Building and Evaluating a k-Resilient Mobile Distributed File System Resistant to Device Compromise

Scott Huchton
Department of Computer Science
Naval Postgraduate School
Monterey, California
Email: sthuchto@nps.edu

Geoffrey Xie
Department of Computer Science
Naval Postgraduate School
Monterey, California
Email: xie@nps.edu

Robert Beverly
Department of Computer Science
Naval Postgraduate School
Monterey, California
Email: rbeverly@nps.edu

***Abstract*—Deploying mobile devices to frontline troops presents many potential benefits, e.g. situational awareness, enhanced communication capabilities, etc. However, security remains an impediment to realizing such capability. In this research, we develop and evaluate an approach to securing the non-volatile storage of a collection of mobile devices. Our technique relies on well-established cryptographic primitives, combining them in a unique way to meet military mission specific security and resiliency requirements. Specifically, we create MDFS, a distributed mobile file system using erasure coding, Shamir's threshold secret sharing, and the symmetric AES block cipher. The resulting system provides two important properties: (1) data at rest is protected even after total compromise of up to k devices, and (2) data is replicated within an infrastuctureless ad hoc network and, as such, resilient to device outages. We implement MDFS on Android mobile devices and achieve ≃10Mbps throughput in real-world performance experiments, suggesting that MDFS is suitable for a variety of practical workloads.**

I. Introduction

The proliferation of mobile "smartphones" is receiving considerable attention within the military community [1], [2]. Providing frontline troops with such mobile devices promises to enhance situational awareness, provide enhanced communication and computing abilities, etc. A collection of mobile devices could form a mobile ad-hoc network (MANET) capable of distributing tactically relevant information to and within the battlefield.

However, communicating and storing sensitive information comes at a potential cost to mission security. For example, if a device is captured by an adversary, the mission might be compromised due to either information leakage or unavailability of the device. Common approaches to data sharing on mobile devices, e.g. [3], tend to focus on routing and transmission security. Transmission security addresses one important aspect of the overall threat model, but offers no solution to data resiliency or data security when a device is captured. Other approaches, e.g. tamper resistance [4], require specialized manufacturing and increase per-device cost beyond levels realistic for widespread deployment.

To address the security of data at rest in military mobile phone MANETs, we develop and evaluate a prototype k-resilient storage system: the Mobile Distributed File System (MDFS). As opposed to authenticating devices or securing the communication channel, we focus on securing content in the non-volatile storage of the mobile devices. Thus, our system is complementary to existing authentication, privacy, and integrity techniques.

More specifically, MDFS ensures data privacy through a *group secret sharing scheme* instead of relying on conventional independent encryption keys per device or per user. As a result, the system is resistant to total device compromise as long as fewer than k devices are captured or lost, where k is a customizable parameter. We call such a storage system k-resilient in withstanding device captures and note that when $k > 1$ the system provides a *stronger security guarantee* than simply encrypting stored data per-device.

We imagine MDFS useful in many practical military scenarios. Suppose a team of soldiers are on a mission where access to, and sharing of, data is critical to mission success. The soldiers require devices capable of transmitting and storing sensitive data, but the loss of one or more of those devices could prove devastating if the enemy is able to gain access to the sensitive data. Encryption on the mobile device only partially solves this problem and does not address the issue that the data stored on the lost device is no longer available to the rest of the team. More importantly, if the encryption key were coerced or otherwise recovered, all data residing on the compromised device is revealed. Building a "remote kill" feature into the mobile devices can mitigate the problem, but such a solution works only when the captured devices remain connected to the network.

MDFS combines several well-established concepts, including Shamir's threshold based secret sharing scheme [5], erasure coding [6], and AES encryption [7] to create a functional design for secure and resilient data sharing among a collection of mobile devices. For security, MDFS fragments content and securely stores fragments across multiple devices such that content can be reconstituted only when greater than a prespecified number of distinct fragments are accessible contemporaneously. For resilience, MDFS employs information redundancy such that different collections of devices may feasibly collaborate to recover the content. An added benefit of MDFS is that each file is encrypted with a one-time dynamically generated encryption key; as such, MDFS requires no pre-sharing of encryption keys and thus avoids security issues associated with key distribution and management.

One major practical concern for deploying MDFS is that the combined processing requirement of its various components may exceed the capabilities of mobile phone hardware. Even in the future, data size requirements will scale in proportion to the computational abilities of mobile devices. To understand the practicality of MDFS, we describe and perform a series of experiments with a prototype implementation of MDFS in Java and C on a common 1GHz smartphone, an Android 2.2 [8] device.

This paper describes the MDFS design in detail and presents experimental results that demonstrate the feasibility of implementing MDFS on current Android hardware. Our primary contributions include:

1) The design and implementation of MDFS on current state-of-the-art mobile hardware.
2) Quantification, via real-world experiments, of MDFS performance.

The remainder of this paper discusses related work (§II), the design of MDFS (§III) and performance results (§IV). Finally, we discuss implications of our work on the design of secure mobile file systems and conclude with future work.

II. Related Work

Substantial work in disk forensics has shown the vast amount of sensitive information available to an adversary with physical access to a device that has not been properly secured. Specific to mobile phones, Distefano et al. [9] demonstrate the extent to which Android is vulnerable to forensic techniques while commercial entities now provide such forensic capabilities for smartphones [10]. Thus, protecting non-volatile storage is an essential component of mobile device security.

Conventional approaches to securing data at rest rely on symmetric encryption [11] of files, the file system, or the physical device. Modern operating systems include the ability to encrypt non-volatile data, while a wide variety of third-party software provides additional enhancements, including plausible deniability [12]. Storage device manufacturers now provide hardware-based encryption schemes. The primary weakness of these schemes lies in how the encryption keys are stored. Trusted Platform Modules can tie storage devices to hardware, but none of the existing techniques prevent key recovery by coercion.

Specialized, security hardened mobile phones exist for military applications [13], [14], however these devices are expensive to deploy and maintain. Perhaps more importantly, they are designed to secure the device, not a mission that involves multiple communicating parties. Thus, our focus on ensuring resilience, e.g. data availability, in the face of device capture, is complementary to these hardware approaches.

The concept of physically separating file fragments to provide security has been explored before. Unisys Systems provides this ability for enterprise networks with a product called Stealth Technology [15]. Stealth uses a subsystem called SecureParser that is capable of generating encryption key fragments in much the same way that MDFS does. Stealth uses distributed storage and encryption to allow an organization to simulate multi-level security access based on workgroup affiliations. Stealth uses the Multi-Level Security Tunneling Protocol (MLSTP) at a gateway to break data into fragments and then encrypts those fragments using a session key before storing the fragments in cloud storage. Stealth is similar to MDFS in that access to the shares acts as the authority to view the data. However, MDFS is implemented on relatively constrained resources and does not rely on gateway servers or tunneling protocols to achieve security.

MDFS shares design aspects with the open source Tahoe-LAFS project [16]. Tahoe-LAFS stores file fragments across many physical drives to provide "provider independent security." However, to the best of our knowledge, the use of fragmented storage on mobile devices has not been explored. MDFS's ability to operate without infrastructure in a wireless environment differentiates it from existing solutions.

A. Erasure Coding and Shamir's Algorithm

Erasure codes are forward error correction (FEC) codes that translate some message M of length $|M|$ into a coded message with a length greater than $|M|$ such that M can be recovered from some subset of the coded message. In 1960, Reed and Solomon introduced a Maximum Distance Separable (MDS) algorithm [17]. An MDS erasure code stores a message M in n fragments of size $|M|/k$ such that any $k < n$ of the fragments is sufficient to reconstruct M.

Shamir's Secret Sharing Algorithm [5] is closely related to Reed-Solomon and other erasure codes [6]. First proposed in 1979, the algorithm computes from a secret S a set of partial secrets and distributes the partial secrets to multiple entities so that the recovery of S requires some minimum number of these entities to cooperate and share their partial secrets. Unlike a regular erasure code, Shamir's algorithm guarantees that an adversary learns *nothing* about S as long as the number of exposed partial secrets does not reach the minimum threshold.

III. Design

The current MDFS design targets primarily the risk of data leakage resulting from device loss or capture. As discussed earlier, several complementary features may be added to the design to further enhance its security and scalability. For brevity, we omit a detailed exploration of these features. Instead, we focus our discussion with these simplifying assumptions:

- All mobile devices support link-level encryption (e.g., with Type 1 hardware) so that no additional message authentication is required for MDFS.

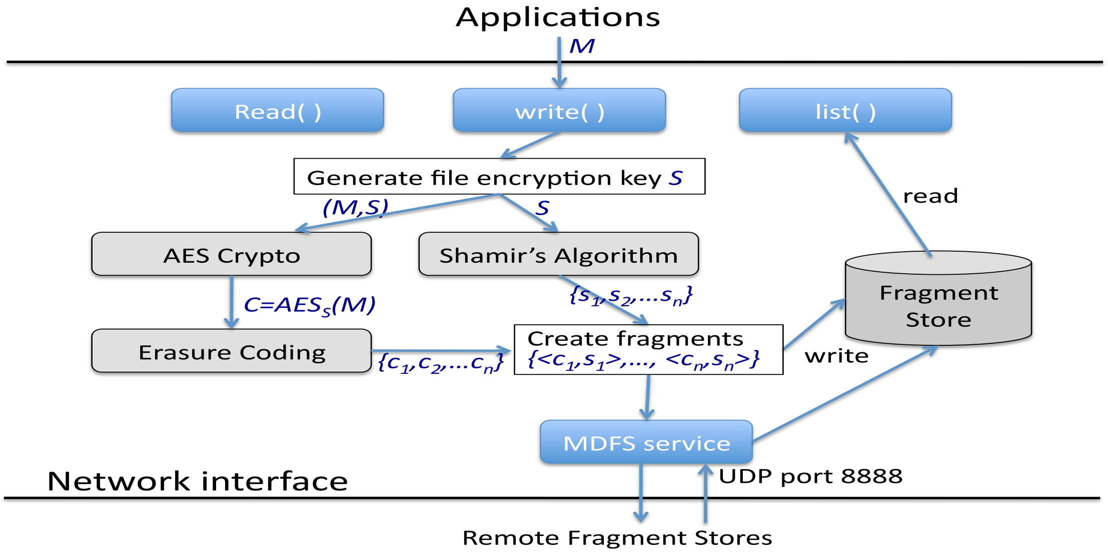


Fig. 1. MDFS Functional Overview: the write(M) API function generates a secret key S that is used to encrypt M. Forward error correct breaks M into chunks which are added to shares of secret S to create an MDFS fragment.

- MDFS is to be deployed to small front line troop units (e.g., a company). A typical deployment consists of fewer than one hundred mobile devices.

We next formulate the main security and resiliency requirements under the MDFS design framework. We then detail our design of the write() and read() API functions meeting these requirements as well as the design of a supporting, but essential, distributed directory service.

A. Requirements

We first define our terminology. Let M be arbitrary data (e.g., a message, GPS reading, or photographic image file) stored in MDFS. Let C be the ciphertext version of M, encrypted using a block cipher with key S. Finally, let F be an MDFS fragment, a container object we describe next.

MDFS aims to protect M by encrypting and distributing M across multiple (say n) mobile devices. MDFS's objective is to protect M from an adversary able to compromise some fraction of the n total devices. In particular, MDFS is designed to be k-resilient in its data protection, which can be formally defined by the following two criteria:

- **Security:** An adversary learns nothing about M even after obtaining up to $k < n$ distinct fragments $\vec{F}$ of M, i.e., $Pr(M|\vec{F}) = Pr(M)$ for any set $\vec{F}$ of no more than k distinct fragments.
- **Resiliency:** A legitimate user can reconstruct M after obtaining any $k+1$ distinct fragments of M. That is, a legitimate user can access M even after up to $n-k$ of the fragments are inaccessible.

For example, n and k may be set to 7 and 3, respectively. In this case, MDFS needs to transform a file into 7 distinct fragments, distribute them to 7 different devices, and guarantee that the file cannot be leaked as long as no more than 3 devices are compromised. Clearly, stronger security is ensured with a larger k value. The trade-off is that n needs to be increased accordingly to maintain sufficient data redundancy for legitimate users.

B. Overview

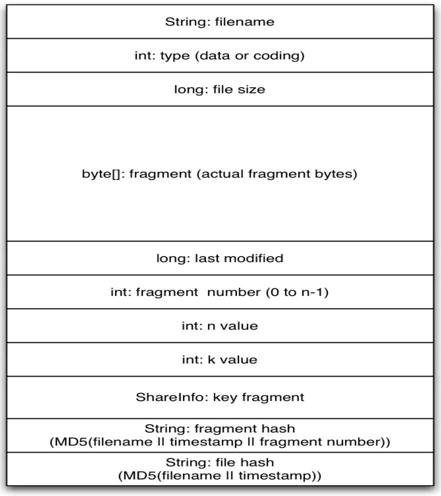
Before detailing MDFS, we provide an overview of its operation. MDFS combines several well-established cryptographic primitives. For privacy, MDFS uses the AES block cipher for encryption. To protect the AES symmetric key, MDFS employs Shamir's secret sharing algorithm. Finally, to add redundancy, thereby potentially increasing availability in a MANET, MDFS performs forward error correction in the form of Erasure coding. Note that the notion of k-resilient translates to a threshold of $k+1$ in both Shamir's algorithm and erasure codes.

Figure 1 provides the high-level intuition of MDFS operation. Within each node is an MDFS *agent* that provides a file system abstraction to higher-layers. An MDFS agent is a piece of functionality that coordinates with other MDFS agents that are available via the local MANET at any given time.

MDFS provides an API to applications that request to store some plaintext content M. Such a request invokes the following:

- MDFS dynamically generates a symmetric key S that is used to create ciphertext $C = AES_s(M)$.
- The key S is not persisted to storage; instead S is broken into shares $\{s_1, s_2, \ldots s_n\}$ using threshold secret sharing.
- Forward error correction creates fragments $\{c_1, c_2, \ldots, c_n\}$ of C.
- MDFS combines each $< c_i, s_i >$ into *fragment containers* that are distributed among devices.
- MDFS registers the location and identifiers of fragments within a distributed fragment store.

These actions are detailed next; §V removes some of our simplifying assumptions.



String: filename

int: type (data or coding)

long: file size

byte[]: fragment (actual fragment bytes)

long: last modified

int: fragment number (0 to n-1)

int: n value

int: k value

ShareInfo: key fragment

String: fragment hash
(MD5(filename II timestamp II fragment number))

String: file hash
(MD5(filename II timestamp))

Fig. 2. MDFS Fragment Container

C. `write()` Function

The design of the MDFS `write()` function is illustrated in Figure 1. To meet the security requirement, MDFS uses AES to encrypt each file (M) before breaking it into fragments, $C = AES_S(M)$, where S is an automatically generated symmetric secret key.

To meet the resiliency requirement, MDFS uses an erasure encoder (e.g., Reed-Solomon [17]) to break the encrypted file (denoted by C) into n distinct fragments, $c_1, c_2, ..., c_n$, while allowing the reconstruction of C from any $k + 1$ of the fragments. Meanwhile, MDFS uses Shamir's algorithm [5] to break S into n pieces, $s_1, s_2, ..., s_n$, as well, while guaranteeing that S can be recovered if and only if more than k distinct pieces are obtained.

At the final step of `write()`, MDFS combines the C and S fragments into n new fragments, $< c_1, s_1 >, < c_2, s_2 >, ..., < c_n, s_n >$, and writes each of them into the Fragment Store of a different device. Note that a separate module called "MDFS Service" – a daemon in the Unix terminology – continuously runs in the background on a designated UDP port of each device to handle all inter-device MDFS communication tasks including neighbor discovery and the write of fragments into remote devices.

Each fragment is packaged in a fragment container as illustrated in Figure 2. The container stores the fragment bits, the metadata required by the erasure decoder and the Shamir coding library, a unique MD5 fragment hash based on the filename, creation timestamp, and the fragment number to identify individual fragments within a file, and finally, another MD5 file hash based on just the filename and creation timestamp. The two hash values provide a method to identify distinct fragments from the same file. Specifically, fragments with the same file hash value, but a different fragment hash value are unique fragments of the same file.

We use the file's creation timestamp in the hash rather than some other metadata for two reasons. It is straightforward to obtain a timestamp at the creation of a file and the likelihood of two different files having the same filename and the exact same timestamp is remote. Thus, this is an easy way to differentiate between two files with the same filename. Furthermore, it allows for future development where files could be updated and versioned based on their timestamp. In other words, if a file is updated, it would have a more recent timestamp than the original. If the updated file's metadata contains a reference to the original file hash, then file versioning over time could be maintained.

D. `read()` Function

The step-by-step operations of the MDFS read function are mostly inverse of that of the write function. For brevity, we omit the details.

However, there is one question uniquely relevant to the read function. How does a user (and application) know what data is available in MDFS before calling `read()`? Like any computer file system, MDFS provides a directory service for applications, as described in the next subsection.

E. Directory Service

The third major design component of MDFS is a directory service that provides an interface for applications to view what data are available in the file system. To be compatible with the MANET environment, we have adopted a *totally distributed* design for the directory service. The design requires all devices to have the same functionality and share the work load to avoid choke points or single point of failure.

Specifically, each device maintains a local file directory along with its fragment store. The directory mainly stores the following information: (i) a list of current files in MDFS, and (ii) for each current file, the source of the file and a list of nodes that hold fragments for the file. The directory may be used to also track which devices are currently connected to the network. When connected to the network, a device periodically synchronizes its file directory with that of other devices. Whenever MDFS adds a new file fragment to some device's fragment store during a `write()` call, MDFS also registers information about the file and the new fragment with that device's file directory.

A primary part of the MDFS directory service is a `list()` function, which is the API for applications to use the MDFS directory service. The `list()` function should support a full listing of files, query of files by their source or keywords, advanced file search based on regular expressions, and query of status of other devices, among other features.

IV. Experimental Results

To evaluate the performance of MDFS on current mobile devices, particularly smart phones, we have created a prototype implementation of the design on unmodified HTC Evo 4G (Snapdragon 1GHz processor, 512MB DRAM) smart phones running Android 2.2. Because stock Android does not support ad hoc mode networking, our prototype may not be suitable for evaluating MDFS in a true MANET environment with a fully distributed directory service. Therefore, the focus

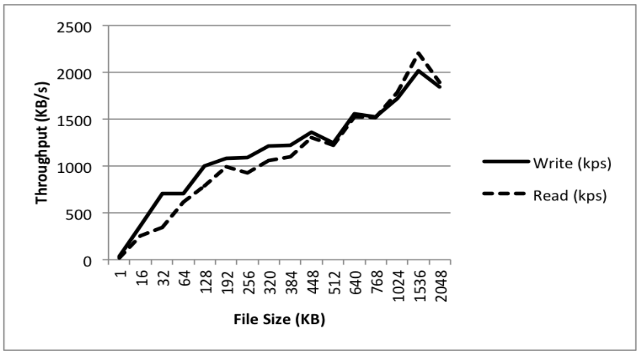


Fig. 3. Throughput (KB/s) of Read and Write for Different File Sizes

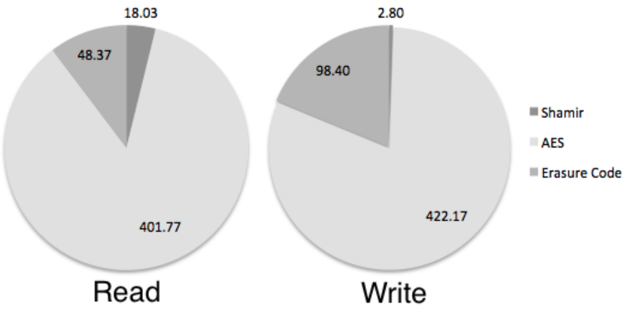


Fig. 4. Average Executive Times (ms) for 1MB file

of our initial performance evaluation is on the processing overhead of the three primary building blocks of MDFS: AES encryption, Shamir's algorithm, and erasure coding. We defer the evaluation of networking related performance issues to future work.

A. Implementation Details

The MDFS Android implementation consists of a total of 3332 lines of original Java code. It utilizes a Java library called SecretShare [18] for the Shamir's algorithm. SecretShare is a Java implementation of the LaGrange Interpolating Polynomial Scheme as described in Applied Cryptography [11]. The SecretShare library consists of 1559 lines of Java code. AES encryption is accomplished using the standard Java AES library that comes with JDK 6. Finally, erasure coding is implemented using a native C library called JErasure 1.2 [19] which constitutes 2161 lines of ANSI C code. We created an API interface to JErasure using Android's Native Development Kit (NDK).

B. Benchmark Results

For the first benchmarking experiment, we measured the throughput of the `write`() and `read`() functions for different files sizes between 1K and 2048K (2MB). The parameters n and k were set to 7 and 3, respectively. Each file was written and read with MDFS 30 times and then the average throughput was computed. Figure 3 shows the average throughput results, in kilobytes per second (KB/s). Observe that both the write and read throughputs increase steadily as the file size increases. It's no surprise that MDFS is less efficient with smaller files due to the initial setup overhead of erasure coding, Shamir's algorithm, and AES cryptography. More importantly, we can conclude that the MDFS prototype has no problem handling file sizes up to 2MB.

The second set of experiments consisted of analyzing the cost breakdown of each of the three primary building blocks of MDFS using a 1024KB (1MB) file. Figure 4 shows the average execution time (in milliseconds) of each component of MDFS over 30 runs. Table I shows the same results as a percentage of total execution time required by MDFS. It shows that AES operations dominate the processing overhead for both read and write. This is good news given that the current prototype uses a Java AES library; we anticipate a significant improvement of MDFS performance if a more efficient AES library (e.g., C based) is used instead.

V. Discussion

Note that several implicit assumptions exist in our design with subtle implications. For example, we make a conscious choice to couple the degree of error correction with the number of secret shares (i.e. the same n is used for both the erasure coding and Shamir's algorithm). Thus, each fragment contains both a share of the key (s_i) and a piece of error corrected content (c_i). Similarly, the thresholds for both recovering the secret and reconstituting the ciphertext are the same.

However, other designs may be appropriate for specific situations. In general, the thresholds and replication factor might be different for security and resilience. A fragment container might contain only fragment content (c_i) and no key shares, for instance, if selected nodes have different expected probability of capture, or there exist other military hierarchical order constraints. While we do not explore such situations here, MDFS is flexible enough to accommodate these different schemes.

In a similar vein, the MDFS API, as described, does not expose either parameter n or k to the application designer. We assume these parameters are pre-configured into each device according to the network environment and mission requirements. However, the design of the fragment container metadata is such that each write can be made with individual parameters appropriate to the content. For example, data might be replicated in proportion to the number of military platoons, or might only require two users within a company in order to reconstitute other content.

Second, our design as detailed does not provide message level integrity check. An adversary might capture a phone and respond maliciously to MDFS queries – thereby compromising the integrity of the content. In a deployed military environment, we anticipate standard techniques, e.g. using PKI along with common access cards (CACs) or biometric identification, to support digital signatures on top of MDFS.

A related issue is about timely detection of device capture. MDFS currently provides little protection against undetected device capture, particularly when the personnel operating the device is also captured. We defer the question of how to detect device capture and the question of how to exclude a suspicious device expeditiously from MDFS to future work.

TABLE I
MDFS PROCESSING OVERHEAD AS A FUNCTION OF API OPERATION

	Shamir	**AES**	**Erasure Coding**
Read	4%	86%	10%
Write	1%	81%	19%

Third, our current implementation assumes the availability of n distinct nodes running MDFS agents during a write operation. To remove this strong assumption, we plan to add appropriate queuing mechanisms to MDFS such that fragments are distributed on demand when required, when new peer MDFS agents become available.

Fourth, we imagine enhancing MDFS with the inclusion of a per-fragment time-to-live (TTL) value (e.g., 5 minutes) that is customizable at the file creation time. MDFS will remove and discard all fragments whose TTL has expired. Whenever MDFS discards a file fragment due to an expired TTL, MDFS also removes information about the file fragment and file from the corresponding file directory. The TTL potentially provides additional protection and scalability in some environments by preventing the fragment store of an MDFS agent from growing indefinitely. For example, a TTL is valuable for some types of information, e.g., GPS coordinates, where the content is valuable to legitimate users only for a small time window, but may be harmful if known by adversaries.

Finally, the security of MDFS as presented lies purely in preventing an adversary from obtaining $k + 1$ fragments for a file, i.e. the security of the system is entirely compromised if $k + 1$ phones are captured. However, as previously noted, MDFS is *complementary* to other security techniques. To mitigate this weakness, one lightweight (as opposed to PKI) solution is to deploy a shared symmetric secret key P and use it to encrypt the content M prior to encryption with the secret sharing key: $C = AES_S(AES_P(M))$. We imagine P to be a mission-specific master secret that has been distributed to all MDFS device operators, where pre-placed keys are feasible in such military operations. With this minor enhancement, an adversary must capture $k+1$ phones and recover P, rendering MDFS strictly stronger than conventional encryption of non-volatile storage.

VI. CONCLUSION

The objective of our research is two-fold. First, we describe a means to secure data at rest, MDFS, thereby removing one obstacle to realizing the vision of deploying smartphones to front line soldiers. Second, we examine the *feasibility* and real-world performance of MDFS by developing a working prototype.

With several simplifying assumptions, we develop our MDFS prototype and find performance more than adequate for the majority of practical military missions and tasks. This paper focused on the feasibility of performing calculation necessary to generate and decode encrypted file shares using MDFS. Performance increases as file size increases, so larger files are more efficient than small files. However, even though MDFS is less efficient in processing small files, given the minimal time required to process very small files, MDFS is still a viable option for secure distribution of file fragments.

Thus far, we focus solely on the performance of MDFS with respect to encryption and decryption on an Android device. Since the security of MDFS relies on the physical separation of the encrypted file fragments, the next step is to benchmark the performance of MDFS as a system of connected mobile devices. A functional prototype of MDFS was demonstrated in [20], but no benchmarks were collected. One major component to MDFS that needs attention is the directory service. The demonstration version of MDFS used a simple directory service to manage file fragments, but the directory service ran on a chosen device and was referenced by the other connected devices. This implementation presents a single point of failure, but provided sufficient functionality to successfully test the design of MDFS. The design for MDFS calls for a more robust directory service that uses a multicast channel to keep data on each device synchronized.

REFERENCES

[1] S. Ackerman, "First look: Inside the army's app store for war," *Wired*, Apr. 2011.

[2] J. Cox, "Android-based devices to be adopted by us army," *Network World*, Apr. 2011.

[3] P. Papadimitratos and Z. Haas, "Secure link state routing for mobile ad hoc networks," in *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, jan. 2003, pp. 379 – 383.

[4] R. J. Anderson and M. G. Kuhn, "Low cost attacks on tamper resistant devices," in *Proceedings of the 5th International Workshop on Security Protocols*. London, UK: Springer-Verlag, 1998, pp. 125–136. [Online]. Available: http://portal.acm.org/citation.cfm?id=647215.720528

[5] R. L. Rivest, A. Shamir, and Y. Tauman, "How to share a secret," *Communication of the ACM*, 1979.

[6] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, February 2009.

[7] *Advanced Encryption Standard (AES)*, NIST Std. FIPS 197, 2001.

[8] Google. (2011) Android ndk. [Online]. Available: http://developer.android.com/sdk/ndk/index.html

[9] A. Distefano, G. Me, and F. Pace, "Android anti-forensics through a local paradigm," in *Proc. DFRWS Annual Digital Forensics Research Conference*, Aug. 2010.

[10] "Viaforensics android forensic investigations," 2011, http://viaforensics.com/viaforensics-articles/viaforensics-aflogical-tool-android-forensic-investigations.html.

[11] B. Schneier, *Applied Cryptography*. New York, NY: John Wiley & Sons, Inc., 1996.

[12] "Truecrypt: Free open-source on-the-fly encryption," 2011, http://www.truecrypt.org/.

[13] "Blackberry department of defense solutions," 2011, http://us.blackberry.com/business/types/government/dod.jsp.

[14] "Privaltel secure phones," 2011, www.l-3com.com/gns/privatel.htm.

[15] R. A. Johnson, "The unisys stealth solution and secureparser: A new method for securing and segregating network data," White Paper, 2008.

[16] Tahoe-LAFS, "Tahoe least authority file system," 2010, http://tahoe-lafs.org/trac/tahoe-lafs.

[17] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, Jun 1960.

[18] T. Tiemens. (2011) Shamir secret sharing in java. [Online]. Available: http://sourceforge.net/projects/secretsharejava/

[19] C. D. S. James S. Plank, Scott Simmerman. (2008) Jerasure: A library in c/c++ facilitating erasure coding for storage applications. [Online]. Available: http://web.eecs.utk.edu/ plank/plank/papers/CS-08-627.html

[20] S. Huchton, "Secure mobile distributed file system MDFS," Master's thesis, Naval Postgraduate School, Monterey, CA, 2011.