Missive: Fast Application Launch From an Untrusted Buffer Cache

Jon Howell, Jeremy Elson, Bryan Parno, John R. Douceur
Microsoft Research, Redmond, WA

Abstract

The Embassies system [18] turns the web browser model inside out: the client is ultra-minimal, and hence strongly isolates pages and apps; every app carries its own libraries and provides itself OS-like services. A typical Embassies app is 100 MiB of binary code. We have found that the first reaction most people have upon learning of this design is: how can big apps start quickly in such a harsh, mutually-untrusting environment?

The key is the observation that, with appropriate system organization, the performance enhancements of a shared buffer cache can be supplied by an untrusted component. The benefits of sharing depend on availability of commonality; this paper measures a hundred diverse applications to show that applications indeed exhibit sufficient commonality to enable fast start, reducing startup data from 64MiB to 1MiB. Exploiting that commonality requires careful packaging and appropriate application of conventional deduplication and incremental start techniques. These enable an untrusted client-side cache to rapidly assemble an app image and transfer it—via IP—to the bootstrapping process. The result is proof that big apps really can start in a few hundred milliseconds from a shared but *untrusted* buffer cache.

1 Introduction

When a user installs a new desktop application, he accepts the risk that the new app may compromise any other app he uses. In contrast, web sites he visits are responsible for managing their own servers; a visit to a new site doesn't present a threat to the servers that run other sites he uses. A site manager is better equipped than her users to make security decisions about her server, and the server's isolation gives her the autonomy to effect those decisions.

This benefit should accrue to the web as a whole, except that the client side is bloated and vulnerable; thus clicking a web link can be as risky as installing a desktop app. The Embassies project [18] proposed refactoring the web client interface to isolate client-side apps as effectively as servers are isolated in multitenant data centers, so that the site manager becomes autonomously responsible for her client code, too. We call this model the "pico-datacenter" – the client becomes a hosting site for mutually distrustful applications, providing no semantics other than a VM-like container, IP and the thinnest UI interface (each app paints raw pixels on its part of screen).

The Embassies design aims to mimic the relationships among software components found in a shared data center: each vendor enjoys strong isolation, retaining autonomy even as it communicates with other vendors. This isolation promises to protect Embassies from the bloat that afflicts prior client models; but it demands a truly minimal host.

Unlike Embassies' pure shared-nothing model, though, existing web clients extract substantial performance benefits from sharing. The host operating system's buffer cache and the browser's HTTP object cache share content across sites. The lumbering 100 MiB browser process itself is shared, since it need not restart for each new site the user visits. Many people's first reaction to the Embassies proposal is alarm at the idea of shipping such big apps around; surely it must lead to unbearably slow app launch times?

Surprisingly, such big apps can be started nearly as quickly as a conventional web page. It is one thing to make an abstract argument that it should be possible; the aim of this paper is to decisively demonstrate so. This paper shows that ideal isolation does not funamentally conflict with good application-launch performance.

We construct a content cache that is untrusted (as untrusted as a random neighboring tenant in a shared datacenter), and yet enables mutually distrustful sites to share content and reap the benefit of fast app launch, while using end-to-end cryptographic checks to protect their own integrity. Essentially, we show that the OS buffer cache and browser object cache can be evicted from the trusted computing base (TCB) and replaced with an untrusted cache that delivers similar performance benefits.

For the untrusted cache to be effective, there must be commonality to exploit; we must demonstrate the performance equivalent of many applications sharing a common browser infrastructure. This paper

- demonstrates that a hundred diverse applications exhibit great commonality, enabling efficient transfers and fast launches
- shows an integrated pipeline that packages, transfers, caches, and launches large application images in a secure manner.
- characterizes the sensitivity of performance to

pipeline parameters, and

- demonstrates hot- and warm- app launch times comparable to that of a conventional OS buffer cache and shared library mechanisms (in which apps are mutually trusting).

Missive is best motivated by the extreme minimality of Embassies [18], but it applies more broadly. Other client app delivery systems such as Tahoma [10], Xax [17], Native Client [43], and Drawbridge [33] ship VMs or large binary programs, and an untrusted cache would reduce their TCBs. VM images in any context are big, and launch times can be slow [4, 24]. Fast launch is particularly relevant for security applications that spawn a VM per user [32] or per connection attempt [41].

2 Context

We focus on Missive's applicability to the Embassies client application environment [18], since it takes host minimality to the extreme, making a shared cache particularly challenging.

2.1 Embassies Overview

With Embassies, apps are strongly isolated, communicating with other apps and with the outside world only via IP. The intent is to create an environment analogous to the server app environment, where each vendor is completely isolated from other vendors and exercises full control over its own software stack: If a server app is compromised, it is because the vendor chose a poor library, misconfigured a firewall, or failed to patch its software. No careless or ignorant user decision can be blamed. By analogy, on an Embassies client, the user's decision to open a new app cannot compromise any other app on the client, since the apps are as isolated as two tenants on a hosting server. The simple communication semantics of IP make it clear how a vendor protects itself: if a vendor's software selection and administration can protect its server-side software from attacks arriving over the Internet, then the same follows for its Embassies client software.

The Embassies model deviates from the server-side model in a few respects. For example, it offers apps raw, pixel-level access to the display for interactivity. However, the most important distinction is the workload; typical server software multiplexes many users over one installation of long-lived code and database. In contrast, at the client, we expect the user to frequently switch context between apps and to often launch altogether new apps (analogous to clicking on links in a conventional web browser). Worse, these apps are likely to be large: Rather than a skinny JavaScript atop a big shared browser or a small executable atop a dozen shared libraries, each app is more like a standalone Virtual Machine (VM) image. However, in pursuit of strong isolation, the Embassies client platform aims for minimality, which obstructs conventional performance optimizations that tightly couple sharing of libraries and caches.

Indeed, the Embassies client omits facilities for a buffer cache or wide area transfer (MIME, HTTP, or even TCP). Missive fills that gap, showing how *mutually-untrusting* applications can cooperate to exploit the sharing opportunities that lead to good performance.

2.2 Embassies App Start

Figure 1 shows how an app starts on an Embassies client. First, some *invoking* app A, perhaps one in which the user has clicked a link, identifies a public key that represents the target app B; it also fetches B's signed boot block. It is app A's responsibility (not the client kernel's) to verify that it has found the correct principal (here it uses DNSSEC).

Second, in steps 2 through 5, the kernel receives the signed boot block, checks the signature, associates a fresh process with the signature's public key (B), and begins executing the boot block in the new process. This is the only phase whose shape is imposed by the Embassies TCB.

Third, in steps 7 through 10, the untrusted cache gathers the metadata and data required to assemble the image for App B. None of these interactions require integrity other than to avoid wasting the cache's time.

Finally, the cache sends the entire image in a single IPv6 jumbo packet into App B's process. App B then verifies the image's integrity, for example, by checking the image's hash against a hash value included in the boot block. Finally, App B transfers control to the code in the image.

For communication between apps and to the greater Internet, the client kernel provides only an untrusted IP channel. The mapping of name to app key, fetching of boot block, and fetching of image content are all built from the IP primitive, making it easy to reason about isolation.

The client kernel provides no storage; instead, it is assumed that some anonymous vendor (e.g., Seagate) provides an untrusted, insecure storage app. The client kernel does provide each application with a single secret specific to the host and the app's identity:

$$K_{app} = \text{PRF}_{K_{host}}(\text{ID}_{app}).$$

That secret is only available to processes started from a suitably-signed boot block. The secret enables the app to convert untrusted storage into secure storage via encryption and cryptographic integrity checks, while requiring the client kernel to store no per-app state.

These are all of the primitives Embassies offers for the app-start process. Missive's mission is to provide high performance app starts from only these primitives. One

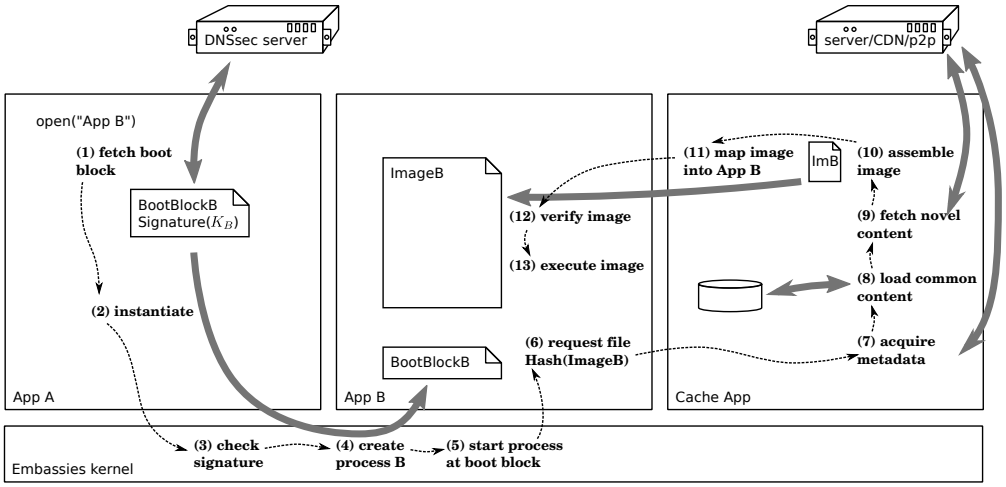

Figure 1: **App start process.** *To launch a new process for App B, App A fetches a signed boot block for B, the kernel verifies the signature, the cache assembles the required full application image, and sends it to App B to execute. Dashed lines show control flow. Heavy grey arrows show data flow. All data flows, other than the boot block passing through the kernel, are via IP.*

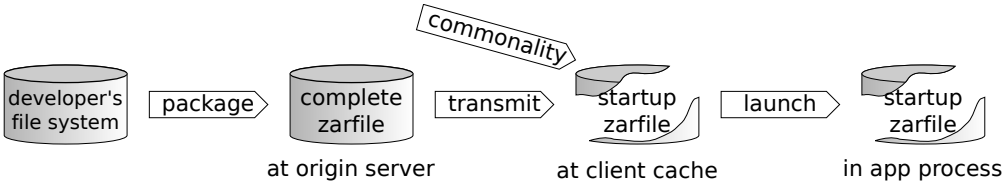

Figure 2: **System diagram.** *The developer's files that comprise the app image are packaged into a "zarfile". The client cache fetches the part of the zarfile required for app start, and the zarfile is delivered into an isolated process to launch the app.*

naïve approach would appoint a distinguished app vendor to supply the trusted cache, but then to enjoy the performance benefit of a shared cache, apps must trust the cache vendor; it becomes an implicit part of the TCB. Instead, Missive's architecture lets every app exploit a single shared cache, without trusting that cache.[1]

3 System Design

Missive comprises three steps (Figure 2): The *packaging* step collects binary libraries and data files from a developer's machine into an image, called a *zarfile*. Files are placed in the zarfile to expose sharing opportunities. The *transmission* protocol transmits zarfiles across the network; it is designed to minimize round trips, exploit commonality to minimize bandwidth, and enable incremental access. The *launch* procedure transfers a zarfile from an untrusted cache into the booting app's process, with a focus on minimizing the startup latency.

3.1 Packaging

An Embassies app is completely specified by its vendor. Thus, the vendor can configure the app on a development machine using any available framework or tool. She might install required framework components, such as a Python math library; she might carefully select a specific version of a subpackage (such as an audio rendering library); and she might even hand-patch a component or configuration file to fix a security vulnerability.

Once all of the components are in place, the vendor runs the packaging tool, which enumerates the set of files that comprise the app, including the app executable, data files, libraries, and library OS components [17, 20, 33]. This is the *complete* application image. The tool also captures a dynamic run of the application, identifying the subset of the complete image necessary to bring the app to a usable interactive state; this is the *startup* set. The startup set is captured at sub-file granularity so that it skips chaff such as symbols.

By identifying the startup set, Missive enables the developer to reduce the size (and increase the speed) of the initial app transfer. After app launch, the remaining components may be fetched from the complete image on demand, or preemptively in the background, so that they will be available when the client is disconnected.

Missive's zarfile is a simple tar-like format. It specifies a master index, string and data-chunk lookup tables, file `stat` metadata, and file contents.

Below, we elaborate on the challenges involved in im-

[1] Missive does not prevent side channels: by probing the cache's response time, one app can learn about content accessed by others.

age capture and ensuring zarfile stability. The capture process also honors memory layout constraints designed for fast app start as described in §3.3.3.

3.1.1 Image Capture

Some tuning is required to extract a minimally-sized image from a conventional POSIX development system. For example, we found that the Gnome system-wide icon cache may be 50 MB, but a single app may access only a few kilobytes of icons from it; our packaging tool strips the icon cache apart to avoid the waste. Similar techniques could be applied (although we have not yet done so) to strip unneeded code from shared libraries. (On the other hand, leaving libraries untuned may enhance commonality; see §4.)

3.1.2 Image Stability

As §4 discusses in detail, a critical component of Missive's good performance is detecting common content shared between indifferent apps. To facilitate this detection, Missive's packaging tool is aware of the block size used during transmission (§3.2), and it strives to ensure that small changes in file selection, file content, or file length will produce zarfiles in which most blocks have the same content and location.

Block Content Stability. In typical file-size distributions almost all files are small files, but a few large files comprise almost all the bytes [2, 12, 36], and our file set is no exception (Figure 3). The tail of tiny files makes it impractical to give every small file its own block; padding would expand the image by $2 - 10\times$ (§5.2), wasting too much physical memory.

Instead, Missive's packager aligns large files—those bigger than a block—on block boundaries to maximize commonality detection, and it uses small files to fill in the gaps left at the end of large files. While some extra space still remains, in practice, the overhead of padding in a zarfile is generally below 2% (§5.2). In wide-area transit, the wasted bytes are compressed away, while the effect on disk is negligible. In memory, the bytes are moved cheaply by reference, so overall, the layout has little performance penalty.

The benefit of this layout is that it makes it likely that, if two zarfiles share many large input files, then they contain proportionally many identical blocks. Furthermore, a change in a small file affects only the block whose tail it occupies. Thus, for each file different between two images, the zarfiles differ by $\lceil \frac{file\_len}{block\_size} \rceil$ blocks.

In summary, the packaging tool ensures that two similar zarfiles share almost all of their aligned blocks. Therefore, launching an app similar to one already cached requires transmitting bytes proportional only to the amount the images differ.

Block Position Stability. The block-aligned layout policy described above is close to what we want, but it leaves

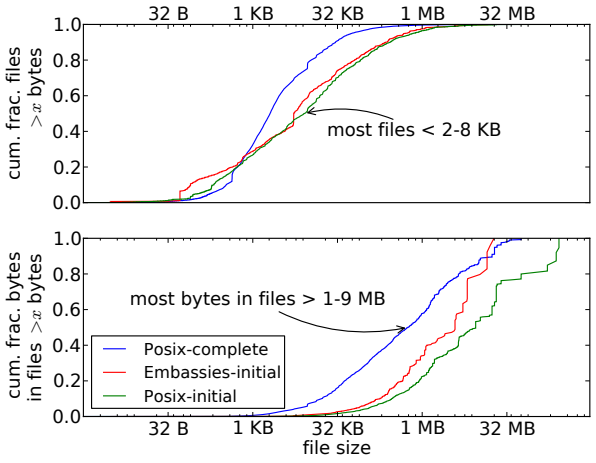

Figure 3: **File size distribution.** *All of our experimental data sets (details in §4.1) obey the typical distribution: almost all files are small, and almost all bytes belong to the few big files.*

three problems. First, a change in the size of a big file or the set of small files may cascade, changing all of the tail-gap assignments for the small files. Second, perturbing the file order perturbs all of the Merkle hashes in the transmission phase (§3.2), foiling opportunities to share Merkle subtrees.

Third, and most importantly, since we strive so enthusiastically for minimality, we cannot assume the kernel supports a `gather-send` operation. Yet we still wish to extract maximum performance. Absent gather, if two zarfiles share most of their blocks, but those blocks are reordered, construction will require the cache to copy all of the blocks into the correct offset in the outgoing message. For our parameters, the copying alone can add 100–150 ms to startup latency. When we provide position stability, the cache exploits it in the warm case by assembling the first zarfile in a buffer with some slack memory at the end. Then, when a request for the second zarfile arrives, the cache need only patch the blocks that differ from the first zarfile.

To foster position stability, we refine Missive's placement algorithm to produce zarfiles that not only share blocks with common content, but whose common blocks will appear at common offsets. Let t be the total size of the input files, i.e., the minimum size of the output. Let u be the nearest power of two greater than t, and l be the nearest power of two less than t. Pick a random $seed$, and hash each input file f along with $seed$, producing $h_f = hash(f||seed)$. If $h_f \bmod u < t$, then the file's preferred placement is $h_f \bmod u$; otherwise, the preferred placement is $h_f \bmod l$. We truncate all preferred placements to block boundaries to produce, for each file, a preferred block-aligned location in the range $[0, t)$. For each file, we evaluate the placement expression with ten seeds to produce a prioritized list of preferred locations.

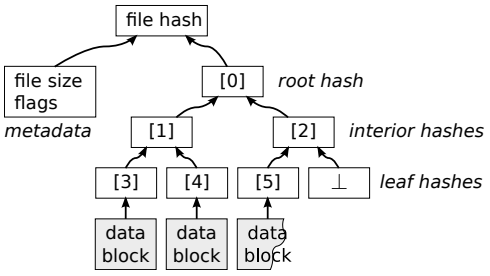

Figure 4: **Image file protocol representation.** *Each arrow represents an input to a hash operation. The Missive transmission protocol represents a zarfile with a conventional Merkle tree, plus the metadata required to define the tree's dimensions. After learning the metadata, the receiver can query hashes and data blocks in any order, and verify them incrementally.*

The input files are placed into the zarfile greedily, by descending file size. If a file collides with a prior placement, we try its lower-priority placements. If no preferred placements work, the file is dumped into a leftover bucket. After every file's preferred placements have been attempted, the leftover files are placed into gaps or concatenated to the end of the file.

Intuitively, the algorithm ensures that: (a) A change to a small file will, due to the greedy placement order, make small changes to the overall zarfile, since most bytes have already been placed by the time the change impacts the algorithm. (b) A change to a large file perturbs the algorithm early, but only affects that file and those later files whose placement depend on a collision created or eliminated by the change. This contrasts with the basic greedy algorithm where any change affects *every* later placement decision. Conversely, a change to a large file that preserves its length will not perturb the basic algorithm, but will perturb the position-stable algorithm.

3.2 Transmission

Once the zarfile is defined, it must be fetched to the client; this occurs using well-understood techniques: To preserve the integrity of the vendor's app image specification, zarfiles are specified by content. To exploit commonality, the content is specified by hashing blocks; blocks already cached at the receiver because they're common with other apps do not require transmission. Finally, to enable the app to fetch subsets of the zarfile (such as fetching the startup set from inside the complete zarfile), the block hashes are arranged into a Merkle tree [27]. The file is self-certified [16] by the *file hash*, i.e., a hash of the Merkle tree root and the file metadata (Figure 4). The metadata consists of the file length and a flag indicating whether the file contents should be interpreted as a directory.

The Missive transmission protocol is packet-based and incrementally self-verifying (Figure 5). In the first round, the receiver asks for a file by its file hash, and receives

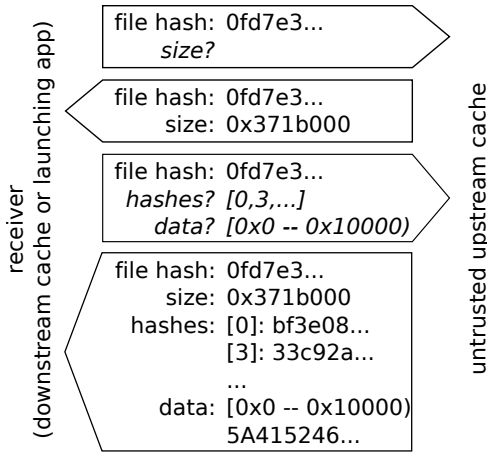

Figure 5: **Transmission protocol.** *The meaning of each reply packet is independent of the query that spawned it, and each reply contains enough information to verify its contents.*

the file metadata; because of the structure of the file hash, it can immediately verify the reply. The receiver determines the data flow, so in each later round, the receiver may request any subset of the Merkle node hashes by their tree indices, and may request any byte range of the file contents.

A bandwidth-constrained receiver may learn the tree one level at a time to avoid fetching even the Merkle hashes of subtrees it already knows; each layer can be incrementally verified. A receiver desiring to minimize round trips will instead ask for all the required leaf hashes in one step, plus any Merkle siblings required to compute all the required interior nodes. Section 5.1 analyzes the trade-off. Our implementation uses the latter algorithm.

Once the leaf hashes are known, the receiver can fetch actual file data, verifying that the data is sane at block granularity. We use a 4 KiB block size, which has reasonable overheads (§5.2).

Because the protocol is receiver-directed, it adapts to diverse scenarios. The receiver chooses which and how much hash and file data arrive in each reply, and hence adapts to networks with varying MTU. It detects integrity failures immediately for metadata and Merkle hashes, and after receiving each block's worth of file data, so it can quickly reject faulty senders. The receiver selects whether to compress file data, based on whether network bandwidth or receiver CPU is scarcer. When an app receives transmissions from a cache on the same host, we use UDP to keep the boot block tiny; in a cache receiving transmissions across the network we use TCP to tolerate latency and congestion.

3.3 Launch

When a new application starts, it consists of a tiny binary boot block running in an isolated process (§2.2).

The boot block contacts a cache on the local machine and asks for its application image by file hash, requesting the required range of bytes. Since the cache is not trusted, the boot block can locate the cache by broadcast. Thus the boot block is quite simple and only understands a subset of the Missive protocol; it leaves the work of fetching the image to the cache, though it may provide a URL or other hint as to where the image may be found.

Launch is optimized to minimize latency, which comes from *data transfer*, *verification time*, and *application start overheads*.

3.3.1 Data Transfer Latency

In a conventional buffer cache, data transfer is very fast: the application names a few dozen files to map, and the OS page-remaps those files from the buffer cache to the process' address space. In the context of Embassies, Missive's untrusted cache achieves a similar effect by transmitting the entire image in a single IPv6 jumbo frame. The Embassies client kernel implements large local transmissions with page remapping, while preserving IP's predictable by-value semantics.

The untrusted cache can only exploit this performance boost if it can prepare the image for transmission efficiently (§3.1.2).

3.3.2 Verification Latency

Since the buffer cache is untrusted, the boot block must verify the image the cache provides before it starts executing the image. This verification is on the critical path. The boot block's contents are public, so it initially has no secret key with which to perform verification. Thus, the boot block includes a hash value, e.g., from SHA-256, to verify the integrity of the image it receives from the cache.

Unfortunately, secure hashes are costly. To reduce the cost of verification in the common case, the boot block substitutes the computation of a hash with a Message Authentication Code (MAC), a faster message summary that requires the sender and receiver share a secret. Initially, the boot block does not have such a secret, so on its first execution, it verifies the image based on its hash. Once the hash verifies, the boot block computes a MAC over the same data, using K_{app}, the app-specific secret key provided by the client kernel (§2.2). The boot block stores the MAC in untrusted storage (e.g., with the cache), since the use of a secret key makes the MAC neither private nor subject to integrity attack by the untrusted store. Standard techniques (not implemented here) can prevent attacks on data freshness [25, 31].

On subsequent starts, the boot block queries the cache for the MAC it previously computed. If the MAC is present, the boot block verifies the image's integrity with the MAC, skipping the hash altogether, resulting in a faster startup. We use VMAC [23], a MAC ten times faster than SHA-1. On our experimental hardware (§5), SHA-1 costs 3.9 ms/MiB ($\sigma = 0.1\%$), whereas VMAC is 0.29 ms/MiB ($\sigma = 1.0\%$).

We aim to reduce user-perceived startup latency, and both hash and MAC computations are embarrassingly parallelizable. Thus our boot block exploits all available cores to trade a wide burst of computation for reduced latency.

Another way to reduce the verification latency is to overlap it with later steps: the host could provide additional primitives to allow speculative execution [29] while the verification process continues (not implemented here). Embassies discards this option because it adds additional host complexity.

3.3.3 App Start Latency

Once the image has been transferred and verified, the app begins executing. Several factors affect how quickly the app is ready for user interaction.

Page Alignment. In our implementation, the bulk of the image consists of shared libraries. Libraries expect to load at 4 KiB memory-page boundaries. Missive's block alignment policy ensures page alignment for large files. When a boot block requests the zarfile from the cache, it specifies a padding header that compensates for the IP header and host packet buffer offsets, making the first byte of the zarfile fall on a page boundary. Correcting these alignment issues forestalls runtime memcpy operations that otherwise impair startup latency.

ELF Section Layout. The ELF standard adds additional complications: ELF-format files have a non-trivial mapping between on-disk structure and in-memory structure. Typically, an ELF file has a large text segment, then a smaller initialized data segment that is expected to appear at an offset in memory different than its offset from the text segment in the file. The ELF file also specifies an uninitialized data (bss) memory region with no corresponding data in the file, as well as file regions (such as debugging symbols) that are not mapped into memory.

Missive addresses this complexity as follows: at image capture time, it records how regions of the library file are mapped into memory. The text segment is recorded in the zarfile in an oversized region adequate to hold the final in-memory layout. The data segment is recorded in a separate region. At runtime, the data segment is copied into place at the appropriate offset, and the bss segment is zeroed. This arrangement lets the library run directly from the launched image, eliminating the bulk of memory copy operations. The cost of the empty space in the image is tolerable, as zero-filled regions compress nicely for wide-area transfer.

4 Image Commonality

To deliver on the promise of fast launch of big, independent applications, Missive assumes that most applications actually share a fair amount of common infrastructure. Broadly, we conjecture that most apps will exhibit commonality with at least some other apps, because there will be only a few popular app-building frameworks. Over time, some will fork and others will ebb. This evolution will look less like today's Web client, where standards make it difficult to fork away from HTML and JavaScript, and more like frameworks on servers, where Django and Rails evolve competitively. This intuition does suggest increased variability, but not unbounded schisms.

4.1 Characteristics of Our Data Sets

We evaluate this conjectured application commonality by examining the commonality within three data sets drawn from two application populations.

First, we study a population of 100 interactive desktop applications from Ubuntu Linux 12.04. We selected them using Linux "best-of" application lists [15, 42] representing a wide variety of domains (e.g., web browsing, vector illustration, word processing, video editing, music composition, software development, chemical analysis) built using various languages (C, C++, Java, Mono, Python, Perl, Tcl) and GUI frameworks (Gnome, KDE, Qt, Tk, Swing).

From this population, we first constructed the **POSIX-complete** data set: for each application, we constructed a zarfile containing every file the application might touch while running. We used Ubuntu's package management system to find this list: we queried the package manager for all packages that are dependencies of the application's base package. (Such dependencies are declared manually by Ubuntu's package maintainers.) We then recursively found sub-dependencies, eventually enumerating the entire dependency tree for each application. Each application's zarfile contained the union of all files in all packages in its dependency tree.

The second data set, **POSIX-startup**, is drawn from the same population. However, instead of the complete zarfiles, we measured only the portion that is accessed while the app launches to the point of interactivity, as captured by `strace`. This provides a more accurate picture of the time a user might be expected to wait to use an app after launching it.

The third data set, **Embassies-startup**, consists of eight apps we adapted from the POSIX world to run in Embassies to validate that the minimal client kernel really can support rich apps. These include Midori/Webkit (an HTML renderer), Abiword (a word processor), Gnumeric (a spreadsheet), Gimp (a raster image editor, like Photoshop), Inkscape (a vector image editor, like Illustrator), Marble (an interactive globe, like Google Earth), Gnucash (an accounting app, like Quicken), and Hyperoid (a video game). These are the types of real rich applications we would like to see deployed in the Embassies model. These apps use some common and some distinct components: most use X windows as a rasterizer, for example, but some use the Qt graphical toolkit, while others use Gtk. Porting the apps to Embassies entails packaging them into images that encompass executable libraries and runtime data.

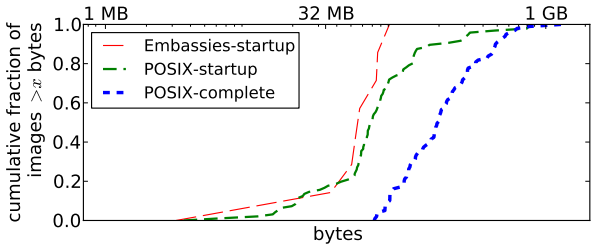

Figure 6: **Zarfile size distribution.** *The distribution of image sizes used to test commonality across images. The -startup sets have medians of 54 and 66 MiB. The complete set images range 68 MiB–1.0 GiB.*

While the Embassies-startup population loses fidelity because it is much smaller than the POSIX apps, it more accurately represents the anticipated ecosystem in that each app image is a real binary that launches in Embassies. The POSIX data sets are less accurate; for example, they omit about 10 MiB of functionality associated with the Embassies POSIX emulation, TCP stack, and X rasterizer.

The agreement between measurements of the POSIX-startup and the Embassies-startup sets suggests that the POSIX-startup set is a reasonable approximation of what the apps would look like if ported to Embassies, and hence we can use this larger set of apps to evaluate such a world. The introduction of the POSIX-complete set lets us reason about the cost of transmitting complete app images for offline use.

Figure 6 shows the distribution of image sizes in each data set.

4.2 Commonality Measurements

To measure commonality, we simulated the transmission of each data set to a client machine assuming that some apps are already cached there. Our hypothesis is that applications share enough common infrastructure (and, thus, common zarfile blocks) that installation of a new application will require significantly less data transfer when other applications are already cached. The block size was 4 KiB.

The results for POSIX-complete are shown in the CDFs in Figure 7a. The bottom curve shows the worst case: transfer of a single zarfile to an empty cache. It is

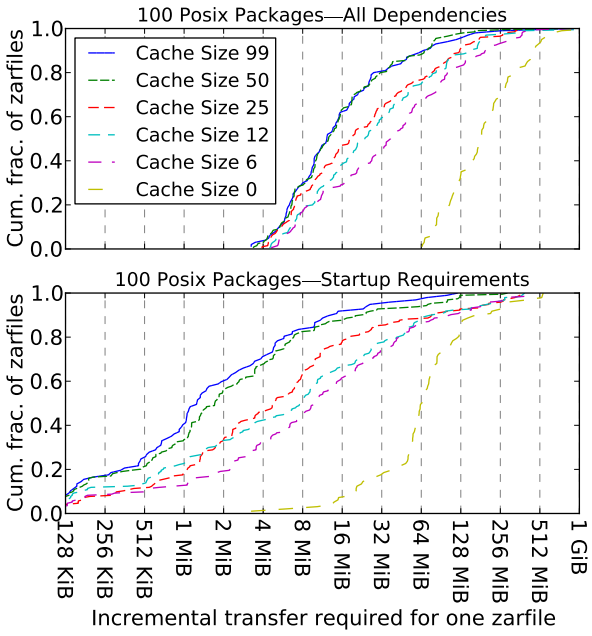

Figure 7: **Content commonality across zarfiles generated for Posix applications.** *Cumulative distribution function of the required transfer size to install the $n+1$st zarfile on a Missive system where n zarfiles are already cached. The block size is 4 KiB. 7a (top) shows complete applications with all dependencies. With 100 apps, the cache reduces the median transfer size from 181.6 MiB to 12.2 MiB. Little incremental benefit is gained beyond 50 cached zarfiles. 7b (bottom) shows results for the portions of the zarfiles required for application start. The cache reduces median transfer size by nearly 98%; typical apps in the startup set exhibit more commonality than the complete zarfiles.*

nearly equivalent to the zarfile size distribution (modulo duplicated blocks), ranging from 64.8 MiB to 1.3 GiB (median 181.6 MiB). The top curve shows the best case: transfer of a single zarfile when all other 99 zarfiles are cached. The number of unique bytes requiring transfer was reduced to an average of only 6.7% of the original, to a median of 12.2 MiB.

The middle curves in Figure 7a show intermediate cases when some (n = 6, 12, 25, and 50) of the 100 zarfiles are cached before the transfer of one additional zarfile. In each simulated transfer, we first populated the cache with n zarfiles selected uniformly at random from the $\binom{100}{n}$ possibilities, then selected one to transfer randomly from among the $100-n$ that remained. The simulation suggests virtually all of the cache's benefit is realized with 50 zarfiles cached.

Figure 7b shows the same experiment performed on the POSIX-startup data set. These zarfile subsets were about 36% the size of the full zarfiles, ranging in size from 3.2 MiB to 543.1 MiB (median 65.6 MiB). But the reduced set isn't just smaller: it also exhibits far

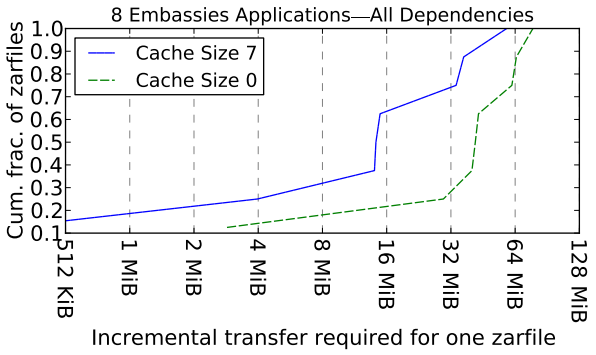

Figure 8: **Content commonality across zarfiles generated for Embassies applications.** *Cumulative distribution function of the required transfer size to install the $n+1$st zarfile on a Missive system where n zarfiles are already cached.*

more commonality between applications. When all but one were cached, retrieval of the final zarfile required a transfer of between 8.0 KiB and 121.3 MiB (median 1.3 MiB)—a reduction to just 2% of the median transfer with a cold cache. To give these numbers context, a typical app that had not been previously installed could start in less than two seconds on a 6 Mbps cable connection.

Figure 8 shows the same analysis for the 8 apps in the Embassies-startup data set. With seven apps cached, transfer time for installation of the eighth was reduced to about 34% of its size. This is roughly comparable to the efficacy of caching for Posix apps with a cache that size. This suggests that in a larger Embassies ecosystem, efficacy of caching will approach the 98% seen in our Posix study.

This simulation does have inaccuracies. It overestimates the available commonality in a real Missive ecosystem because all the applications we tested are from the Posix world, none from Windows or Mac. In addition, where two of our apps share a framework, they use the same version. However, it also underestimates the opportunity for efficient transfer: in an ecosystem that exploits Missive, libraries and frameworks may be repackaged to make their components easier to share; here, no such optimization has been performed. The experiment also understates the benefits of apps that share nearly identical stacks, such as multiple apps built on an HTML renderer.

4.3 Block Size Selection

Block size is an important parameter in Missive. Larger blocks produce less metadata, while smaller blocks might expose more commonality across zarfiles, depending on the distribution of the variations. Figure 9 shows our analysis of Missive's sensitivity to block size for the Posix-complete dataset. We filled the cache with 99 of the zarfiles and simulated the transfer time required

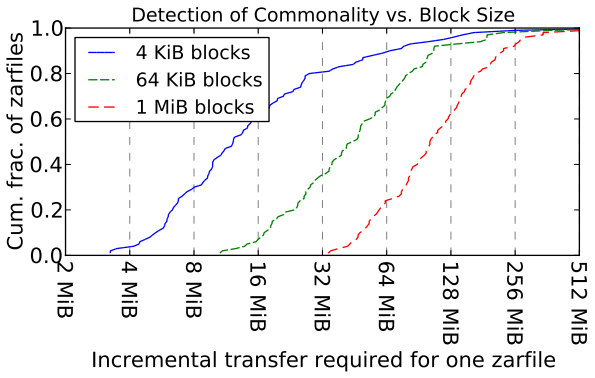

Figure 9: **Sensitivity of commonality-discovery to block size.** *Cumulative distribution function of the required transfer size to install the 100th zarfile on a Missive system where the other 99 are already cached. The size of the blocks hashed is varied. Computing hashes for smaller blocks results in significantly more discovery of commonality across zarfiles and much smaller transfers. Simulation uses the POSIX-complete dataset.*

to install the 100th. With a block size of 1 MiB, that final zarfile required a transfer of between 34.0 MiB and 1.1 GiB (median 102.5 MiB). 64 KiB blocks reduced the median transfer size to 45.3 MiB and 4 KiB blocks reduced it further to 12.2 MiB. The increase in metadata is clearly worth reducing the block size to 4 KiB.

4.4 App Patching

One important special case of commonality is patching: replacing an image already present at the client with a similar one. Note that patching is a domain-specific application of compression, and hence is amenable to specialized optimization. Chrome's Courgette tool produces binary patches ten times smaller than a structure-oblivious binary diff [1].

Missive is designed to extract commonality implicitly across apps, without the receiver identifying a source version, but such specialized tools layer nicely on top of Missive: If an app wants to patch itself, it can identify a previous version against which the patch should be applied, and which is available locally. In that case, a specialized tool like Courgette can be executed either by the app or by the untrusted cache to generate new content from an efficient patch and supply it to the shared cache, ready for future app launches.

5 Evaluation

We analyze the choice of Merkle degree, evaluate the overheads due to Missive's packaging strategy, and measure the overall performance on app startup time. All measurements were collected on an HP z420 workstation with a four core 3.6GHz Intel Xeon E5-1620 CPU and 4GB of RAM.

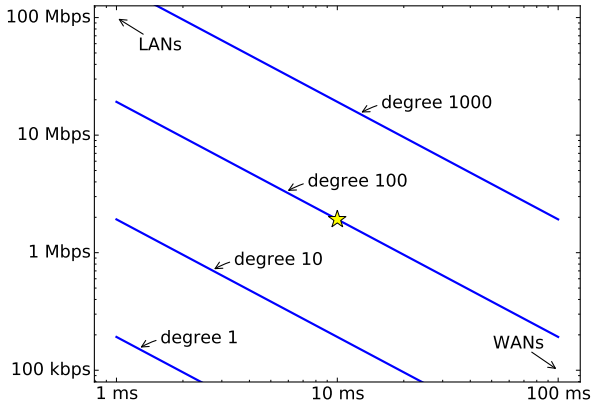

Figure 10: **Ideal Merkle tree degree is a function of bandwidth-delay product.** *On networks with high bandwidth-delay, the receiver might as well request many intermediate hashes rather than spend an RTT hoping to reuse the hashes of an existing subtree.*

5.1 Selection of Merkle Tree Degree

Merkle trees enable a receiver to verify and begin using a partial image before the entire image is transferred and verified [3]. A smaller benefit is that the receiver can skip gathering a subtree of Merkle hashes if the root of that subtree is already cached.

A small-degree tree exposes more such opportunities, but those bandwidth savings come at the cost of round trips. Thus the ideal tree degree is determined by the bandwidth-delay product [21] of the network transport (Figure 10). For example, on a 1.92 Mbps, 10 ms connection (star), receiving 100 192-bit hash records is as cheap as an RTT, thus the optimal tree degree is 100.

We configured our Merkle tree with degree two because it was easy. This value only makes sense for slow networks, but receivers on faster networks can emulate higher degree by requesting multiple layers in each RTT.

5.2 Overheads

The image file format is structured to create opportunities for commonality exploitation, but that structure can introduce overheads. The most important overhead is the alignment-inducing padding, but in practice, our measurements show that it remains below a few percent (Figure 11).

The padding overhead is affected by the choice of block size (Figure 12): larger blocks incur more padding, but increasing the block size also marks more files as "small", packing them into the tail of partially-used blocks. It turns out padding is worst at 64 KiB: smaller blocks generate less padding per "large" file, and larger blocks reduce the number of "large" files that generate padding. Regardless of block size, the padding never becomes a significant overhead.

Packing small files into block tails hides opportunities

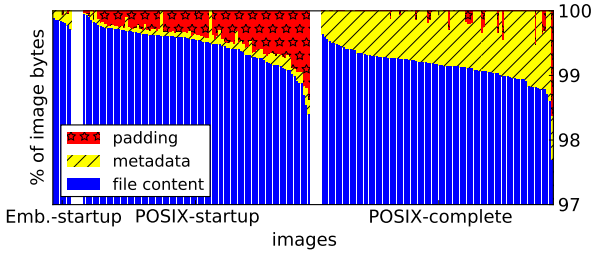

Figure 11: **Image file format overheads run 1-2%.** *The image file format incurs about a percent overhead for metadata such as file stat metadata and a name index. Padding varies depending on the distribution of file lengths, but it remains below a few percent. Note the y axis begins at 97%. Images constructed with 4 KiB blocks.*

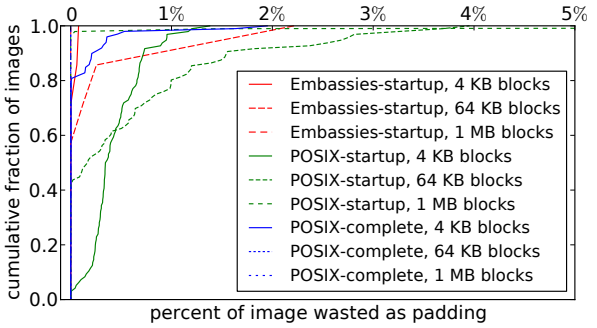

Figure 12: **Padding overhead** *is insensitive to block size.*

for sharing; is it necessary? Yes: although there are few bytes in small files (Figure 3), giving each file its own block introduces considerable overhead that grows with block size (Figure 13).

5.3 The Bottom Line: Startup Latency

Ultimately, we aim to demonstrate that, by judiciously coupling image transmission and buffer cache, Missive achieves interactive performance without a trusted file system or buffer cache infrastructure. To that end, we measure end-to-end network transmission and application launch times.

5.3.1 Launch Time

Besides fast transmission, Missive should add minimal additional time moving an app from the trusted cache into an executable condition in a new process; this is its buffer-cache-like function.

Figure 14 shows time to launch Gimp and several Midori-based apps, for which we have good internal probe points and can measure startup time all the way to the point of user interactivity. It contrasts launching inside Embassies with starting the same content on Linux. The "hot" start represents the primary function of a buffer cache: bringing a machine-resident app into memory for prompt execution. Missive's hot start is about 100 ms or less overhead.

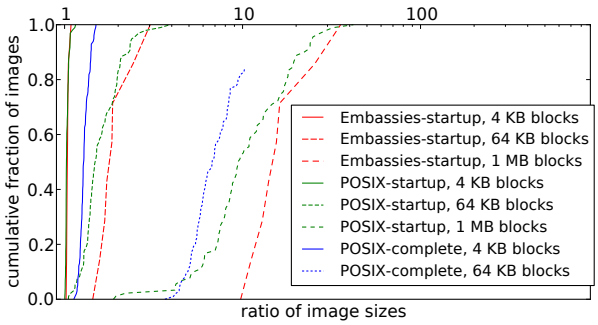

Figure 13: **Small-file packing is necessary.** *Giving small files their own blocks maximizes sharing opportunity, but the file distribution includes so many small files that doing so generates significantly more padding than the actual file content. The smallest block sizes have median bloat of 2–28%.*

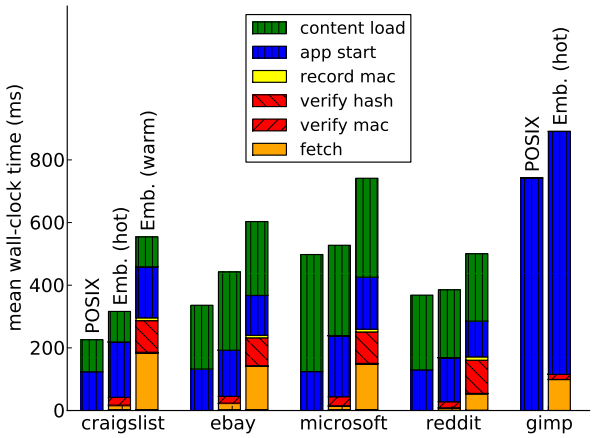

Figure 14: **Missive's launch is comparable to POSIX buffer cache.** *In the hot case, the app's blocks are in the untrusted cache; the primary overhead is the MAC verification (§3.3.2). In the warm case, missing app blocks are fetched from a fast server; the primary overheads are the slower hash and the memcpys required to assemble the image (§3.1.2).*

A 100 ms delay may seem expensive [26], but three factors mitigate it. First, it only affects the launch of a new binary; navigation among pages or activities within a site's application are unaffected. Second, once a site's binary is running, the vendor controls both ends; it is free to deploy SPDY [30] or the next innovation anytime. Third, we have only performed black-box optimizations; application-specific tuning could dramatically reduce the amount of data needed for startup to the first point of interactivity.

In the "warm" start case, the cache contains a copy of the Midori browser with a vulnerable version of libpng. We measure the time to start an app using a similar stack (exploiting that local commonality) with a patched libpng. Since the network overhead is a function of content size (studied in Figures 7 and 8), this experiment uses a local network connection to focus attention on the

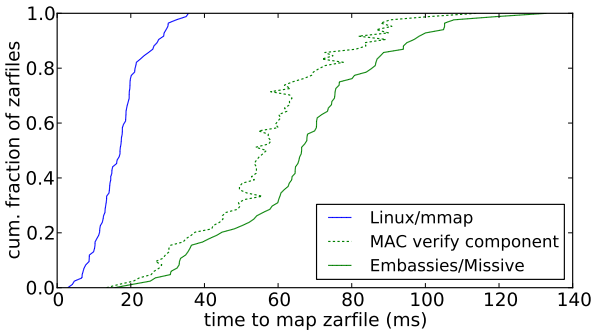

Figure 15: **Contrasting just launch time** *across the broader Posix-initial data set, the median Missive app takes about 50 ms longer, most of which is verification time. (Verification times are sorted with the corresponding app, not cumulatively.)*

system's inherent sources of latency. In the warm start case, Missive's cache requires about 150 ms to assemble the outgoing zarfile from cached blocks, and the receiver pays as much again to verify the zarfile with a hash.

Figure 15 measures the broader Posix-startup data set, but more shallowly, in that it only captures the cost of mapping the executables into memory. For this hot start experiment, we had a Linux process `mmap` every file in the data set and `read` the rest; we contrasted that to Missive's launch step, which pulls one zarfile into its memory. In both cases, the test apparatus touched every memory page. The median Linux time is 17 ms. Most of Missive's 66 ms is the cost of integrity verification. Although the cost is $4\times$ higher in relative terms, the overall burden is not overwhelming compared to the overall start time of typical applications, which this experiment excludes.

While Missive is not completely without cost, in the hot case where it can use VMAC (§3.3.2), it hovers very close to the performance of a native, trusted buffer cache.

6 Related Work

Numerous systems have proposed content-addressable techniques for identifying and routing content, as well as reducing bandwidth [5, 7, 11, 14, 28, 39]. We briefly touch on some of the most related efforts below.

Tolia et al. proposed a content-oriented data transfer service with the aim of decoupling application-level content negotiation from data transfer, hence enabling greater innovation in transfer protocols [38]. Like Missive, they divide objects into chunks identified by hash, allowing caches to identify shared content across applications. The receiver drives the data transfers by specifying chunks of interest. Since Tolia et al. focus on issues related to data transfer, they do not consider, as Missive does, how to identify and package app-related files, they assume the local content cache is trusted, and they do not address the final step of rapidly transferring the app image into a booting process and verifying it.

Van Jacobson et al. and Trossen et al. have proposed building efficient commonality-exploiting data transfer by addressing content at the network layer [19, 40]. Rhea et al. use content hashes to reduce the bandwidth needed for Web traffic [34]. Spring and Wetherall also use hash-based matching to perform data deduplication at the IP layer [35].

Tangwongsan et al. propose a multi-resolution handprint for selecting a content chunk size that optimally exploits data redundancy in files [37].

Multiple projects (e.g., the Collective [6] and Internet Suspend/Resume [22]) proposed distributing applications as full VMs, both to simplify management and to minimize cross-application conflicts. On the server side, SnowFlock [24] proposed a data-center-wide VM fork operation for instantiating a single VM on hundreds of machines. To minimize launch latency, they developed several techniques for lazily replicating only the active working set of the VM being forked. Unlike Missive, these projects assume a trusted local cache.

The deduplicating transport of Missive is constructed of fairly conventional techniques. We considered using BitTorrent [8] or other deduplicating transports [13, 28]. Using a custom transport protocol, however, admits three advantages: its use of Merkle trees enables immediate use of partially-loaded images, it is optimized around the untrusted cache's extreme latency constraints, and its simplicity enables the same protocol to function well in the wide-area and be implemented in a tiny boot block.

The most crucial abstract idea in Missive is the separation of sharing content for performance from sharing by reference; this distinction was significantly inspired by the Slinky system [9].

7 Conclusion

By virtue of the protocol that boot blocks use to share it, Missive's untrusted cache supplants functions normally supplied by a host's (very trusted) buffer cache. It coordinates memory as a cache for disk, and disk as a cache for the network. It rapidly assembles an application's image from parts both unique and common with other apps. It exposes sharing in a way that the underlying memory page manager can exploit for performance, while preserving for apps the abstraction of private copies; but because naming is by content, the isolation is much stronger than in the conventional use of a buffer cache.

References

[1] ADAMS, S. Smaller is faster (and safer too). `http://blog.chromium.org/2009/07/smaller-is-faster-and-safer-too.html`, July 2009.

[2] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R.,

AND LORCH, J. R. A five-year study of file-system metadata. *Trans. Storage 3*, 3 (Oct. 2007).

[3] BAKKER, A. Merkle hash torrent extension. http://www.bittorrent.org/beps/bep_0030.html, Mar. 2009.

[4] BAUMANN, A., LEE, D., FONSECA, P., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing OS extensions safely and efficiently with Bascule. In *EuroSys (to appear)* (2013).

[5] BRESSOUD, T. C., KOZUCH, M., HELFRICH, C., AND SATYANARAYANAN, M. OpenCAS: A flexible architecture for content addressable storage. In *Workshop on Scalable File Systems and Storage Technologies* (Sept. 2004).

[6] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C. P., AND LAM, M. S. The Collective: A cache-based system management architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2005).

[7] COHEN, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[8] COHEN, B. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems* (2003), vol. 6, pp. 68–72.

[9] COLLBERG, C., HARTMAN, J. H., BABU, S., AND UDUPA, S. K. Slinky: static linking reloaded. In *USENIX ATC* (2005).

[10] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[11] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2001).

[12] DOUCEUR, J. R., AND BOLOSKY, W. J. A large-scale study of file-system contents. In *SIGMETRICS* (1999).

[13] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. The Utility Coprocessor: Massively parallel computation from the coffee shop. In *USENIX ATC* (2010).

[14] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peerto-peer storage utility. In *Proceedings of the HotOS Workshop* (May 2001).

[15] FARSHAD. Best 60 linux applications for year 2011. http://www.addictivetips.com/ubuntu-linux-tips/best-60-linux-applications-for-year-2011-editors-pick/, Jan. 2012.

[16] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. In *OSDI* (2000).

[17] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008).

[18] HOWELL, J., PARNO, B., AND DOUCEUR, J. Embassies: Radically refactoring the web. In *NSDI (to appear)* (2013).

[19] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking named content. In *5th international conference on Emerging networking experiments and technologies (CoNEXT)* (2009).

[20] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[21] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. In *SIGCOMM* (2002).

[22] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications* (June 2002).

[23] KROVETZ, T., AND DAI, W. VMAC: Message authentication code using universal hashing. Internet Draft: http://http://fastcrypto.org/vmac/draft-krovetz-vmac-01.txt, Apr. 2007.

[24] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: Rapid virtual machine cloning for cloud computing. In *EuroSys* (Apr. 2013).

[25] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small trusted hardware for large distributed systems. In *NSDI* (2009).

[26] LINDEN, G. Make data useful, 2006. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf.

[27] MERKLE, R. C. A certified digital signature. In *CRYPTO* (1989), pp. 218–238.

[28] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP* (2001).

[29] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP* (2005).

[30] PADHYE, J., AND NIELSEN, H. F. A comparison of SPDY and HTTP performance. Tech. Rep. MSR-TR-2012-102, Microsoft Research, July 2012.

[31] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Symposium on Security and Privacy* (2011).

[32] PARNO, B., MCCUNE, J. M., WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. CLAMP: Practical prevention of large-scale data leaks. In *Symposium on Security and Privacy* (2009).

[33] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *ASPLOS* (2011).

[34] RHEA, S. C., LIANG, K., AND BREWER., E. Value-based web caching. In *Proceedings of the World Wide Web Conference* (May 2003).

[35] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM* (Sept. 2000).

[36] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. File size distribution on UNIX systems: then and now. *SIGOPS Oper. Syst. Rev. 40*, 1 (Jan. 2006), 100–104.

[37] TANGWONGSAN, K., PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Efficient similarity estimation for systems exploiting data redundancy. In *Proceedings of IEEE INFOCOM* (Mar. 2010).

[38] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for internet data transfer. In *Proceedings of USENIX NSDI* (May 2006).

[39] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., PERRIG, A., AND BRESSOUD, T. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2003).

[40] TROSSEN, D., SARELA, M., AND SOLLINS, K. Arguments for an information-centric internetworking architecture. *SIGCOMM Comput. Commun. Rev. 40*, 2 (Apr. 2010), 26–33.

[41] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *SOSP* (2005).

[42] WHEATLEY, R. Top 100 of the best (useful) opensource applications. http://www.ubuntulinuxhelp.com/top-100-of-the-best-useful-opensource-applications/, Feb. 2008.

[43] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A sandbox for portable, untrusted x86 native code. In *Symposium on Security & Privacy* (2009).