resultant quality of randomness as seen from the point of view of an application.

(1) *Fresh randomness*: An application is always provided new, private, uniform bits.
(2) *Reused randomness*: An application is provided private, uniform bits, but these bits might have been provided to the application before.
(3) *Exposed randomness*: An application is provided uniform bits but attackers later learn these bits.
(4) *Predictable randomness*: An application is provided random bits that are predictable by an adversary.
(5) *Chosen randomness*: An application is provided adversarially-chosen random bits.

For simplicity we will sometimes refer to fresh randomness as *good* and any of the four other kinds of randomness as *bad*. Our threat model is potentially malicious failure (at any level) of the RNG stack when performing routine randomized cryptographic operations, for example encrypting a message, signing a message, or performing key exchange. Randomness vulnerabilities lead to applications using one of the four kinds of bad randomness.

**Previous failures.** We classify previous randomness vulnerabilities into these threat models. We are unaware of any reports on vulnerabilities leading to reused randomness that didn't also lead to predictable randomness. Becherer et al. recently describe a possible vulnerability on Amazon EC2 [17] that leads to exposed randomness. Due to the architecture of EC2, an attacker might learn the initial state of a victim virtual machine's RNG. They conjecture (but provide only anecdotal evidence) that one might use this information to recreate cryptographic keys generated by the victim.

Examples of vulnerabilities leading to predictable randomness abound. Wagner and Goldberg exhibit session compromise attacks against SSLv2 because clients used predictable randomness for session keys [32]. Gutterman et al. point out that some systems may not have access to a sufficiently rich entropy layer, for example network routers without disks [36]. Gutterman and Malkhi discuss predictable randomness vulnerabilities in Java session identifiers [35]. Dorrendor et al. [28] point out that the Window's kernel RNG stored randomness in a way that is accessible to unprivileged programs. Woolely et al. uncovered a bug in the FreeBSD RNG that led to no entropy for several minutes after a reboot [51]. Perhaps the most spectacular example thus far is the Debian OpenSSL vulnerability, originally discovered by Bello [13]. Here a bug was introduced in the OpenSSL crypto library that tragically rendered impotent the RNG. Mueller [42] shows how to abuse the ensuing predictable randomness to recover long-lived TLS secret keys. Abeni et al. [16] and Yilek et al. [53] show how to abuse the predictable randomness to perform session compromise attacks.

## 3. VM Reset Vulnerabilities Affecting TLS

In this section we explore virtual machine (VM) reset vulnerabilities. These arise when applications' security-critical state is captured by a VM snapshot and starting the VM repeatedly from the snapshot leads to security problems. The VM reset vulnerabilities we consider are due to cryptographic randomness being cached by applications and caught in a snapshot. Running multiple times from the snapshot results in cryptographic operations consuming repeated randomness, and in turn, failing to provide security.

**Virtualization and snapshots.** Modern virtual machine monitors (VMMs) allow physical multiplexing of hardware resources between numerous guest operating systems, each run within a virtual machine (VM). Virtualization technologies have become almost ubiquitous. Consumers use VMs for home PC security, for example to contain malware infections. Businesses use virtualization to more efficiently manage computing infrastructure. The (so-called) cloud computing paradigm is powered by virtualization.

An important feature of almost all modern VMMs is the ability to take VM snapshots. A *snapshot* is a copy of the entire state of a VM, including active memory. This allows restarting the VM exactly in the state at which the snapshot was taken. VMWare [12], VirtualBox [10], and Virtual PC [11] all include snapshot mechanisms and advertise them as a core feature. Some VMMs offer a more limited *volume snapshot*, which is a copy of the persistent state (volumes) of a VM (but not active memory). Restarting from a volume snapshot requires booting the guest OS from the persistent state. Modern third-party cloud computing services such as Amazon EC2 [3], Microsoft Azure [7], Mosso Rackspace [1] all rely on volume snapshots to allow users of the service to save convenient server configurations for later use.

The attacks we describe below work against TLS [9] clients and servers when run multiple times from full snapshots. We did not investigate vulnerabilities due to volume snapshots; the particular vulnerabilities we found would not seem to carry over.

### 3.1. TLS Client Vulnerabilities

Recall that TLS is used to secure HTTP connections over the Internet. Thus, TLS protects the security of online banking, shopping, and other sensitive traffic. Every popular web browser therefore includes a TLS client, which is used to negotiate a shared secret, called a session key, between it and the remote HTTP server. The most prevalent [53] mode for establishing a session key is RSA key transport. Here the client chooses a secret value, called the premaster secret (PMS), encrypts it under the server's public RSA key, and then sends the resulting ciphertext to the server. The symmetric session keys used to secure the rest of the session are then derived from the PMS and two other values that are sent in the clear.

In abstract, a VM reset vulnerability could arise if the PMS, or the randomness used to create it, is generated before a snapshot and consumed upon resumption after the snapshot. This vulnerability would lead to an immediate compromise of sessions if the same PMS is sent to multiple different servers.

Before assessing whether this can occur in practice, we first ask: Why might a user run their browser in a virtual machine? Security experts recommend users do their web browsing within a VM to *increase* security. The idea is that if the browser has a vulnerability and a malicious site exploits it, the damage is contained to the VM. A user can revert to a previous snapshot taken before the browser and VM were compromised to undo the effects of any malware.

We performed experiments on a variety of browsers on both Linux and Windows to determine if there is a real problem. There is. Our results are summarized in Figure 1. We explain the results in detail below.

**Experimental setup.** We used two Apache web servers (call them server1 and server2) running on two separate physical machines. The servers used an instrumented version of OpenSSL that, upon receipt of the client's key exchange message in a TLS session using RSA key transport, would decrypt the premaster secret and write it to a file. Each server was given an RSA certificate signed by our own certificate authority (CA). We ran the various browsers (listed in Figure 1) within the indicated operating systems as guests inside a VM running in either VMWare 1.0.10 or VirtualBox 3.0.12. The physical host ran Ubuntu 8.04 Desktop. The client browsers, excepting Safari in Windows, were configured to accept our CA. This ensured that, upon visiting one of our servers, a browser in the guest OS would not complain about a certificate signed by an untrusted CA. (For Safari, we ended up just clicking "continue" when presented with a warning about an untrusted certificate.)

**Experiments.** We start with the following test sequence.

(1) Reboot the OS.

(2) Load the browser.

(3) Take a snapshot of guest in this state.

(4) Reset the VM.

(5) Navigate browser to server1.

(6) Reset the VM.

(7) Navigate browser to server2.

For each VM manager OS combination, steps (1-3) were performed once followed by 3 iterations of steps (4-7) for each browser. For Chrome on Linux, we also ran a separate test sequence where step (2) was changed to
(2a) Load the browser, navigate to an HTTPS url, and then navigate to a blank page.

The results were consistent between the two VM managers, meaning the VMM used had no impact on client behavior. For Firefox on Windows or Linux, the same PMS was sent to both servers in all 3 trials. If the user caused 100 mouse events (e.g., moved the mouse 100 pixels) between steps (4) and (5) or (6) and (7) then distinct PMS values were sent to the servers. This is because Firefox folds new entropy into the RNG every 100 mouse events. For Chrome on Linux, when step (2a) was used then the same PMS was sent to both servers in all 3 trials. When step (2) was used, distinct PMS values were sent to the two servers.

On Windows, all browsers except Firefox always sent distinct PMS values to both servers. We note however that on Windows, the same PMS value was sent to the same server in many of the trials. While this does not admit an obvious attack, it violates the TLS specification. For example, on IE 6.0 and VMWare, 2 out of the 3 PMS values sent to server1 were the same and 2 out of the 3 PMS values sent to server2 were the same. We note that all the browser/VMM combinations showed this problem; for Chrome in Windows, it did not even matter whether or not step (2a) or (2) was used.

## 3.2. TLS Server Vulnerabilities

We turn our attention to TLS servers. We target TLS servers using authenticated Diffie-Hellman key exchange when the digital signature scheme used is DSA. Ignoring numerous (but not really relevant) details, the protocol works as follows. The client sends a nonce of its choosing to the server. The server chooses a random $y$, computes $g^y$, and uses DSA to sign $g^y$ together with session state including the client nonce. Now, if the server uses the same randomness to sign in two different sessions, an attacker who knows the messages signed and sees the resulting signatures can extract the server's DSA signing key. This attack is well-known; see Appendix A for details.

A VM reset vulnerability could therefore arise if running a server multiple times from a snapshot led to the same randomness being used by DSA. This situation would occur, for example, if an HTTP server seeds its RNG at startup and uses the RNG (without new seeding) to generate randomness when handling a new connection. If a snapshot of the server is taken after startup, then the RNG's state would be captured. Reuse of the snapshot could lead to the same randomness being consumed by DSA.

If real, such attacks would be particularly threatening

| TLS Client | Guest OS | Same PMS to diff. sites? | Same PMS to same site? | Comments |
|---|---|---|---|---|
| Firefox 3.5 | Windows | Always | Always | Mouse moves $<$ 100 pixels |
| Chrome 3.0 | Windows | Never | Sometimes | - |
| IE 6.0 | Windows | Never | Sometimes | - |
| Safari 4.0 | Windows | Never | Sometimes | - |
| Firefox 3.0 | Ubuntu | Always | Always | Mouse moves $<$ 100 pixels |
| Chrome 4.0 | Ubuntu | Always | Always | Visit one HTTPS site before snapshot |

**Figure 1. Summary of our TLS client attacks. We performed all of the experiments on both VMWare Server version 1.0.10 and VirtualBox version 3.0.12 and observed the same behavior. Ubuntu refers to version 8.04 (Hardy) Desktop, Windows refers to XP Professional with Service Pack 2.**

due to the ever-increasing use of virtualization to manage servers. For example, cloud computing services such as EC2 (popular for hosting HTTP servers) utilize volume snapshots to store customer server configurations for rapid deployment. Progressing to full snapshots that include active memory could make provisioning of VMs even faster. In other settings, snapshots are often suggested as a simple mechanism for fast server crash recovery.

We therefore investigate whether servers are vulnerable. We focus on Apache using mod_ssl, the most widely-used HTTPS web server software. Below we describe how Apache generates randomness and then experimentally validate that, in fact, VM reset vulnerabilities can be exploited by an attacker to extract a server's DSA secret key.

**The Apache + mod_ssl RNG**. Apache uses the OpenSSL cryptographic library for its RNG and TLS implementation. On Apache startup, the main process seeds the OpenSSL RNG from various entropy sources. Which sources to use can be specified in the Apache configuration file. By default, Apache only uses time (seconds since the Epoch), process ID, and a portion of the runtime stack to seed the RNG; this is referred to as "built-in" in the configuration file. The OpenSSL RNG will also seed itself from a variety of sources depending on how it is configured; these sources usually include `/dev/random` or `/dev/urandom` on Unix-based systems. Apache then forks off into child processes that will actually handle the incoming HTTP and HTTPS requests. At this point each child process has its own copy of the now-initialized RNG state.

When an HTTPS request comes in, Apache assigns a child process to handle it. Before handling the process, the child adds more entropy to its RNG. Here what is added again depends on the Apache configuration, which by default is set to "built-in", meaning only the time, process ID, and some of the runtime stack are added to the RNG before the request is handled. The nature of our attack is such that only these new additions affect whether randomness will be repeated between two reverts. In other words, if we can connect to a server twice run from the same snapshot and the time, pid, and runtime stack are the same in both cases, the server will use the same randomness for both sessions.

**Experiments without Clock Synchronization**. Because time is added to the RNG, it is clear that the server's clock plays a crucial role in whether an attack could be successful. Specifically, if the server resynchronizes its clock after starting execution from a snapshot but before an adversary can connect to it, then the RNG will never supply the same value twice. Thus, as a first step, we experiment with VMs that do not perform guest clock synchronization.

We set up a default installation of an Ubuntu 8.04 Hardy desktop guest OS inside VMWare Server. Here the guest does not synchronize its clock. (If one additionally installs VMWare Tools, which are guest utilities supplied by VMWare, then the guest defaults to synchronizing its clock.) On the guest, we also set up OpenSSL 0.9.8k and the latest version of Apache web server with mpm-prefork (i.e., unthreaded) and using the default configuration. The only changes we made to the configuration file were to point the server at a DSA key and certificate.

We then performed the following steps. We rebooted the guest VM, started the web server, paused the VM, and took a snapshot. Then we repeatedly attempted to start the VM from the snapshot and have a client connect to the server at exactly the same time (as read on the guest's clock). We did this by setting a client machine physically next to the server's host machine, and having one person watch for the clock on the guest VM to tick to the next minute after the snapshot was taken. Immediately, that person executed the `openssl s_client` command on the client machine to initiate a DSA-authenticated key exchange with the server.

Somewhat surprisingly, this worked. The randomness used by the server was repeated. We observed repeated session IDs (generated using the RNG), and repetitions of the randomness used to sign with DSA. We tried many more times with similar success. Apparently the same child PID and stack contents were used each time to generate the randomness, and it was easy for the single operator to get the timing right.

| | VMM | Time Sync | Restart? | # Session ID pairs | # DSA extract pairs |
|---|---|---|---|---|---|
| 1 | VMWare | no | no | 6/10 | 6/10 |
| 2 | VMWare | no | yes | 3/10 | 1/10 |
| 3 | VBox | yes | no | 10/10 | 10/10 |
| 4 | VBox | yes | yes | 10/10 | 10/10 |
| 5 | VMWare | yes | no | 0/10 | 0/10 |
| 6 | VMWare | yes | yes | 4/10 | 3/10 |

**Figure 2. Summary of Server attacks. Each row corresponds to five trials. The second-to-last column reflects how many pairs of trials (out of a possible ten) contained sessions using the same session IDs. The last column reflects how many pairs of trials contained sessions using the same randomness for DSA signing.**

Our next step was to try to automate the attack to see if it would work remotely. We worked under the assumption that after a reset, the VM would usually take about the same amount of time to start servicing HTTPS requests. Thus, our attack strategy was to repeatedly attempt to connect to the server (starting around the time we knew the reset occurred) with the hope of achieving a successful connection at the very moment the server started answering requests. If after two different resets the server takes the same number of seconds to start answering, then the attack should succeed. To test this, we created a script that attempted to connect to the server once every 0.1 seconds and recorded any session data. We then performed five trials consisting of the following steps using VMWare 1.0.10 as our VMM, Ubuntu 8.04 Desktop as our host and guest OS, and Apache 2.2.14 mpm-prefork with OpenSSL 0.9.8k:

(1) Start the VM from the snapshot.

(2) Start the attack script.

(3) Stop the script after a few seconds of successful connections.

We did this both when restarting the host physical machine before each trial and restarting before only the first trial. We call trials executed immediately after a reboot "cold" trials, while we call trials executed after other trials (without a reboot in between) "hot" trials.

The results are summarized in the first two rows of Figure 2; the first row represents hot trials (except the first of the five, which is cold), while the second row represents cold trials. Now, we are interested in whether or not randomness is reused across resets. Thus, for each set of five trials, we count how many pairs of trials contain sessions using the same randomness. We particularly record if a pair had the same session ID (chosen using the RNG) or used the same randomness for DSA signing.

As can be seen randomness repetition occurred in both sets of 5 trials. We noticed that during cold trials, resuming from a snapshot is significantly slower than in hot trials. This affected timing. For example, row 1 in the table did not have 4 randomness collisions because the first trial there was cold while the others were hot, leading to timing differences. Also, we believe the timing variability of cold trials accounts for the lower success rate seen in row 2.

**Experiments with Clock Synchronization.** We would like our automated attack to work even if the guest is synchronizing its clock. Though synchronization would seem to bar any chance of attack, it turns out that there is often a window of opportunity for an attacker. Once a VM is loaded from a snapshot, everything needs to be reinitialized; this includes re-loading values into memory, re-enabling networking, synchronizing the clock, and more. We observed that VMs running on both VirtualBox and VMWare VMMs would, after starting from a snapshot, often enable networking and service HTTPS requests *before* synchronizing the clock.

We experimented with VMWare 1.0.10 and VirtualBox 3.1.0 using the same configuration, automated script, and steps as above. For VMWare, we tested our attack with VMWare tools installed to ensure clock synchronization. As before and for each VMM, we performed five trials rebooting the host machine between trials and five trials rebooting only before the first trial. The results are shown in Figure 2, rows 3 through 6. Again, sessions used the same randomness across multiple TLS sessions. VirtualBox had very consistent resumption timing. The higher variability of VMWare leads to lower success rates. In particular, as seen in row 5 of the table, when doing hot trials VMWare actually synchronized the guest clock before Apache started servicing the attack connections.

**Discussion.** One might wonder why session IDs repeat more often than DSA randomness; while we do not have a definitive answer for this, we suspect it is because OpenSSL mixes in an additional time value immediately before DSA signing, and for some sessions this time ends up being time X while for others it ends up as time X+1.

In these experiments, we also observed repeat Server-Randoms and Diffie-Hellman key exchange values. These values should all be unique in a proper TLS handshake. We do not know how to exploit this repetition, but nevertheless believe care should be taken to avoid it.

## 4. On Fixing the Vulnerabilities

In this section we provide a brief discussion about fixing the VM reset vulnerabilities uncovered. In the TLS clients and servers we described above, we saw that good randomness was sampled at some point (such as starting the program or launching a child process) and buffered until it was needed at some much later time. This allowed a large window in which snapshots would capture to-be-used randomness. In the browser client vulnerabilities, the randomness was used directly in a cryptographic operation after the snapshot. On the other hand, with Apache, new entropy was added to the RNG right before its output was used in the cryptographic operation — unfortunately the sources had little to no entropy conditioned on their being used already by a previous snapshot.

In abstract, fixing these vulnerabilities requires ensuring that RNGs get access to sufficient entropy after a snapshot and ensuring that applications take randomness from an RNG at the time of the cryptographic operation. For example, one approach would be to mandate using a guest OS source such as `/dev/random` or `/dev/urandom` to generate randomness right before a cryptographic operation is performed.

Unfortunately, the state of these sources is also reset by snapshots, and so it is unclear whether sufficient entropy is generated between a snapshot resumption and randomness consumption by the cryptographic operation. In general, a better option would likely be linking guest RNG services with hardware-based RNGs or other external sources.

This is a large topic, and we leave finding the best solutions to future work. Instead, we turn our attention to strategies for mitigating the threat of all types of RNG failures to better protect against future problems.

## 5. A General Framework for Hedging against Randomness Failures

As mentioned in the introduction, there is a long history of RNG failures [13, 17, 28, 32, 35, 36, 51] stemming from a variety of issues. The VM reset attacks discussed in Section 3 show yet another manner by which RNG's fail. Additionally, the reset attacks, among other attacks [16, 53], target routine cryptographic operations that are fragile in the face of bad randomness. For example, most constructions for key exchange, randomized signing, and encryption admit damaging attacks given bad randomness, even when good long-lived key material is used.

We propose a general framework for hedging against RNG failures. Our method modifies routine cryptographic operations so that they can defend themselves against various forms of bad randomness. By focusing on the cryptography, our framework is application- and VMM-agnostic. It protects against many different kinds of bad randomness. It is simple to implement and deploy.

Hedging is not a replacement for good RNGs. In particular, for many cryptographic tasks one needs randomness to achieve the most desirable security levels (e.g., public-key encryption). In these cases, hedging provides graceful degradation of achieved (provable) security.

**Hedged cryptography.** Our framework is the following. A *hedging function* Hedge is a deterministic algorithm with inputs being an arbitrary-sized bit string $R$ and an arbitrary number $p$ of associated data bit strings $(d_1, \ldots, d_p) \in (\{0,1\}^*)^p$. (In implementations, we will actually have to cap the maximum length of inputs treated by Hedge to, e.g., $2^{64}$. We omit this detail throughout for simplicity.) We write $\mathbf{d}$ to mean the vector $(d_1, \ldots, d_p)$. The algorithm outputs a bit string of size $|R|$ bits, where $|R|$ is the length of $R$ in bits. We write $\mathsf{Hedge}(R, \mathbf{d}) = \mathsf{Hedge}(R, d_1, \ldots, d_p)$ to denote running the algorithm. A hedging function handles variable-length keys, inputs, and outputs.

Let Op be a randomized cryptographic operation taking inputs $i_1, i_2, \ldots, i_k$ and using an RNG-supplied bit string $R$. Denote execution of it by $\mathsf{Op}(i_1, i_2, \ldots, i_k\ ;\ R)$. We hedge by replacing calls

$$\mathsf{Op}(i_1, i_2, \ldots, i_k\ ;\ R)$$

with

$$\mathsf{Op}(i_1, i_2, \ldots, i_k\ ;\ \mathsf{Hedge}(R, \langle \mathbf{d} \rangle))$$

where $\mathbf{d} = (\text{OpID}, i_1, i_2, \ldots, i_k)$. That is, we apply Hedge to the RNG-supplied randomness and the inputs to the operation become the associated data. We also include OpID, which denotes some unique identifier for the operation, to provide domain separation between uses of Hedge with distinct operations. Note that we will omit explicit mention of OpID later for brevity, but it is crucial to use within implementations. The output of Hedge is used as the "randomness" for the operation. Note that functionality is not changed since randomized cryptographic operations must work for any $R$.

The idea of hedging originates with work by Bellare et al. where they treat the specific case of public-key encryption [19]. One of their constructions is a special case of ours, where Hedge is replaced with a cryptographic hash function and Op is specifically public-key encryption. Yilek [52] treats the special case of randomness reuse attacks against public-key encryption. His construction is a special case of ours where Hedge is just a PRF and Op is public-key encryption. These two approaches achieve dif-

| Primitive | Repeat | Exposed | Predictable | Chosen |
|---|---|---|---|---|
| Public-key encryption | $\star$ | $\diamond$ | $\diamond$ | $\diamond$ |
| Symmetric encryption | $\star$ | $\star$ | $\star$ | $\star$ |
| Digital signatures | UF | UF | UF | UF |

**Figure 3. Summary of hedged primitive's provable security in the face of Repeat, Exposed, Predictable, or adversarially Chosen randomness. Symbol $\star$ means no partial information about plaintexts leaked except plaintext equality. Symbol $\diamond$ means no partial information about plaintexts leaked assuming plaintexts have high min-entropy (unpredictable to attacker). Symbol UF means no attacker can forge an honest party's signature.**

ferent (and orthogonal) security guarantees for public-key encryption, as discussed further in the next section. Suffice to say, we simultaneously want both guarantees. Moreover, we want the same hedging approach to work for other cryptographic primitives. We can achieve this by instantiating Hedge with an object that is good both as a hash function and as a PRF.

We suggest the widely-available HMAC algorithm [20], built from a sufficiently strong underlying hash function, such as SHA-256, SHA-512 or the upcoming SHA-3. HMAC takes as input an arbitrarily-long key $K$ and message $M$ and outputs a string of $n$ bits (e.g., $n = 256$ for SHA-256). We can use HMAC to instantiate $\mathsf{Hedge}(R, d_1, \ldots, d_k)$ for some number $k$ of associated data strings as follows. Let $p$ be the smallest integer such that $np \geq |R|$. Compute $R'_i = \mathsf{HMAC}(R, \langle d_1, \ldots, d_k \rangle \mid\mid i)$ for $1 \leq i \leq p$ and then output the first $|R|$ bits of

$$R'_1 \mid\mid \cdots \mid\mid R'_p \ .$$

Here $\langle d_1, \ldots, d_k \rangle$ is some unambiguous encoding of the associated data and $\mid\mid$ represents concatenation of bit strings. In words we apply HMAC several times, using the RNG-supplied randomness $R$ as the key and the associated data combined with a counter as the message. The counter allows us to produce $p$ times the output size of HMAC (e.g. $512p$ if using SHA512 within HMAC). We then run Op using the appropriate number of bits of HMAC output.

**Discussion.** As mentioned above, functionality is not hindered. That means that hedging a cryptographic operation has no impact on other, related operations (e.g., decryption need not be changed when hedging an encryption routine). This crucially means that hedging is legacy-compatible: any party can use it and no other parties need know.

We point out that the framework can be just as easily applied to long-lived key generation (in addition to routine cryptographic operations). However here one will not, generally, achieve significant security improvement: there are usually no other inputs to such routines beyond the randomness used. Nevertheless in cases where there are (e.g. the identity of a party generating a public key, secret key pair) it might prove beneficial for some kinds of randomness failures. Note that long-lived key generation is a rare operation and ensuring it access to fresh randomness might therefore be easier than more routine operations.

Lastly, the focus of this work is cryptographic consumers of randomness. However there are other security-critical uses of randomness and hedging might prove useful in these as well.

## 6. Security of Hedging

In this section we discuss the security that hedging provides. First, we present some general security properties of hedging, namely that the hedge function does not degrade the quality of good randomness given to an underlying operation. This is important because it provides some argument that hedging won't hurt most security properties. We will then discuss hedging of four important primitives: public-key encryption (PKE), symmetric encryption (SE), digital signatures (DS), and key exchange. For all these primitives, many in-use schemes fail completely to provide security in each of the randomness failure models.

Figure 3 provides a summary of the security provably achieved by hedging public-key encryption, symmetric encryption and digital signing. (This is assuming the underlying primitive is secure when randomness is good.) Briefly, hedged PKE will not leak anything but plaintext equality when randomness is repeated. If randomness is adversarially chosen, predictable, or exposed, then hedged PKE will not leak any partial information assuming unpredictable, public-key independent messages. Hedged SE leaks nothing but plaintext equality even against adversarially-chosen randomness. Hedging essentially removes the need for randomness in digital signing — the traditional notion of unforgeability is achieved. All these results, including the more complex situation for hedged key exchange, are discussed in further detail in the remainder of this section.

**Formalisms and notation.** We formalize our security notions using code-based games [25]. In this framework, one models security as a game played with an adversary. A game (see Figure 5 for an example) has an **Initialize** procedure, procedures to respond to adversary oracle queries, and a **Finalize** procedure. First, **Initialize** executes and its outputs are given as inputs to an adversary $A$. Next $A$ executes and can adaptively make queries to procedures (other than **Initialize** and **Finalize**), receiving the computed responses. When $A$ terminates with some output, this becomes the input to **Finalize**. We denote running a game $G$ with adversary $A$ as $G^A$ and let $G^A \Rightarrow w$ be the event (in the probability space induced by $G^A$) that the output of game $G$, when run with adversary $A$, is $w$. An adversary's run time is the time to run $G^A$, meaning particularly that we charge the adversary for its queries. If working within the random oracle model (i.e. assuming Hedge behaves like an ideal hash function), then the game has one more procedure implementing the random oracle. This procedure, usually denoted **H**, returns for each (distinct) message queried a randomly chosen value.

## 6.1. General Security Properties of Hedging

Proving the security improvements achieved by hedging requires focusing on individual primitives, as we do in the following sections. First however, we discuss general security properties needed from Hedge and (informally) how they lead to security gains. To start, we point out that one desires that the output of Hedge is indistinguishable from true randomness whenever one of the following holds:

(1) fresh randomness $R$ is used;

(2) repeated $R$ is used, but all pairs of associated data used with $R$ are distinct; or

(3) adversarially chosen $R$ is used, but some portion of the associated data is unpredictable to and hidden from the adversary (e.g. a secret key or large plaintext).

Informally, property (1) holds under the very mild assumption that Hedge is a good pseudorandom function (keyed by $R$) for a very small number of queries. Property (2) should hold under the assumption that Hedge is a good PRF for many queries and property (3) should hold if Hedge is an ideal hash function (a random oracle). We therefore discuss how Hedge meets the preconditions just described (being a weak PRF, a PRF, and an ideal hash).

We first show that Hedge is a one-time secure PRF. Formally, a variable-key-length one-time PRF (ot-prf) adversary $A$ takes no input, can query a pair $r, \mathbf{d}$ where $r > 0$ is a number and $\mathbf{d}$ is a vector of bit strings to an oracle, and outputs a bit. Let $H$ be a hedging function (as per Section 5). Game OT-PRF$_H$ is defined in Figure 4. The advantage of ot-prf adversary $A$ against a keyed function $H$ is $\mathbf{Adv}^{\text{ot-prf}}_H(A) = \Pr\left[\text{OT-PRF}^A_H \Rightarrow \mathsf{true}\right] - \Pr\left[\text{OT-PRF}^A_H \Rightarrow \mathsf{false}\right]$.

Let $H$ be a variable-key-length (VKL) function with output length $n$. This means $H$ is a deterministic algorithm

```
Game OT-PRF_H

Initialize:
$b \leftarrow_{\$} \{0,1\}$

Func($r, \mathbf{d}$):
$y_0 \leftarrow_{\$} \{0,1\}^r$
$y_1 \leftarrow H(y_0, \mathbf{d})$
Return $y_b$

Finalize($b'$):
Return $(b = b')$
```

```
Game PRF_{H,k}

Initialize:
$K \leftarrow_{\$} \{0,1\}^k$
$b \leftarrow_{\$} \{0,1\}$

Func($\mathbf{d}$):
$y_1 \leftarrow H(K, \mathbf{d})$
$y_0 \leftarrow_{\$} \{0,1\}^{|y_1|}$
Return $y_b$

Finalize($b'$):
Return $(b = b')$
```

**Figure 4. One-time PRF and PRF security games for variable-key-length function $H$.**

that takes a key $K$ of some arbitrary length $k > 0$ and an arbitrary-sized-input and outputs a string of length $n$. HMAC is an example of a VKL function. Game PRF defines normal PRF security, see Figure 4. The prf advantage of adversary $B$ against $H$ is $\mathbf{Adv}^{\text{prf}}_H(B) = \Pr\left[\text{PRF}^A_{H,k} \Rightarrow \mathsf{true}\right] - \Pr\left[\text{PRF}^A_{H,k} \Rightarrow \mathsf{false}\right]$.

It is well-known that HMAC is a secure PRF [18, 20] assuming the underlying hash function's compression function has suitable PRF-like security properties. (Technically, one needs resistance to a mild form of related-key attack [22] in addition to the standard PRF security.) Namely, one can use the results of [18, 20] to derive bounds for $\mathbf{Adv}^{\text{prf}}_{\mathsf{HMAC},k}(B)$ for any $k$. A simple hybrid argument establishes the following theorem, which is stated for an arbitrary VKL function $F$.

**Theorem 6.1** Let $F$ be a VKL function with output length $n$. Let Hedge be the hedging function built from $F$, as in Section 5. Let $A$ be a ot-prf adversary running in time $t$, making at most $q$ queries specifying lengths $r_1, r_2, \ldots, r_q$. Then there exists $q$ prf adversaries $B_i$ such that

$$\mathbf{Adv}^{\text{ot-prf}}_{\mathsf{Hedge}}(A) \le \sum_{i=1}^{q} \mathbf{Adv}^{\text{prf}}_{F,r_i}(B_i)$$

where $B_i$ runs in time that of $A$ and makes $\lfloor (r_i + n - 1)/n \rfloor$ queries. $\square$

We can see that the security required from the VKL function, e.g., HMAC is mild, because $m = \lfloor (r_i + n - 1)/n \rfloor$ is generally very small. For example when one uses SHA-256 and $r_i = 1024$, we have that HMAC must resist merely $m = 4$ queries for the same key. Finally, we point out that this reasoning doesn't hold if $r_i$ is too small. However, for the primitives that we suggest hedging, $r_i \ge 128$.

Finally, security relies on choosing $n$ sufficiently large (e.g., if using SHA-256, SHA-512 in HMAC).

When Hedge is used repeatedly with the same randomness, security as a one-time PRF no longer suffices. Here we require it to be a secure PRF, but this provably holds under the assumption that HMAC is a secure PRF for an appropriate number of queries.

Finally, and as mentioned above, we also require that Hedge enjoys security properties when randomness is known (or even chosen) by an adversary, meaning in particular that the randomness input can no longer serve as a secret key (allowing us to use the PRF security of HMAC). In these contexts we'll appeal to modeling Hedge as ideal, or in cryptographic parlance, a random oracle. This means that it maps every input to an output randomly chosen from the space of all outputs for that length. Note that this requirement excludes many other potential instantiations (instead of HMAC), such as most block-cipher-based [15] or universal-hashing-based [50] MACs, which are not suitable for modeling as random oracles.

## 6.2. Public-key Encryption

Public-key encryption (PKE) schemes are used to provide message privacy, and they have the benefit of asymmetry: only the recipient's public key value is needed to encrypt a message. Existing PKE schemes fail spectacularly in the face of randomness failures. For example, all hybrid encryption schemes (those used most frequently in practice) allow plaintext recovery given predictable randomness and some (e.g., those based on CTR-mode symmetric encryption) allow plaintext recovery given repeat randomness [19]. Predictable randomness leads to plaintext recovery for the Goldwasser-Micali scheme [33] and the El Gamal scheme [30]. Brown gave a plaintext-recovery attack against OAEP abusing predictable randomness [26]. Ouafi and Vaudenay gave a plaintext-recovery attack against Rabin-SAEP [45].

Hedged public-key encryption was recently introduced by Bellare et al. [19]. They propose to hedge arbitrary PKE schemes with their Randomized Encrypt with Hash (REwH1) approach, which is the hedging approach described in the last section except explicitly using a normal hash function $H$ (instead of Hedge). In [19] it is shown that if $H$ is modeled as ideal (a random oracle) then the hedged PKE scheme simultaneously enjoys two security properties. The first is the traditional notion of semantic security [33]. The second is a new notion they introduce called indistinguishability under chosen distribution attack (IND-CDA). Intuitively this latter notion means the scheme behaves like a secure deterministic encryption scheme — no partial information about plaintexts is leaked as long as they are drawn from a sufficiently large "space". That is, one gets stronger guarantees of security even when randomness is adversarially chosen. In subsequent work Yilek [52] treats the case of reused (but not adversarially chosen) randomness. He suggests a construction which is what we described above except Hedge is replaced by an arbitrary keyed PRF. His security notion is orthogonal to that of [19], meaning meeting one does not imply meeting the other (and vice versa). Our hedging framework was inspired by these works.

**Hedged security.** Formally, a PKE scheme consists of a triple of algorithms $(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$. Key generation Kg uses randomness to generate a public key, secret key pair $(pk, sk)$. Encryption Enc takes inputs a public key $pk$, message $M$, and randomness $R$ and outputs a ciphertext. We denote this computation by $\mathsf{Enc}(pk, M\ ;\ R)$. Decryption Dec takes inputs a secret key $sk$ and a ciphertext $C$ and outputs either a message $M$ or a distinguished error symbol $\perp$. We denote computing this by $\mathsf{Dec}(sk, C)$. We hedge a scheme PKE by replacing $\mathsf{Enc}(pk, M\ ;\ R)$ with $\mathsf{Enc}(pk, M\ ;\ \mathsf{Hedge}(R, pk, M))$. As before, decryption need not be modified.

Since we specify that Hedge be both a secure keyed PRF and an ideal hash function, hedging a PKE scheme simultaneously achieves the security goals of [19] and of [52]. Indeed, the analyses given by Bellare et al. and Yilek apply directly, and so for the (lengthy) technical details we refer the reader to these papers [19, 52].

## 6.3. Symmetric Encryption

Symmetric encryption (SE) schemes are used to provide message privacy and authenticity when two parties share a secret key. Many schemes have randomized encryption algorithms. In this context the randomness used is often called the $IV$ (initialization vector). When the $IV$ is repeated, existing schemes leak partial information about plaintexts (e.g., CBC-based modes [44], OCB [47]) or even leak plaintexts completely (e.g., CTR-based modes [44] including GCM [40]).

Symmetric encryption robust to randomness failure was first proposed by Rogaway and Shrimpton [48] where they formalize misuse-resistant authenticated encryption. They provide new cryptographic schemes that meet this stronger goal. A viewpoint inherent in their work is that of nonce-based symmetric encryption due to Rogaway [46], where the IV is an explicit input to encryption. We inherit this viewpoint as well. Kamara and Katz [37] also suggest a form of SE that survives some kinds of randomness failures, but these goals and schemes handle more limited randomness threats. Applying our hedging framework to a traditional authenticated-encryption scheme results in the same security levels achieved in [48], while retaining backwards compatibility. Informally, the achieved security prevents

```
Initialize:                 Enc(M, R):                          Dec(C):                                  Game MRAE_SE
  $K^* \leftarrow_\$ Kg$      $C \leftarrow Enc^H(K^*, M ; R)$      If $b = 1$ then Ret $Dec(K^*, C)$
  $b \leftarrow_\$ \{0,1\}$   If $b = 1$ then Ret $C$               Ret $\perp$                     H(X):
                              $C' \leftarrow_\$ \{0,1\}^{|C|}$                                        If $H[X] = \perp$ then
                              Ret $C'$                            Finalize(b'):                       $H[X] \leftarrow_\$ \{0,1\}^r$
                                                                    Ret $(b = b')$                    Ret $H[X]$
```

**Figure 5. Security game for symmetric encryption (MRAE).**

leaking anything but plaintext equality no matter how bad the randomness.

**Hedged security.** Formally, an SE scheme is a triple of algorithms $(\mathsf{Kg}, \mathsf{Enc}, \mathsf{Dec})$. Key generation $\mathsf{Kg}$ uses randomness to generate a key. Encryption $\mathsf{Enc}$ takes as input a key $K$, message $M$, and uses randomness $R$ to generate a ciphertext (or $\perp$). We say that SE has randomness length $r$ when $\mathsf{Enc}$ only accepts $R$ with $|R| = r$. (This check will always be implicit.) We denote executing encryption by $\mathsf{Enc}(K, M\ ;\ R)$. Decryption $\mathsf{Dec}$ takes as input a key $K$ and ciphertext $C$ and outputs either a message $M$ or distinguished error symbol $\perp$. We denote this process by $\mathsf{Dec}(K, C)$. We hedge a scheme SE by replacing $\mathsf{Enc}(K, M\ ;\ R)$ with

$$\mathsf{Enc}(K, M\ ;\ \mathsf{Hedge}(R, K, M))\ .$$

Decryption works as before.

We utilize the notions of security due to Rogaway and Shrimpton [48] for misuse-resistant authenticated encryption. Here we measure an adversary's ability to distinguish between two pairs of oracles, one pair being an encryption oracle and a decryption oracle and the other pair being an oracle that returns an appropriately long string of random bits and an oracle that always returns $\perp$. Let SE be a symmetric encryption scheme with randomness length $r$. Game $\mathrm{MRAE}_{\mathsf{SE}}$ in Figure 5 defines security, in the random oracle model [23], in the sense of misuse-resistant authenticated encryption (MRAE). The notation $\mathsf{Enc}^{\mathbf{H}}$ means that $\mathsf{Enc}$ might use access to the random oracle implemented by procedure **H**. An MRAE adversary $A$ takes no inputs, never repeats an encryption query or **H** query, and never queries **Dec** on a value returned by **Enc**. Its advantage against scheme SE is defined by

$$\mathbf{Adv}^{\mathrm{mrae}}_{\mathsf{SE}}(A) = 2\cdot\Pr\left[\,\mathrm{MRAE}^A_{\mathsf{SE}} \Rightarrow \mathsf{true}\,\right] - 1\ .$$

We define the traditional notion of symmetric encryption security via game $\mathrm{AE}_{\mathsf{SE}}$ (not shown) which is the same as $\mathrm{MRAE}_{\mathsf{SE}}$ except that there is no input $R$ to **Enc** and instead this value is picked uniformly at random upon each encryption query. (We omit random oracles here since we will not need them.) An AE adversary $A$ takes no inputs and never queries **Dec** on a value returned by **Enc**. Its advantage against scheme SE is defined by

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

A standard fact is that $\mathbf{Adv}^{\mathrm{mrae}}_{\mathsf{SE}}(A) = \Pr\left[\,\mathrm{MRAE1}^A_{\mathsf{SE}} \Rightarrow \mathsf{true}\,\right] - \Pr\left[\,\mathrm{MRAE0}^A_{\mathsf{SE}} \Rightarrow \mathsf{false}\,\right]$ where MRAE1 (resp. MRAE0) is the same as game MRAE except that the challenge bit $b$ is set to one (resp. zero). The same holds for $\mathbf{Adv}^{\mathrm{ae}}_{\mathsf{SE}}(A) = \Pr\left[\,\mathrm{AE1}^A_{\mathsf{SE}} \Rightarrow \mathsf{true}\,\right] - \Pr\left[\,\mathrm{AE0}^A_{\mathsf{SE}} \Rightarrow \mathsf{false}\,\right]$. Let SE be any scheme, let $\mathsf{MsgSp}$ be the set of all messages SE handles, and let $\mathsf{SE}^+$ be the hedged version of it. We implement Hedge via the appropriate unambiguous encoding of the triple $(R, K, M)$ into a value $X$ that is then queried to the random oracle **H**. (For simplicity, we ignore algorithm identifiers.) We have the following theorem.

**Theorem 6.2** Let $A$ be an MRAE adversary making at most $q_h$ hash queries, $q_e < |\mathsf{MsgSp}_{\mathsf{SE}}|$ encryption queries, and $q_d$ decryption queries. Then there exists an AE adversary $B$ such that $\mathbf{Adv}^{\mathrm{mrae}}_{\mathsf{SE}^+}(A) \leq 2\cdot\mathbf{Adv}^{\mathrm{ae}}_{\mathsf{SE}}(B)$. Moreover, $B$ runs in time at most $\mathbf{T}(A) + q_h\mathbf{T}(\mathsf{Enc})$, makes the same number of encryption queries, and makes at most $q_h + q_d$ decryption queries. $\square$

Note that our reduction is not entirely tight. We believe a tighter analysis can be given, at the cost of a slightly more involved reduction than the one we give.

**Proof:** We use a sequence of games to prove the theorem. Let game G0 work exactly like $\mathrm{MRAE1}_{\mathsf{SE}^+}$ except that we set a flag bad if $A$ makes a query to **H** encoding a triple $(R, K^*, M)$ for some $R, M$. Let G1 be the same as G0 except that **Enc** now implements $\mathsf{Enc}$ instead of the hedged function. That is, it uses true randomness and does not query the random oracle. We have by the fundamental lemma of game playing [25] that

$$\Pr\left[\,\mathrm{G0}^A \Rightarrow \mathsf{true}\,\right] - \Pr\left[\,\mathrm{G1}^A \Rightarrow \mathsf{true}\,\right] \leq \Pr\left[\,\mathrm{G1}^A \text{ sets } \mathsf{bad}\,\right]$$

where "$\mathrm{G1}^A$ sets $\mathsf{bad}$" is the event that $A$ forces $\mathsf{bad}$ to be set in game G1. Game G2 works like G1 except that **Enc** queries are responded to with randomness of equal length (i.e., using the **Enc** procedure of game $\mathrm{AE0}_{\mathsf{SE}}$) and all **Dec** queries are responded to with $\perp$. We have that G2 is equivalent to $\mathrm{MRAE0}_{\mathsf{SE+}}$, meaning that $\Pr[\mathrm{G2}^A \Rightarrow \mathsf{true}] = \Pr[\mathrm{MRAE0}_{\mathsf{SE+}} \Rightarrow \mathsf{true}]$. Moreover we can build an adversary $B'$ such that

$$\Pr\left[\mathrm{G1}^A \Rightarrow \mathsf{true}\right] - \Pr\left[\mathrm{G2}^A \Rightarrow \mathsf{true}\right] \leq \mathbf{Adv}^{\mathrm{ae}}_{\mathsf{SE}}(B') .$$

The adversary $B'$ just runs $\mathrm{G1}^A$ except implementing **Enc** and **Dec** using its oracles instead. All that remains is to bound the setting of $\mathsf{bad}$ in game G1. This event indicates that $A$ managed to query the secret key $K^*$ when given an encryption oracle for $\mathsf{Enc}$ (using real randomness) and decryption oracles. We will bound $\Pr[\mathrm{G1}^A \text{ sets } \mathsf{bad}]$ by an ae adversary $B''$ against $\mathsf{SE}$.

The adversary $B''$ works as follows.

```
Run $A$, simulating its oracles by
  $\mathbf{Enc}(M, R)$:
    $\mathcal{M} \xleftarrow{\cup} M$; $C \leftarrow \mathbf{Enc}'(M)$; $\mathcal{C} \xleftarrow{\cup} C$; return $C$
  $\mathbf{Dec}(C)$:   return $\mathbf{Dec}'(C)$
  $\mathbf{H}(R, K, M)$:   $\mathcal{K} \xleftarrow{\cup} K$; return $Y \leftarrow\$ \{0,1\}^r$
$A$ halts with output $b'$
Choose $M \notin \mathcal{M}$
Foreach $K \in \mathcal{K}$ do
  $C \leftarrow\$ \mathsf{Enc}(K, M)$; If $C \in \mathcal{C}$ then Output 0
  $M' \leftarrow \mathbf{Dec}(C)$; If $M' = M$ then Output 1
Output 0
```

That is $B''$ simulates $\mathrm{G1}^A$ except using its own oracles $\mathbf{Enc}'$ and $\mathbf{Dec}'$ to reply to $A$'s encryption and decryption queries. At the end it checks what set of oracles it has by using the keys queried by $A$ to $\mathbf{H}$. We have that $\Pr[\mathsf{QueryK}] = \Pr[\mathrm{G1}^A \text{ sets } \mathsf{bad}]$ where "$\mathsf{QueryK}$" is the event that $K^*$ is queried by $A$ in the event space defined by $\mathrm{AE}^{B''}$. Moreover, we have that $\Pr\left[\mathrm{AE1}^{B''} \Rightarrow \mathsf{true} \,|\, \mathsf{QueryK}\right] = 1$ because if the key is queried then one of $K_i$ will pass the decryption check $B''$ executes. Moreover $\Pr[\mathrm{AE0}^{B''} \Rightarrow \mathsf{false}] = 0$ because in AE0 the decryption oracle always returns $\perp$. By conditioning on $\mathsf{QueryK}$ we derive that $\mathbf{Adv}^{\mathrm{ae}}_{\mathsf{SE}}(B'') \geq \Pr[\mathsf{QueryK}] = \Pr[\mathrm{G1}^A \text{ sets } \mathsf{bad}]$. Letting $B$ be the adversary $B'$ or $B''$ with better advantage gives the theorem statement. ▌

## 6.4. Digital Signatures

A digital signature (DS) scheme is used to sign a message in an unforgeable manner. Many DS schemes use randomized signing algorithms. Randomness failures cause significant security problems for these schemes. For example, well-known secret key recovery attacks work against DSA when repeated or predictable randomness is used. This property of DSA was exploited by our attacks in Section 3. Such attacks also affect many schemes built using the Fiat-Shamir transform [29].

Note that there is a folklore technique for removing randomization from signature schemes (e.g. see [38]). It involves adding to the secret key of a randomized signature scheme a key for a secure PRF (e.g. HMAC). To sign a message, then, one generates randomness for the $\mathsf{Sign}$ algorithm deterministically by applying the PRF to the input message. Our hedged construction follows this in spirit, but crucially does not require modifying the description of the secret key. Moreover, the security achieved when randomness is fresh is better than that obtained by the folklore technique, because the assumption on Hedge is milder.

**Hedged security.** Formally, a randomized DS scheme consists of a triple of algorithms $(\mathsf{Kg}, \mathsf{Sign}, \mathsf{Vf})$. Key generation $\mathsf{Kg}$ uses randomness to generate a (public) verification key, signing key pair $(pk, sk)$. Signing $\mathsf{Sign}$ takes inputs a signing key $sk$, message $M$, and randomness $R$ and outputs a signature. We denote computing a signature by $\mathsf{Sign}(sk, M\,;\,R)$. Verification $\mathsf{Vf}$ takes inputs a verification key $pk$, a message $M$, and a signature $\sigma$ and outputs a bit. We denote verification of a signature by $\mathsf{Vf}(pk, M, \sigma)$. We hedge a scheme DS by replacing $\mathsf{Sign}(sk, M\,;\,R)$ with $\mathsf{Sign}(sk, M\,;\,\mathsf{Hedge}(R, sk, M))$. Verification remains unmodified.

DS schemes should be what is called existentially unforgeable against chosen message attacks (UF-CMA) [34]. Intuitively, this means that an adversary should not be able to forge a signature on a new message, even after seeing signatures on many chosen messages. We extend this definition to treat chosen message and chosen randomness attacks. Game UFCMRA in Figure 6 specifies UF-CMRA security in the random oracle model. A uf-cmra adversary $A$ takes input a public key, never repeats a query to the random oracle $\mathbf{H}$, and outputs a bit. Game UFCMA is the same as UFCMRA except that $R$ is not chosen by adversaries for **Sign** queries, but rather chosen freshly at random each for each query. We define the uf-cmra advantage of an adversary $A$ against signature scheme $\mathsf{DS}$ and the uf-cma advantage of an adversary $B$ also against $\mathsf{DS}$ by

$$\mathbf{Adv}^{\text{uf-cmra}}_{\mathsf{DS}}(A) = \Pr\left[\mathrm{UFCMRA}^A \Rightarrow \mathsf{true}\right] \quad \text{and}$$
$$\mathbf{Adv}^{\text{uf-cma}}_{\mathsf{DS}}(B) = \Pr\left[\mathrm{UFCMA}^B \Rightarrow \mathsf{true}\right] .$$

For any digital signature scheme DS, let $\mathsf{DS}^+$ be the hedged version of it where the hedge function is modeled by the random oracle. Then we have the following theorem.

**Theorem 6.3** Let DS be a digital signature scheme and $\mathsf{DS}^+$ be its hedged version. Let $A$ be a uf-cmra adversary against $\mathsf{DS}^+$ making at most $q_h$ hash queries. Then there exists a uf-cma adversary $B$ such that $\mathbf{Adv}^{\text{uf-cmra}}_{\mathsf{DS+}}(A) \leq$

```
Initialize:                  Sign(M, R):                      Finalize(M, σ):                  Game UFCMRA_DS
  $(pk^*, sk^*) \leftarrow_\$ \mathsf{Kg}$   $\sigma \leftarrow \mathsf{Sign}^{\mathbf{H}}(sk^*, M\ ;\ R)$   If $\mathsf{Vf}(pk^*, M, \sigma) = 1$ then
  $S \leftarrow \emptyset$                   Ret $\sigma$                          Ret true                  H(X):
  Ret $pk$                                                          Ret false                        If $\mathtt{H}[X] = \bot$ then
                                                                                                       $\mathtt{H}[X] \leftarrow_\$ \{0,1\}^r$
                                                                                                     Ret $\mathtt{H}[X]$
```

**Figure 6. Security game for digital signatures (**UFCMRA**).**

$2 \cdot \mathbf{Adv}^{\text{uf-cma}}_{\mathsf{DS}}(B)$. Adversary $B$ makes the same number of signing queries as $A$ and runs in time that of $A$ plus the time to compute $q_h$ signatures and verify each. □

**Proof:** Let $B$ work as follows. On input public key $pk^*$ it runs $A(pk^*)$. When $A$ makes a random oracle query $X$, adversary $B$ parses $X$ as a triple $(R, sk, M)$ and then use $sk$ to a sign a new message $M'$ not before queried and see if it verifies under $pk^*$. If so, halt and output the forgery. Otherwise return a random value to $A$. When $A$ makes signing queries, Adversary $B$ answers $A$'s signing queries using its own oracle (and ignoring the queried value $R$). When $A$ halts outputting a forgery attempt, $B$ outputs it.

We can see that

$$\Pr\left[\mathrm{UFCMRA}^{A}_{\mathsf{DS}+} \Rightarrow \mathsf{true}\right] \leq \Pr\left[\mathrm{UFCMRA}^{A}_{\mathsf{DS}+} \Rightarrow \mathsf{true} \mid \mathsf{QuerySK}\right] + \Pr\left[\mathrm{UFCMRA}^{A}_{\mathsf{DS}+} \Rightarrow \mathsf{true} \mid \overline{\mathsf{QuerySK}}\right]$$

where the event $\mathsf{QuerySK}$ represents the event that $A$ queries challenge $sk$ to $\mathbf{H}$ and $\overline{\mathsf{QuerySK}}$ is its complement. But the right hand terms are both bounded by $\mathbf{Adv}^{\text{uf-cma}}_{\mathsf{DS}}(B)$ since, in the first case, $B$ succeeds using the secret key and, in the second case, $B$ simulates $A$'s experiment. ▮

## 6.5. Key Exchange

A key exchange (KE) protocol involves two parties, which we'll call a client and server. The goal is for the client and server to privately agree on a secret session key. Typically this session key is used as a key for an SE scheme. There are a wide variety of key exchange protocols, but generally they can be defined via two update algorithms $\mathsf{UpdateClnt}$ and $\mathsf{UpdateSrvr}$ run by the client and server respectively. These take as input an internal state $st$, a string msgs describing all the messages sent and received thus far, and a string of random coins used for randomness. We write $\mathsf{UpdateClnt}(st, \text{msgs}\ ;\ R)$ to denote computing, using randomness $R$, the next message sent by the client when its current state is $st$ and all the messages so far sent and received are encoded in msgs. Likewise we write $\mathsf{UpdateSrvr}(st, \text{msgs}\ ;\ R)$. Then we can simply apply hedging in the now usual way, by instead running $\mathsf{UpdateClnt}(st, \text{msgs}\ ;\ \mathsf{Hedge}(R, \text{AlgID}, st, \text{msgs}))$ and similarly modify $\mathsf{UpdateSrvr}$ to use hedged randomness.

To make this concrete, we give the hedged RSA key transport and authenticated Diffie-Hellman key exchange protocols used in TLS and elsewhere. See Figure 7 for a (slightly simplified) explanation of the protocols. In RSA key transport, the client uses a server's public key to encrypt a secret value $R_1'$ that is used as the secret material to derive a session key. In Diffie-Hellman the session key is chosen as a combination of randomness chosen by both the client and the server. The server's contribution is signed using a DS scheme. Note that for brevity we do not show hedging of the (random) nonces used in these protocols, nor do we show hedging of the encryption and signing operations.

Security definitions for key exchange are complex (e.g., see [24, 27, 39]), and extending existing definitions to model randomness failures is a considerable topic of its own. We leave it to future work. That said, we can make several meaningful statements about security. First, hedging does not hinder meeting traditional goals here as per the discussion in Section 6.1. Second, hedging ensures that key transport will not send the same session key to different servers, meaning in particular a hedged TLS client from Section 3 will never send the same PMS to different servers. (For the server attack, hedging DSA already protects from extraction of the secret key.)

## 7. Implementing Hedging in OpenSSL

We report on a case study of implementing hedging within the OpenSSL code base. This implementation consists of a cryptographic tools library (the OpenSSL crypto library), and a TLS library (the OpenSSL ssl library). The former is widely used within security applications beyond TLS, and so hedging it can have far-reaching consequence. Besides hedging an important code base, this case study will allow us to evaluate the performance impact of hedging in-use cryptographic tools. Looking ahead, hedging appears to have little performance impact for numerous operations and usage scenarios. This case study indicates that many cryptographic deployments can easily support hedging, motivating the adoption of hedging to protect against unforeseen RNG failures.

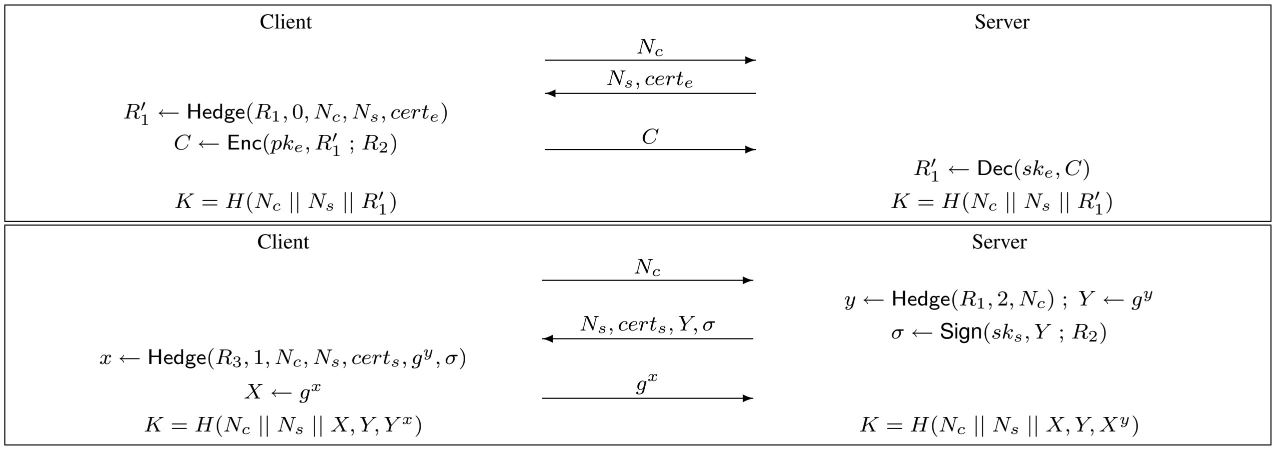
Client
Server
$N_c$
$N_s, cert_e$
$R_1' \leftarrow \mathsf{Hedge}(R_1, 0, N_c, N_s, cert_e)$
$C \leftarrow \mathsf{Enc}(pk_e, R_1' \; ; R_2)$
$C$
$R_1' \leftarrow \mathsf{Dec}(sk_e, C)$
$K = H(N_c \mid\mid N_s \mid\mid R_1')$
$K = H(N_c \mid\mid N_s \mid\mid R_1')$
Client
Server
$N_c$
$y \leftarrow \mathsf{Hedge}(R_1, 2, N_c) \; ; \; Y \leftarrow g^y$
$N_s, cert_s, Y, \sigma$
$\sigma \leftarrow \mathsf{Sign}(sk_s, Y \; ; R_2)$
$x \leftarrow \mathsf{Hedge}(R_3, 1, N_c, N_s, cert_s, g^y, \sigma)$
$X \leftarrow g^x$
$g^x$
$K = H(N_c \mid\mid N_s \mid\mid X, Y, Y^x)$
$K = H(N_c \mid\mid N_s \mid\mid X, Y, X^y)$

**Figure 7. Hedging (top) RSA key transport and (bottom) signed Diffie-Hellman authenticated key exchange. We use $N_c, N_s$ to denote nonces and $R_1, R_2, R_3$ to denote RNG-derived randomness. Application of Hedge includes numbers 0,1,2 as stand-ins for unique algorithm identifiers. (Top) The server has a certificate $cert_e$ for a PKE scheme with keys $(pk_e, sk_e)$. The client checks this certificate, aborting if the check fails. (Bottom) The server has a certificate $cert_s$ for a DS scheme with keys $(pk_s, sk_s)$. The client checks the certificate and signature $\sigma$, aborting if either fails.**

Our starting point was the current stable version of OpenSSL at the time of writing, version 0.9.8k. We hedged the most commonly utilized operations: RSA public-key encryption with PKCS#1 v1.5 randomized padding, RSA OAEP public-key encryption, DSA signing, and the ephemeral Diffie-Hellman (DHE) key exchange client and server. The first three operations involved only modifying code within the crypto library while the key exchange code is in the ssl library. Note that TLS 1.0 only uses stateful symmetric encryption and as such this version of OpenSSL does not implement randomized symmetric encryption primitives[1]. Future work could evaluate the hedging of symmetric encryption primitives in TLS 1.1 or 1.2 or in other tools, also comparing them to custom solutions proposed by [48]. (These solutions will likely be included in future versions of TLS.)

Our hedged version of OpenSSL 0.9.8k will be referred to as "hedged", while "plain" refers to the original, unmodified OpenSSL 0.9.8k. We used the cloc utility for counting lines of code [5] to get a sense of the scale of modifications due to hedging. The hedged ssl library added 45 more physical lines of code and the hedged crypto library had 728 more physical lines of code.

In the following we use two machine configurations for benchmarking. Our server system is a Pentium 4 2.0 GHz with 1 GB of RAM running Ubuntu Linux 8.04 Server. It ran Apache 2.2.13 with mod_ssl built from either the hedged OpenSSL or plain OpenSSL library, with both RSA and DSA keys setup, and all other options set to their defaults. We recompiled Apache when switching between libraries. Our client systems are Dual Pentium 4 3.20 GHz systems with 1 GB of RAM running Ubuntu Linux 8.04 Desktop.

**Performance of Hedge implementations.** Recall that Hedge makes black-box use of HMAC, which in turn uses an underlying hash function. We investigate three natural choices for this hash function: SHA-1, SHA-256, and SHA-512. Recent attacks [49] mean SHA-1 is no longer considered secure. We do not recommend its use but include it for the sake of comparison. New hash functions are being designed for an eventual SHA-3 standard [2]; one can easily upgrade Hedge to use newer hash functions.

We first report on a naïve implementation of Hedge that simply iterates HMAC a sufficient number of times following the description in Section 5. The left graph in Figure 8 depicts the performance of this implementation when using each of the hash functions and when requesting various amounts of random bytes. The benchmarks were performed on one of the client machines. The amount of associated data was set to 3,000 bytes. (The primitives we hedge never supply more than this amount of associated data for standard key lengths.) As expected, SHA-1 is the fastest. SHA-256 provides little performance benefit over SHA-512 for small output lengths and is significantly slower as output length increases. This is because SHA-512 generates more output per iteration.

[1]For example, the CBC block cipher mode of TLS 1.0 does in fact utilize a randomized IV, but it is generated only once per session during key exchange. Further packets are encrypted using as IV the last block of the previous ciphertext. So in this version of TLS, hedging key exchange effectively hedges the symmetric encryption routines as much as is possible. For further details see [8] and [41].

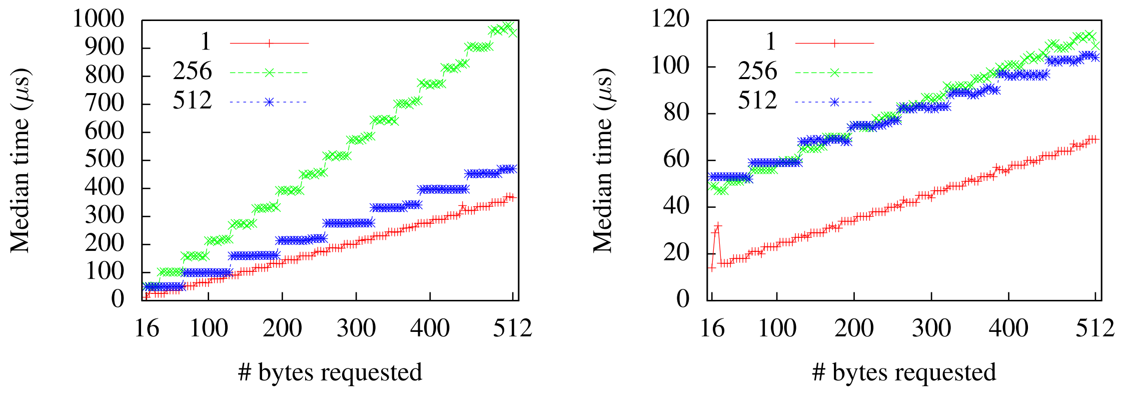

**Figure 8. Comparison of Hedge implementations when requesting various numbers of random bytes and including 3,000 bytes of associated data and using SHA-x for $x \in \{1, 256, 512\}$. Time is median of 1,000 executions after 100 untimed executions. (Left) A naïve implementation that iterates HMAC for each extra block of output needed. (Right) An implementation that reuses internal HMAC state between iterations.**

As the length of output increases, performance severely degrades. We can speed up the implementation with a simple optimization. Note that when executing Hedge all the input to HMAC modulo the iteration counter remains the same for each application of HMAC. Thus we modify the HMAC code to allow computing it up through all of the common values once, and then finishing the computation for each iteration separately. The right graph in Figure 8 depicts the performance of this implementation. (The outliers in the lower left corner were due to unrelated system activity.) As can be seen the improvement is large, and so we utilize this latter implementation for the rest of our tests.

We recommend SHA-512 for greatest security and utilize it for all further benchmarks. Of course, we again emphasize that any user deploying hedging can utilize any (secure) hash function they desire.

**Performance of asymmetric primitives.** We measure the performance overhead of hedging specific asymmetric cryptographic operations: RSA encryption with PKCS#1 v1.5 randomized padding, RSA encryption with OAEP padding, DSA signing, and ephemeral Diffie-Hellman key generation (used in key exchange). For brevity we refer to these operations at PKCS#1, OAEP, DSA, and DHE. Figure 9 shows the results of timing 1,000 repetitions of each operation either without hedging ("Plain") or with hedging ("Hedged"). For both, we always performed an additional 100 repetitions before the 1,000 measured iterations. (This lowered variability in standard deviations.) As one might expect, the overhead due to hedging decreases quickly with increased key size (and, thereby, running time) of the underlying operation. The largest overheads are for client-side operations such as PKCS#1 and OAEP with 1024 bit keys. PKCS#1 has slightly worse performance because one needs to generate more randomness compared to OAEP. The typical server-side operations (where performance tends to matter most) of DSA and DHE have low overhead.

**TLS handshake performance.** The microbenchmarks just given suggest that performance of individual OpenSSL handshakes will not be degraded significantly. To be sure, we measured the time for a client to open a TLS connection with a remote server. Figure 10 reports the results for doing so with both plain TLS and hedged TLS (both the client and server using hedged operations). Here AES128-SHA refers to key exchange using RSA PKCS#1 encryption with 1024-bit RSA keys, DHE-RSA-AES128-SHA refers to key exchange using 1024-bit ephemeral Diffie-Hellman with RSA signing, and DHE-DSS-AES128-SHA refers to key exchange using 1024-bit ephemeral Diffie-Hellman with DSA signing. (AES128-SHA suffixes refers to the symmetric encryption mechanism, which does not affect these timings.) As expected, there is no discernible difference between plain times and hedged times.

**Server overhead.** Individual connections are not slowed down by hedging, but it could be that the extra CPU costs due to hedging significantly burdens heavily loaded servers. We measure average throughput of our Apache HTTPS server when built against plain OpenSSL and against hedged OpenSSL. Note that only the two Diffie-Hellman key exchange protocols have server-side hedging, and so we report only on these. Figure 11 depicts the results of saturating the server with HTTPS requests for a 44-byte HTML file. To perform this experiment, we setup two client

| Operation | Plain time (μs) Median (Min,Mean,Max,Std. Dev) | Hedged time (μs) Median (Min,Mean,Max,Std. Dev) | Ratio |
|---|---|---|---|
| PKCS#1 1024 | 140 (138,141,556,13) | 185 (183,186,301,6) | 1.32 |
| PKCS#1 2048 | 415 (412,417,720,20) | 478 (476,483,722,19) | 1.15 |
| PKCS#1 4096 | 1589 (1580,1591,1836,18) | 1686 (1678,1694,1919,31) | 1.06 |
| OAEP 1024 | 140 (139,140,189,2) | 179 (178,179,254,3) | 1.28 |
| OAEP 2048 | 410 (409,412,646,10) | 457 (455,458,673,12) | 1.11 |
| OAEP 4096 | 1579 (1572,1581,1804,18) | 1632 (1625,1634,1887,19) | 1.03 |
| DSA 1024 | 1324 (1264,1325,1576,23) | 1426 (1381,1429,1682,27) | 1.08 |
| DSA 2048 | 4025 (3898,4027,4441,55) | 4156 (4026,4164,4738,68) | 1.03 |
| EDH 1024 | 7937 (7910,7948,8616,60) | 8002 (7976,8010,8656,56) | 1.01 |

**Figure 9. Comparison of asymmetric cryptographic operations without hedging ("Plain") and with hedging ("Hedged"). All values are time in microseconds measured over 1,000 repetitions. "Ratio" is the median hedged time divided by median plain time.**

| Operation | Plain time (μs) Median (Min,Mean,Max,Std. Dev) | Hedged time (μs) Median (Min,Mean,Max,Std. Dev) |
|---|---|---|
| AES128-SHA | 6941 (6875,6989,8380,231) | 6968 (6890,7310,11334,920) |
| DHE-RSA-AES128-SHA | 52030 (51756,52120,63388,470) | 52828 (51150,52618,62841,735) |
| DHE-DSS-AES128-SHA | 50907 (50567,50959,64224,471) | 51067 (50011,51010,62020,673) |

**Figure 10. Measuring TLS connection time without hedging ("Plain") and with hedging ("Hedged"). Measurements were performed on the client over 1,000 executions. The server had 1024-bit RSA and DSA keys.**

systems running httperf [6] and administered them using the autobench tool [4]. For each rate httperf attempted 3,000 connections with a timeout of 1 second. As can be seen in the graphs, the server was quickly saturated both when performing RSA signing with DHE and DSA signing with DHE. The former became saturated slightly sooner than the latter, perhaps due to RSA's more expensive signing operation. In both cases the hedged server performed as well as the plain server. Note that the server in this experiment was entirely unoptimized, and so this experiment may not be indicative of a performance gap on a fully optimized server. Nevertheless it reveals that for an "out-of-the-box" TLS deployment there is no significant overhead when hedging.

## 8. Conclusions

This paper had two main contributions. First, we revealed the first virtual machine reset vulnerabilities affecting deployed software. Given repeated use of a VM snapshot, we showed how attackers can compromise TLS sessions or even extract a server's secret DSA authentication key. The vulnerabilities stem from a combination of factors. First, applications cache to-be-used randomness long before consumption or do not add enough new entropy to their RNGs right before use. Second, the cryptographic operations that consume this randomness are fragile in the face of the ensuing randomness reuse.

Our second contribution was dealing with this latter problem, the endemic fragility of routine cryptographic operations when given bad randomness. We developed a general framework for hedging cryptographic operations. Our hedging approach is simple and incrementally deployable, and it provides provably better resistance to RNG failures for important primitives. As our implementation within OpenSSL indicates, hedging is fast.

Moreover, we feel that hedging is needed. Generating randomness is inherently complex, as indicated by a long history of RNG failures. Future RNG problems will almost certainly arise, and, as we exemplified by our results on VM reset vulnerabilities, new technologies create new problems. Ensuring that cryptography is built to provide as much security as possible for any given quality of randomness will effectively limit the damage done by future RNG failures.

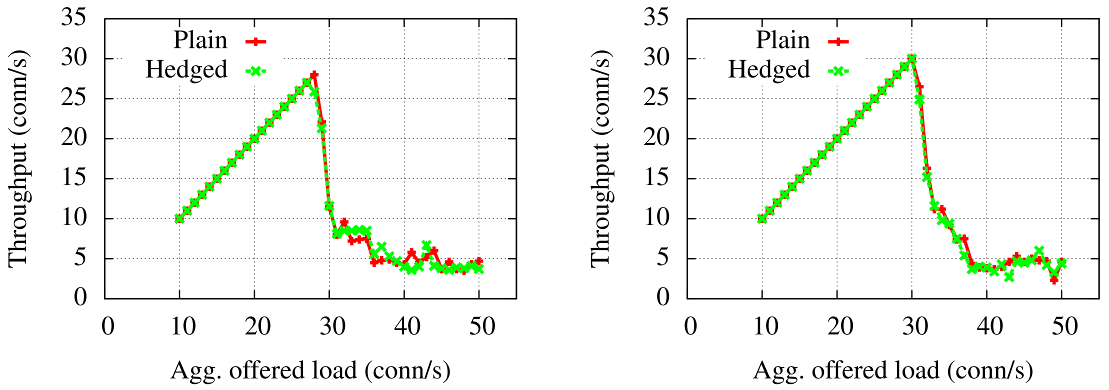

**Figure 11. Saturating an Apache 2 server (in default configuration) with HTTPS requests when using plain mod_ssl and hedged mod_ssl.** (Left) **Using RSA signing with ephemeral Diffie-Hellman (DHE-RSA-AES128-SHA).** (Right) **Using DSA signing with ephemeral Diffie-Hellman (DHE-DSA-AES128-SHA).**

## Acknowledgements

Thomas Ristenpart is supported by Mihir Bellare's NSF grant CNS–0627779 and a gift from Intel Corporation. Scott Yilek is supported by Daniele Micciancio's NSF grant CNS–0831536 and Mihir Bellare's NSF grant CNS–0627779.

## References

[1] `http://www.rackspacecloud.com/`.
[2] `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.
[3] Amazon EC2. `http://aws.amazon.com/ec2/`.
[4] Autobench. `http://www.xenoclast.org/autobench/`.
[5] CLOC. `http://cloc.sourceforge.net/`.
[6] httperf. `http://www.hpl.hp.com/research/linux/httperf/`.
[7] Microsoft azue. `http://www.microsoft.com/azure/`.
[8] The TLS Protocol, Version 1.0. `http://www.ietf.org/rfc/rfc2246.txt`.
[9] The TLS Protocol, Version 1.2. `http://tools.ietf.org/html/rfc5246`.
[10] VirtualBox. `http://www.virtualbox.org/`.
[11] VirtualPC. `http://www.microsoft.com/windows/virtual-pc/`.
[12] VMWare. `http://www.vmware.com`.
[13] Vulnerability note vu925211: Debian and ubuntu openssl packages contain a predictable random number generator. `https://www.kb.cert.org/vuls/id/925211`.
[14] FIPS PUB 186-3. Digital signature standard (DSS). `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, 2009.
[15] ISO/IEC 9797. Data cryptographic techniques – data integrity mechanism using a cryptographic check function employing a block cipher algorithm, 1989.
[16] Paolo Abeni, Luciano Bello, and Maximiliano Bertacchini. Exploiting DSA-1571: How to break PFS in SSL with EDH, July 2008. `http://www.lucianobello.com.ar/exploiting_DSA-1571/index.html`.
[17] Andrew Becherer, Alex Stamos, and Nathan Wilcox. Cloud computer security: Raining on the trendy new parade. BlackHat USA 2009, Slides available from `http://www.slideshare.net/astamos/cloud-computing-security`.
[18] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. In *CRYPTO 2006*. Springer, 2006.
[19] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedge public-key encryption: How to protect against bad randomness. In *ASIACRYPT 2009*. Springer, 2009. To Appear.
[20] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO 1996*, pages 1–15. Springer, 1996.
[21] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "pseudo-random" number generation within cryptographic algorithms: The dss case. In *CRYPTO 1997*. Springer, 1997.
[22] Mihir Bellare and Tadayoshi Kohno. A theoretical treatment of related-key attacks: Rka-prps, rka-prfs, and applications. In *EUROCRYPT 2003*, pages 491–506. Springer, 2003.
[23] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security – CCS 1993*, pages 62–73. ACM, 1993.
[24] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO 1993*, pages 232–249. Springer, 1994.
[25] Mihir Bellare and Phillip Rogaway. Code-based game-

playing proofs and the security of triple encryption. In *EUROCRYPT 2006*. Springer, 2006.

[26] Daniele R.L. Brown. A weak randomizer attack on RSA-OAEP with e=3. IACR ePrint Archive, 2005.

[27] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *EUROCRYPT 2001*, pages 453–474, 2001.

[28] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the windows random number generator. In *CCS 2007*. ACM, 2007.

[29] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO 1986*, pages 186–194. Springer, 1986.

[30] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO 1984*, pages 10–18. Springer, 1985.

[31] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems – HotOS-X*, May 2005.

[32] Ian Goldberg and David Wagner. Randomness and the netscape browser. Dr. Dobb's Journal, January 1996.

[33] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[34] Shafi Goldwasser, Silvio Micali, and Ron Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, 1988.

[35] Zvi Gutterman and Dahlia Malkhi. Hold your sessions: An attack on java session-id generation. In *CT-RSA 2005*. Springer, 2005.

[36] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *Symposium on Security and Privacy 2006*. IEEE, 2006.

[37] Seny Kamara and Jonathan Katz. How to encrypt with a malicious random number generator. In *FSE 2008*. Springer, 2008.

[38] Jonathan Katz and Nan Wang. Efficiency improvements for signature schemes with tight security reductions. In *CCS 2003*. ACM.

[39] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *ProvSec 2007*, pages 1–16. Springer, 2007.

[40] David A. McGrew and John Viega. The security and performance of the galois/counter mode (gcm) of operation. In *INDOCRYPT 2004*, pages 343–355. Springer, 2004.

[41] B. Moeller. Security of cbc ciphersuites in ssl/tls: Problems and countermeasures. `http://www.openssl.org/~bodo/tls-cbc.txt`.

[42] Markus Mueller. Debian OpenSSL predictable PRNG bruteforce SSH exploit, May 2008. `http://milw0rm.com/exploits/5622`.

[43] Mark R.V. Murray. An implementation of the yarrow prng for freebsd. In *BSDCon 2002*. USENIX, 2002.

[44] NIST. Recommendations for block cipher modes of operation. `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`, 2001.

[45] Khaled Ouafi and Serge Vaudenay. Smashing SQUASH-0. In *EUROCRYPT 2009*. Springer, 2009.

[46] Phillip Rogaway. Nonce-based symmetric encryption. In *FSE 2004*, volume 3017, pages 348–359. Springer, 2004.

[47] Phillip Rogaway, Mihir Bellare, and John Black. Ocb: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[48] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. In *EUROCRYPT 2006*. Springer, 2006.

[49] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *CRYPTO 2005*. Springer, 2005.

[50] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *J. of Comp. and System Sciences*, 22:265–279, 1981.

[51] Robert Woolley, Mark Murray, Maxim Dounin, and Ruslan Ermilov. arc4random predictable sequence vulnerability. `http://security.freebsd.org/advisories/FreeBSD-SA-08:11.arc4random.asc`, 2008.

[52] Scott Yilek. Resettable public-key encryption: How to encrypt on a virtual machine. In *Topics in Cryptology – CT-RSA 2010*. Springer, 2010. To Appear.

[53] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When Private Keys are Public: Results from the 2008 Debian OpenSSL Vulnerability. In *IMC 2009*, pages 15–27. ACM, 2009.

## A. DSA Key Recovery Attacks

We review the Digital Signature Algorithm (DSA) [14]. For simplicity, we focus on the 1024-bit case. Let $p$ be a 1024-bit prime, $q$ a 160-bit prime that divides $p-1$, and $g$ an integer with order $q$ mod $p$. These are the parameters. The private key is chosen as a random $x \in \mathbb{Z}_q$, and the public key is $y = g^x \mod p$.

To sign a message $M$, first hash $M$ into a 160-bit value $H(M)$. Then choose a random value $k \in \mathbb{Z}_q$ and compute $r = (g^k \mod p) \mod q$ and $s = (k^{-1}(H(M) + xr)) \mod q$. The signature is $(r, s)$, a pair of at most 160-bit values. We omit the description of the verification algorithm.

DSA is believed to be a secure signature scheme, however it is well-known that if the randomness $k$ is known or even generated by some types of weak RNGs, then an adversary can extract the signer's secret key [21].

It is also the case that if an adversary sees two signatures that use the same $k$ for different messages, then he can efficiently extract the secret key. To see this, consider two signatures $(r, s)$ and $(r, s')$ over messages $M$ and $M'$ hashing to distinct values $H(M)$ and $H(M')$, respectively. An adversary can then compute $k = (H(M) - H(M'))(s - s')^{-1} \mod q$.