DeadDrop-in-a-Flash: Information Hiding at SSD NAND Flash Memory Physical Layer

Avinash Srinivasan and Jie Wu
Temple University
Computer and Information Sciences
Philadelphia, USA
Email: [avinash, jiewu]@temple.edu

Panneer Santhalingam and Jeffrey Zamanski
George Mason University
Volgenau School of Engineering
Fairfax, USA
Email: [psanthal, jzamansk]@gmu.edu

***Abstract*—The research presented in this paper, to the best of our knowledge, is the first attempt at information hiding (IH) at the physical layer of a Solid State Drive (SSD) NAND flash memory. SSDs, like HDDs, require a mapping between the Logical Block Addressing (LB) and physical media. However, the mapping on SSDs is significantly more complex and is handled by the Flash Translation Layer (FTL). FTL is implemented via a proprietary firmware and serves to both protect the NAND chips from physical access as well as mediate the data exchange between the logical and the physical disk. On the other hand, the Operating System (OS), as well as the users of the SSD have just the logical view and cannot bypass the FTL implemented by a proprietary firmware. Our proposed IH framework, which requires physical access to NAND registers, can withstand any modifications to the logical drive, which is accessible by the OS as well as users. Our framework can also withstand firmware updates and is 100% imperceptible in the overt-channels. Most importantly, security applications such as anti-virus, cannot detect information hidden using our framework since they lack physical access to the NAND registers. We have evaluated the performance of our framework through implementation of a working prototype, by leveraging the OpenSSD project, on a reference SSD.**

Keywords*—*Anti-forensics; Covert Communication; Information Hiding; Security; Solid State Drives.

I. Introduction

With IH, a majority of the research has primarily focused on steganography, the concept of hiding information in innocuous existing files. There has also been considerable research on hiding information within file systems. The advent of SSDs, among other things, has also created new attack vectors from the view point of IH. However, little research exists in regards to IH on SSDs. From an IH view point, the combination of simplicity, standardization, and ubiquity of the traditional Hard Disk Drive (HDD) poses a major challenge. The lack of complex abstraction between physical and logical drive, detailed information on the structure of almost all file systems in use, along with open source tools enabling physical access to HDDs as well as analyzing and recovering both deleted and hidden data makes IH on the HDDs futile. This can be noted from the file system-based IH technique proposed in [1], which utilizes fake bad blocks. Although, this sounds like a generic solution since it is specific to file systems and is independent of storage media. In reality, since the mapping of logical blocks to physical flash memory is controlled by the FTL on SSDs, this cannot be used for IH on SSDs. Our proposed solution is 100% filesystem and OS-independent, providing a lot of flexibility in implementation. Note that throughout this paper, the term "physical layer" refers to the physical NAND flash memory of the SSD, and readers should not confuse it with the Open Systems Interconnection (OSI) model physical layer.

Traditional HDDs, since their advent more than 50 years ago, have had the least advancement among storage hardware, excluding their storage densities. Meanwhile, the latency-gap between Random Access Memory (RAM) and HDD has continued to grow, leading to an ever-increasing demand for high-capacity, low-latency storage to which the answer was SSDs. While flash memory has served this purpose for many years in specialized commercial and military applications, its cost has only recently decreased to the point where flash memory-based SSDs are replacing the traditional HDDs in consumer class Personal Computers (PCs) and laptops. Our proposed framework can operate under two different scenarios – 1) A user hides information strictly for personal use; 2) A group of users collude with the SSD as the *DeadDrop* [2], and any user can hide a secret message which can later be retrieved by another user, with the appropriate map-file.

In contrast to traditional HDDs, SSDs introduce significant complexities [3] including: 1) An inability to support in-place data modification; 2) Incongruity between the sizes of a *"programmable page"* and an *"erasable block"*; 3) Susceptibility to data disturbances; and 4) Imposing an upper bound on their longevity due to progressive wear and/or degradation of flash cells. While these complexities are inevitable to provide the expected performance and longevity from SSDs, they also pair well with the notion of IH. These complexities, if exploited effectively, can provide highly secure and robust IH.

Another technique, as noted by Wee [1], is similar to our proposed data hiding methodology, and is to provide the file system with a list of false bad clusters. Subsequently, the file system discounts these blocks when hiding new data, and as such, can be safely used for IH. Nonetheless, in all of the aforementioned techniques, the entire HDD can be easily read and analyzed for the existence of hidden information using both open source and commercial tools. However, SSDs as such have posed to be the biggest hurdle faced by the digital forensics community [4][5]. Hence, our proposed approach is very robust to detection and/or destruction, depending on the motive of the user.

A. Assumptions and Threat Model

We assume that there is a key exchange protocol as well as a Public Key Infrastructure (PKI) in place and known to all participants including the adversary. Figure 1 captures the basic idea of our IH framework. *Alice* and *Bob* are two users, who wish to exchange secret messages in presence of the adversary *Eve*. With our proposed framework, shown in Figure 1, they can securely exchange secret messages using the SSD as the *DeadDrop*. Additionally, they need to exchange the corresponding map-file generated during the hiding process. While several different approaches exist that can be used during the above steps, we employ the very popular PKI approach in our discussions. The tools for hiding and retrieving secret messages on the SSD are available to both Alice and Bob. If Eve wishes to extract the secret message from the SSD, then she will need both of these – the tool for retrieving the secret message and the corresponding map-file. While obtaining the tool is very hard, it cannot be completely disregarded. On the other hand, obtaining the map-file can be safely ruled out, particularly owing to the strong security properties of the underlying PKI system that strengthens session initiation with end user authentication with the help of digital certificates. If Eve, somehow, gets access to the SSD physically, she might try the following attacks. We discuss our defense mechanisms to these attacks in section VI-A.

Attack-1: Get a complete logical image of the drive using any available disk imaging tool.

Attack-2: Try to destroy the drive by erasing it.

Attack-3: Get a physical image of the entire drive.

Attack-4: Launch a Man-in-the-Middle attack to sniff the map-file, and subsequently apply it to the drive.

For any IH scheme to be effective, below are the two key features expected to be satisfied: 1) Confidentiality of the hidden message; and 2) Integrity of the hidden message. Most importantly, a very critical feature for an effective IH scheme is that it should conceal the very fact that a secret message is hidden. Our proposed framework indeed achieves all of the above three properties. We use Elliptic Curve Cryptography (ECC) algorithms for encryption, decryption, digital signatures, and for key exchange. ECC algorithms are chosen because of the strong cryptographic properties they meet with small keys sizes. As noted by Rebahi et al. [6], a 160-bit ECC key provides equivalent security to RSA with 1024-bit keys.

Our proposed "Hide-in-a-Flash" IH framework for SSDs differs from existing techniques proposed for traditional HDDs in several key aspects that have been discussed throughout this paper. However, we have identified ones that are pivotal to our research and present them in our list of contributions below and in Section III-A.

B. Our Contributions

Our contributions, at the time of this writing and to the best of our knowledge, can be summarized as follows:

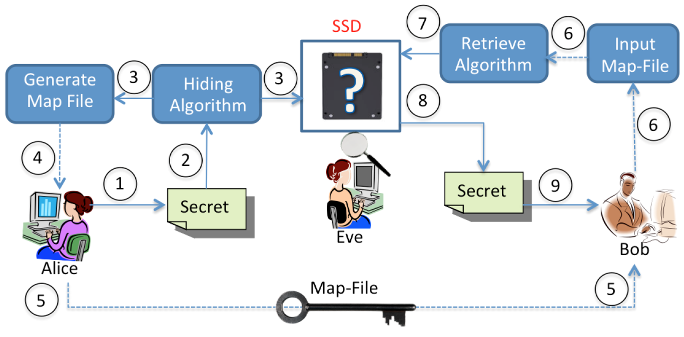

Fig. 1. Hide-in-a-Flash Threat Model.

- This is the first attempt at IH on SSDs at the NAND flash memory physical layer. We have successfully implemented our secure IH technique on the reference *OCZ Vertex Series SATA II 2.5"* SSD. The algorithms used in our framework are *wear leveling* compliant and do not impact the SSD's longevity. Additionally, our implementation of the framework does not affect data integrity and performance of the SSD.
- We have adapted the code from the *OpenSSD* project to bypass the FTL with *Barefoot Controller* firmware, which otherwise completely prevents access to physical flash memory.
- The proposed IH framework is very robust and secure – it can be implemented to be 100% undetectable without prior knowledge of its use, and 100% resistant to manufacturer's updates including destructive SSD firmware updates that completely thwart the proposed IH framework. Most importantly, it is 100% transparent to the user, the OS, and even the FTL.
- Our approach hides information within "false bad blocks" tracked by the FTL in its bad blocks list, thereby preventing any operations on those blocks and making it resistant to updates or overwriting.
- Our framework does not exploit the filesystem's data structure to hide information nor does it hide data in various slack spaces and unallocated space of a drive [7]. Consequently, it does not break the information to be hidden into bits and pieces.
- We have successfully identified functionalities of firmware which are not part of Open-SSDs documentation through firmware reverse engineering. We are working toward making it publicly available through the OpenSSD project website, so that others can continue their research using our information as the baseline.
- Finally, we have designed a tool, by leveraging OpenSSD framework, which can get a physical image of an SSD (with Barefoot flash controller) which we have tested during our evaluations. However, a few minor firmware functionalities are yet to be built into this tool.

C. Road Map

The remainder of this paper is organized as follows. We begin with a review of relevant related works in Section II. In Section III, we provide a background discussion on SSDs, specifically focusing on their departure from traditional HDDs. We also discuss the OpenSSD platform in this section. Section IV investigates various design choices we had to make in designing our system. Later, Section V presents details of the proposed IH framework followed by evaluations methods used and an analysis of the results in Section VI. Finally, in Section VII, we conclude this paper.

II. Related Work

All existing work on IH are proposed for HDDs and nothing specific to SSDs. Those that are for HDDs, the notable ones revolve around hiding information within existing file systems within slack space and unallocated space. Verhasselt [8] examines the basics of these techniques. Another technique, as noted in [1], is similar to our proposed data hiding methodology, and is to provide the file system with a list of false bad clusters. Subsequently, the file system discounts these blocks when hiding new data, and as such can be safely used for IH. Nonetheless, in all of the aforementioned techniques, the entire HDD can be easily read and analyzed for the existence of hidden information using both open source and commercial tools. However, SSDs as such have posed to be the biggest hurdle faced by the digital forensics community [4][5]. Hence, our proposed approach is very robust to detection and/or destruction, depending on the motive of the user.

According to McDonald and Kuhn [9], cryptographic file systems provide little protection against legal or illegal instruments that force data owners to release decryption keys once the presence of encrypted data has been established. Therefore, they propose *StegFS*, a steganographic file system, which hides encrypted data inside unused blocks of a Linux *ext2* file system.

RuneFS [10] hides files in blocks that are assigned to bad blocks *inode*, which happens to be *inode 1* on *ext2*. Forensic programs are not specifically designed to look at bad blocks *inode*. Newer versions of *RuneFS* also encrypt files before hiding them, making it a twofold problem. On the other hand, *FragFS* [11] hides data within Master File Table (MFT) of an New Technology File System (NTFS) volume. It scans the MFT table for suitable entries that have not been modified within the last year. It then calculates how much free space is available and divides it into 16-byte chunks for hiding data.

Khan et. al. [12] have applied steganography to hard drives. Their technique overrides the disk controller chip and positions the clusters according to a code, without which, hidden information cannot be read. In [13], authors propose a new file system vulnerability, *DupeFile*, which can be exploited for IH. The proposed approach hides data in plain sight in the logical disk by simply renaming malicious files with the same name as that of an existing good file. Since the renaming is done at the raw disk level, the OS does not complain to the end user of such file hiding. In another IH method, in [14], authors propose information hiding in file slack space. This technique, called HideInside, splits a given files into chunks, encrypts them, and randomly hides them in the slack space of differ files. The proposed technique also generates a map-file that resides on a removable media, which will be used for retrieval and reconstruction of the randomly distributed encrypted chunks.

In [15], Nisbet et al. analyze the usage of TRIM as an Anti-Forensics measure on SSDs. They have conducted experiments on different SSDs running different operating systems, with different file systems to test the effectiveness of data recovery in TRIM enabled SSDs. Based on their experiments it can be concluded that, with TRIM enabled, Forensic Investigators will be able to recover the deleted data only for a few minutes from the time TRIM was issued.

Wang et. al. [16] have successfully hidden and recovered data from flash chips. Here, authors use the term "flash chips" to refer to removable storage devices like USB flash drives. They use variation in the program time of a group of bits to determine if a given bit is a 0 or a 1. They convert the data to be hidden into bits, and determine the blocks required. Authors have come up with a method to program a group of bits overcoming default page-level programming. While their method is quite robust, it suffers from a significant downside, which is the amount of information that could be hidden. Their method can hide up to 64 MB of data on a 32 GB flash drive, while our proposed IH can hide up to 2 GB of information on a 32 GB SSD, which is an increase in hiding capacity of the channel, by a factor of 16.

III. SSD Background

In contrast to the mechanical nature of the traditional HDDs, an SSD is more than just a circuit board containing numerous flash memory packages and a controller to facilitate the interface between the OS and the physical flash memory. SSDs may utilize either the NOR flash or the NAND flash memory. As the latter is relatively cheap it is highly used for consumer SSDs.

A. Salient Features

Below is a list of salient features of SSDs:

1. Flash Memory: At the lowest level, each flash memory package contains thousands of cells, each capable of holding one or more bits. While read and write operations on a cell are relatively fast, physical limitations imposed by the storage medium necessitate cell erasure before overwriting it with new information. Flash memory cells are logically grouped into pages. A page is the basic unit of reading and writing. Pages are grouped into blocks, which is the basic unit of erasure. Blocks are grouped into dies, and dies are grouped into flash memory packages, aka banks. Within a SSD, multiple flash memory packages are grouped to provide the total capacity of the drive.

2. Flash Translation Layer (FTL): In order to manage the complexities of the physical layout, optimize the use and endurance of the flash memory, and provide the OS with the traditional block device interface of storage devices, SSDs contain a controller which implements an additional layer of abstraction beyond traditional HDDs

TABLE I. REFERENCE SSD OCZ VERTEX SPECIFICATION

Total number of banks	8	Dies per Bank	2
Blocks per Die	4096	Pages per Block	128
Cell Type	2-level cells	Cells per Page	17, 256
Bits per Cell	34, 512	Total Size	32 GB
Advertised capacity	30 GB	Over-provisioning	2 GB

known as the FTL. Below are the three fundamental operations of the FTL – 1) *logical to physical block mapping*; 2) *garbage collection*; and 3) *wear leveling.*

3. Pages Size, Spare Bytes & Error Correction: Traditional HDDs implement storage based on a predefined allocation unit called a *sector*, which is a power of two. To facilitate the mapping of logical blocks to physical flash memory, the flash memory is manufactured with page sizes also being powers of two. However, since flash memory is susceptible to data disturbances caused by neighboring cells, it is critical for the FTL to implement an error correction mechanism. To accommodate the storage requirements of the Error Correction Code (ECC), the flash memory is manufactured with additional spare bytes, in which FTL can store the ECC. For instance, a flash memory page may consist of 8192 bytes with 448 additional bytes reserved for ECC.

4. Bad Blocks: Due to the physical characteristics of the flash memory as well as cell degradation over time, flash memory packages may ship with blocks that are incapable of reliably storing data, even with an ECC employed. These blocks are tested at the factory and marked in a specific location within the block to identify them as initial bad blocks. During SSD manufacturing, the flash memory is scanned for bad block markings and an initial bad block list is stored for use by the FTL. Beyond this initial list of bad blocks, the FTL must keep the list updated with the inclusion of newly identified bad blocks at runtime.

5. Over-Provisioning: Write amplification, a serious concern with SSDs, is an inevitable circumstance where the actual amount of physical information written is greater than the amount of logical information requested to be written. On SSDs, this occurs for several reasons, including but not limited to: *need for ECC storage*, *garbage collection*, and *random writes to logical blocks.* In order to maintain responsiveness when the drive is near capacity and longevity when flash memory cells begin to fail, SSDs may be manufactured with more flash memory than they are advertised with, a concept known as over-provisioning. For example, an SSD containing 128GB of flash memory may be advertised as 100GB, 120GB, or with 28%, 6.67%, or 0% over-provisioning, respectively.

B. OpenSSD

The *OpenSSD Project* [17] was created by *Sungkyunkwan University* in Suwon, South Korea in collaboration with *Indilinx*, to promote research and education on SSD technology. This project provides the firmware source code for the *Indilinx Barefoot Controller* used by several commercial SSD manufacturers including *OCZ*, *Corsair*, *Mushkin*, and *Runcore IV*. The firmware code provided in this project is an open source implementation, and a version of research implementation of a complete SSD known as *Jasmine Board*, is available for purchase.

Table I summarizes the specifications of the reference SSD. During the course of our research, we learned that the *Jasmine Board* uses the same Indilinx Barefoot controller firmware as our reference SSD, which is an *OCZ Vertex Series SATA II.* We also learned that the firmware installation method used by the *OCZ Vertex* SSD. Furthermore, the *Jasmine Board* involved setting of a jumper on the SSD, to enable a factory or engineering mode. Upon setting the jumper on the reference SSD and compiling and running the firmware installation program adapted from the OpenSSD Project, we were able to connect to the SSD in factory mode with physical access to NAND flash memory chips, bypassing the FTL.

IV. Framework Design Choices

We have successfully identified the following critical pieces of information from the OpenSSD code about firmware functionalities through reverse engineering, information which was otherwise not available on OpenSSD documentation:

- Block-0 of each flash memory package is erased and programmed during the firmware installation process.
- First page of block-0 contains an initial bad block list before the firmware installation, which will be read by the installer, erased along with the remainder of block-0, and programmed with identical information as part of the installation process.
- In addition to page-0, a minimal set of metadata such as the firmware version and image size is programed into pages-1 through 3, and the firmware image itself is programmed into sequential pages starting with page-4.

Based on our analysis of the firmware, we came up with the following storage covert channels that can be used in designing our IH framework.

1) *Manipulating the FTL data structure:* We considered the possibility of modifying the firmware and utilizing it for IH. One possibility was to redesign the wear leveling algorithm such that some blocks will never be considered for allocation.
2) *Utilizing the spare bytes:* All spare bytes available per page are not completely used. Some of the spare bytes are used for storing ECC. Thus the remaining spare bytes can be used for IH.
3) *Using the blank pages in block zero:* Only a few pages of block zero were used during firmware installation; the remaining were free for IH.
4) *Manipulating the initial bad block list:* By inserting new bad blocks in the bad block list in block zero, and using them for IH.

With due diligence, we decided against the first three methods because OpenSSD's implementation is a stripped down version of the actual firmware. This means, the full-blown version of the firmware could easily overwrite

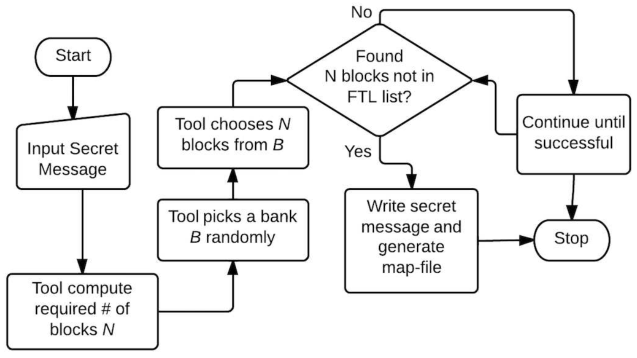
Start
Input Secret Message
Tool compute required # of blocks N
Tool picks a bank B randomly
Tool chooses N blocks from B
Found N blocks not in FTL list?
No
Yes
Continue until successful
Write secret message and generate map-file
Stop

Fig. 2. Flow chart illustrating the initial system design.

any modifications we make to the FTL data structure. Therefore, the first method is not very useful. Though the unused spare bytes can be used, it was uncertain whether or not the firmware would use them during the life of the SSD. Hence, the second method was decided against. As with the blank pages on block 0, they did not provide much room for hiding information because of which third method was ruled out. Finally, we had narrowed down our choice to one stable and robust method – *manipulation of the initial bad block list*, details of which follow in the next section.

V. Information Hiding Framework

A. Process Overview

Scenario: Alice and Bob could be friends or total strangers communicating over an insecure channel. Their goal is to exchange secret messages over the insecure channel in the presence of Eve, the adversary. As noted in Section I-A, we will not discuss the details of key negotiation as plenty of known works exist on this subject. For simplicity, we assume the use of PKI in our discussions.

Step-1: Alice has the secret message M_{sec} she wishes to share with Bob.

Step-2: She generates a random session key K^{Alice}_{rand} that she inputs to the Hiding Algorithm along with M_{sec}.

Step-3: M_{sec} is encrypted with K^{Alice}_{rand} generating the following message:

$$E[M_{sec}]_{K^{Alice}_{rand}} \tag{1}$$

This message is then written into fake bad blocks. Simultaneously, a map-file F_{map} is generated. The purpose of F_{map} is identifying blocks holding the secret message.

Step-4: Alice then encrypts F_{map} and K^{Alice}_{rand} with her private key K^{Alice}_{prv} generating the following message. This is necessary to provide Bob with a message integrity verification service.

$$M^{verify}_{sec} = E[F_{map}||K^{Alice}_{rand}]_{K^{Alice}_{prv}} \tag{2}$$

She then encrypts the M^{verify}_{sec} message with Bob's public key K^{Bob}_{pub}, generating the following message.

$$_{conf}M^{verify}_{sec} = E[(F_{map})||(K^{Alice}_{rand})]_{K^{Bob}_{pub}} \tag{3}$$

```
 1: $file \Leftarrow user\ input\ secret\ message$
 2: $blocksRequired = \frac{sizeOf(file)}{sizeOf(block)}$
 3: $leastBadBlockBank = 0$
 4: for $i = 0$ to $bank.count$ do
 5:     if $(i.badBlocksEraseCount <$
 6:         $leastBadBlockBank.badBlocksEraseCount)$
    then
 7:         $leastBadBlockBank = i$
 8:     end if
 9: end for
10: while $(leastBadBlockBank.blocksinbadBlockList)$
    &&
11: $(leastBadBlockBank.blocks.metadata{=}{=}keyword)$
    &&
12: $(count < blocksRequired)$ do
13:     $newBadBlock.count = leastBadBlockBank.block;$
14:     $count++$
15: end while
16: $Payload = Encrypt(metadata, file)$
17: $payload.Write()$
18: $newKey = encode(leastBadBlockBank, newBadBlock)$
```

Fig. 3. Algorithm for Hiding

The message $_{conf}M^{verify}_{sec}$ encrypted with Bob's public key provides confidentiality service for message exchange. Note that, the use of encryption keys in this specific order also provides communication endpoint anonymity.

Step-5: Alice sends $_{conf}M^{verify}_{sec}$ to Bob. On receiving this message, Bob uses his private key K^{Bob}_{prv} to decrypts the message extracting M^{verify}_{sec}. Then, Bob uses K^{Alice}_{pub} to extract the F_{map} and K^{Alice}_{rand}. Note that Alice and Bob could use either a client-server or P2P architecture to eliminate the need for physical access to the SSD.

Steps-6 & 7: Bob extracts F_{map} and K^{Alice}_{rand}. He inputs F_{map} to the Retrieving algorithm, presented in Figure 4, which applies it to the SSD to retrieve and reconstruct the encrypted secret message $E(M_{sec})_{K^{Alice}_{rand}}$.

Steps-8 & 9: Bob uses K^{Alice}_{rand} to decrypt $E(M_{sec})_{K^{Alice}_{rand}}$ and finally extracts the secret message M_{sec}.

B. Initial Design

In the first phase of our research, we modified the open SSD framework to our specific requirements and tried to hide a test file. As illustrated in the flowchart in Figure 2, we designed a simple tool with a command line interface that receives *filename* as input from the user. Subsequently, the tool decides the number of bad blocks to be allocated for hiding that file based on the file size. Finally, the tool chooses a random bank on the SSD and allocates the required number of blocks. While allocating the blocks, we made sure that the blocks are not part of the initial bad block list the SSD was shipped with. If the allocation is successful, copy the file to the specified blocks and create the map-file (used to identify the bank and block). The map-file is used for the retrieval process.

C. Challenges with the Initial Design

In this section, we address some of the challenges we face with the initial design of our IH framework.

```
1: $map\text{-}file \Leftarrow file\,received\,from\,sender$
2: $bankAndBlock = decode(map\text{-}file)$
3: metadata = $bankAndBlock.Read()$
4: $decrypt(metadata)$
5: $decrypt(file.Read())$
6: $file.Write()$
7: if then$(ReadandErase)$
8:     $Erase(bankAndBlock)$
9:     $eraseCount++$
10:    $eraseCount.Write()$
11: end if
```

Fig. 4. Algorithm for Retrieving

1) As the number of bad blocks increase, firmware installation fails, rendering the drive useless.
2) If we hide data in blocks that hold the firmware, then firmware reinstallation would rewrite these blocks, irrespective of their inclusion in the bad block list.
3) As part of experiment, we did a complete physical image of drive, including the bad blocks, and were able to find that the hidden files signature was visible along with the metadata.
4) Every time we added a new bad block and hid data, we had to reinstall the firmware. This was required because the firmware would only keep track of the bad blocks that were in the list when the firmware was installed.

D. Enhanced design

We shall now discuss our enhanced design with improvements to overcome the challenges of the initial design as delineated above.

- Uninstall the SSD firmware.
- Enable the user to specify the banks on which bad blocks have to be assigned and the number of bad blocks to be allocated on each bank.
- Have the tool allocate the user specified number of blocks on user specified banks and append these blocks to the bad block list maintained by the FTL.
- Reinstall the firmware on the SSD.
- Add metadata to user-created bad blocks to distinguish them from firmware identified bad blocks. In our prototype implementation of the IH framework on the reference SSD, we reserve the very first byte of user-specified bad blocks to keep track of its erase count, which serves as the metadata. The erase count variable is initialized to 0, and is incremented every time new data is written to the corresponding block, since write operation on as SSD is preceded by an erase operation.

Note that, as shown in the Table IV, the first byte is the erase count which is followed by the data. This helps to select the least-erased block every time we pick a block for hiding, and make sure the block is empty.

As can be seen in Figure 3, we pick blocks from banks that have the least erase count. We achieved this by keeping track of the cumulative erase count, comparing the cumulative erase count of all the bad blocks in the banks, and finally pick the one with the least value. Next, in order to address the firmware overwrite problem, we started excluding the first 32 blocks (This was done during the pre-allocation of bad blocks) in every bank. Finally, in order to escape from the physical imaging, we started to encrypt both the metadata and the file. While retrieving the file, we gave an option for the user to erase and retrieve the file or just retrieve the file alone. If user chooses to erase and retrieve the file, we erased the block and increased the erase count by one such that the block was not used until all the other bad blocks have reached a similar erase count.

TABLE II. Information retrieval under different scenarios.

Condition	Hidden File Retrieved
Firmware Reinstallation	Yes
NTFS Formatted Drive	Yes
Partitioned Drive	Yes
Populate Disk to Full Capacity	Yes
Factory Mode Erase	No

VI. Evaluation of Enhanced Design

We confirm through evaluations that our framework is 100% undetectable and robust to firmware updates.

Experiment-1: We test the conditions under which the secret message is retained and retrievable. Table II summarizes the different scenarios under which we evaluated our framework on the reference SSD. As can be seen, we were able to retrieve the secret message in every scenario except when erased in the factory mode. However, this operation requires access to the SSD in factory mode as well as knowledge of factory commands specific to the firmware in use, without which it is impossible to wipe the physical memory of the SSD.

Experiment-2: With this experiment, our objective was to determine the maximum number of blocks that can be tagged as bad, both firmware-detected and user-specified, before the firmware installation starts to fail. This would give us the total amount of data that can be hidden safely, using our IH framework, without causing drive failure. We also wanted to know if hiding data would result in any changes, as perceivable by a normal user. For this, we gradually increased the bad block count in each bank, in increments of 10. With every increment, we did the following – 1) increment the counter tracking the bank's bad block count ; 2) re-install the firmware; 3) install the file system on top of the firmware; and 4) check the available logical disk space. During the experiments, we determined that the threshold for number of blocks, as a fraction of the total number of blocks per bank, that can be tagged as bad, is approximately 2.6% per bank. This is equivalent to 218 blocks per bank. Beyond this, the firmware installation fails. We have summarized the results in Table III. Furthermore, based on the results, we conclude that bad block management is a part of over-provisioning and hence, a typical user won't notice any changes to the logical structure of the disk when information is hidden, proving that our system is 100% imperceptible by end users.

Experiment-3: Finally, we wanted to test if any of the existing computer forensic tools would be able to discover the

TABLE III. Drive size with different bad block count.

Bad Block Count	**Drive Size**
25	29.7GB
50	29.7GB
75	29.7GB
100	29.7GB
109	29.7GB

secret messages hidden on an SSD using our IH framework. We used freeware tools like WinHex and FTK imager. Both the tools, though very popular and powerful, were unsuccessful in getting past the logical disk. We confidently conclude that none of the existing computer forensics tools, at the time of this writing and to the best of our knowledge, have the capability to access the physical layer of an SSD.

TABLE IV. Manipulated bad block layout.

Number of Bytes	**Information**
1	Erase Count
Remaining bytes	Hidden data

A. Defense against attacks

In Section I-A, we presented four possible attacks that an adversary, Eve, can potentially try against our IH framework. We shall discuss why none of these attacks will be successful against our IH framework.

- ***Attack-1 Defense:*** The hidden information is not part of the logical drive. Hence, Eve will not benefit from a logical image of the DeadDrop SSD.
- ***Attack-2 Defense:*** SSD blocks that are tracked as bad blocks by the FTL firmware are never accessed and erased by the firmware. Hence, this attack will not be successful.
- ***Attack-3 Defense:*** Currently, it is impossible for Eve to make a physical image of the SSD without our modified OpenSSD software. Additionally, Eve should have the ability to access the SSD into factory mode with appropriate jumper settings, and should know the firmware functionalities that we have identified beyond those provided in the OpenSSD documentation. Beyond this, she would still need the random session key generated by Alice that was used to encrypt the secret message. Additionally, she would need Bob's (recipient of the map-file) private key to decrypt the random session key and the map-file, without which the secret message cannot be reconstructed. Therefore, the feasibility of this attack can be safely ruled out.
- ***Attack-4 Defense:*** Assuming Eve is able to sniff the map-file from the traffic between Alice and Bob, as discussed in Section V, she still needs Bob's private key to decrypt the map-file. Bob's private key, however, is not accessible to anyone other then Bob himself. Hence, this attack is ruled out.

VII. Conclusion and Future Work

In this paper, we have presented the design, algorithms, and implementation details of secure and robust IH framework that can hide information on SSDs at the physical layer. We have presented multiple methods for IH highlighting their strengths and weaknesses. Finally, we have evaluated the proposed framework through real world implementations on a reference SSD running *Indilinx's Barefoot flash controller*, which is used by various SSD manufacturers including, *Corsair, Mushkin,* and *Runcore IV* [18]. Consequently, this IH framework can be used on SSDs from different manufacturers, making it quite pervasive and ubiquitous.

The ability to interface SSDs with the OpenSSD platform and bypass the FTL has significant impact on the Digital Forensics community. Also, this is the first step toward potential antiforensics techniques. Having discovered this possibility, law enforcement agencies can now focus on potential information theft and antiforensics attacks on SSDs, which otherwise was deemed near impossible. As part of our future work, we would lim to investigate the potential of integrating more support for other popular proprietary firmware. This will enable to expand the existing project to support forensics investigation of SSDs from a wide array of manufacturers.

Acknowledgment

The authors would like to thank the Defense Cyber Crime Centre, Linthicum, Maryland, USA, for the reference SSD drive used in the experiments.

References

[1] C. K. Wee, "Analysis of hidden data in NTFS file system," 2013, URL: http://www.forensicfocus.com/hidden-data-analysis-ntfs [accessed: 2013-04-25].

[2] "DeadDrop," 2014, URL: http://en.wikipedia.org/wiki/Dead_drop [accessed: 2014-07-26].

[3] L. Hutchinson, "Solid-state revolution: in-depth on how SSDs really work," 2014, URL: http://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work/2/ [accessed: 2014-07-25].

[4] G. B. Bell and R. Boddington, "Solid state drives: the beginning of the end for current practice in digital forensic recovery?" vol. 5, no. 3. Association of Digital Forensics, Security and Law, 2010, pp. 1–20.

[5] C. King and T. Vidas, "Empirical analysis of solid state disk data retention when used with contemporary operating systems," vol. 8. Elsevier, 2011, pp. S111–S117.

[6] Y. Rebahi, J. J. Pallares, N. T. Minh, S. Ehlert, G. Kovacs, and D. Sisalem, "Performance analysis of identity management in the session initiation protocol (sip)," in *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on.* IEEE, 2008, pp. 711–717.

[7] E. Huebnera, D. Bema, and C. K. Wee, "Data hiding in the ntfs file system," vol. 3, 2006, pp. 211–226.

[8] D.Verhasselt, "Hide data in bad blocks," 2009, URL: http://www.davidverhasselt.com/2009/04/22/hide-data-in-bad-blocks/ [accessed: 2009-04-22].

[9] A. D. McDonald and M. G. Kuhn, "Stegfs: A steganographic file system for linux," in *Information Hiding.* Springer, 2000, pp. 463–477.

[10] Grugq, "The art of defiling: Defeating forensic analysis on unix file systems," *Black Hat Conference*, 2005.

[11] I. Thompson and M. Monroe, "Fragfs: An advanced data hiding technique," 2004, URL: http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06Thompson-up.pdfTrueCrypt(2006) [accessed: 2014-6-02].

[12] H. Khan, M. Javed, S. A. Khayam, and F. Mirza, "Designing a cluster-based covert channel to evade disk investigation and forensics," vol. 30, no. 1. Elsevier, 2011, pp. 35–49.

[13] A. Srinivasan, S. Kolli, and J. Wu, “Steganographic information hiding that exploits a novel file system vulnerability,” in *International Journal of Security and Networks (IJSN)*, vol. 8, no. 2, 2013, pp. 82–93.

[14] A. Srinivasan, S. T. Nagaraj, and A. Stavrou, “Hideinside – a novel randomized & encrypted antiforensic information hiding,” in *Computing, Networking and Communications (ICNC), 2013 International Conference on.* IEEE, 2013, pp. 626–631.

[15] A. Nisbet, S. Lawrence, and M. Ruff, “A forensic analysis and comparison of solid state drive data retention with trim enabled file systems,” in *Proceedings of 11th Australian Digital Forensics Conference*, 2013, pp. 103–111.

[16] Y. Wang, W.-k. Yu, S. Q. Xu, E. Kan, and G. E. Suh, “Hiding information in flash memory,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy, S&P’13.* IEEE Computer Society, 2013, pp. 271–285.

[17] “OpenSSDWiki,” 2013, URL: http://www.openssd-project.org [accessed: 2013-04-25].

[18] “Barefoot,” 2014, URL: http://en.wikipedia.org/wiki/Indilinx [accessed: 2013-10-02].