HiPDS: A Storage Hardware-independent Plausibly Deniable Storage System

Niusen Chen, Bo Chen, *Senior Member, IEEE*

***Abstract*—A plausibly deniable storage (PDS) system not only conceals the plaintext of sensitive data, but also hides their very existence. It can essentially mitigate a novel coercive attack, in which the adversary captures both a victim and his/her device, and coerces the victim to disclose the sensitive data. A rich number of PDS systems have been designed in the literature. However, all of them are specifically designed for a certain type of storage hardware. In this work, we have designed HiPDS, the first storage Hardware-independent Plausibly Deniable Storage system. HiPDS can defend against a multi-snapshot adversary which can have access to both the external storage and the internal memory at multiple checkpoints over time. By leveraging our adapted chameleon hash, we encode the sensitive data into the non-sensitive cover data in a fine-grained manner, so that both the existence and the access of the sensitive data on the external storage device can be plausibly denied. In addition, to prevent the sensitive data from being compromised in the memory, the encoding/decoding process is run in a secure memory region isolated by the trusted execution environment. A salient feature of HiPDS is that it can ensure deniability on any types of storage media, which is essentially important for users who may change the external storage devices over time. Security analysis and experimental evaluation confirm that HiPDS can ensure deniability against the multi-snapshot adversary at the cost of an acceptable overhead.**

***Index Terms*—confidentiality, coercive attacks, plausible deniability, storage hardware-independent, chameleon hash, trusted execution environment**

I. Introduction

Modern computing devices have been increasingly used to store and process sensitive data and, a critical issue is to ensure confidentiality of the sensitive data over time. This is not only essential for protecting privacy of users, but also necessary for remaining compliant with regulations such as Health Insurance Portability and Accountability Act (HIPAA) [30], Sarbanes-Oxley Act (SOX) [60], etc. To ensure confidentiality of the data, we can simply encrypt them, at either the file level or the disk level. Traditional encryption algorithms, however, transform the problem of data protection to the problem of key protection. Such a transformation is vulnerable to a coercive attack [14], [25], [64], in which the adversary can capture both the device and the victim user, and coerce the user to disclose the key needed for decrypting the ciphertext found in the victim device.

To defend against the coercive attacks, a new plausibly deniable encryption (PDE) [18] has been designed. The PDE transforms a sensitive message to ciphertext in a special way, so that upon decryption, when using a true key, the original sensitive message will be obtained, but, when using a decoy key, a decoy non-sensitive message will be obtained. Upon being coerced, the victim user can simply disclose the decoy key and, using the decoy key, the adversary can decrypt the ciphertext to obtain the non-sensitive message, being unaware of the existence of the sensitive message. The PDE is essentially different from traditional encryption, as it not only conceals the data plaintext, but also plausibly denies its very existence, adding an extra layer of confidentiality protection.

The concept of PDE has been leveraged to build a novel plausibly deniable storage (PDS) system which can protect both the plaintext and the existence of the sensitive data being stored. The PDS has been built for computing devices using hard disk drives (HDD) as external storage. Such systems include TrueCrypt [67], VeraCrypt [29], steganographic file systems [8], [37], [50], [54], Hive [14], Datalair [19], Artifice [12], PD-DM [23], etc. Those PDS systems however, have been specifically designed for the block device and may not be secure for computing devices using flash memory as external storage, due to the unique security compromises rooted at flash memory. Especially, when the adversary can gain access to the raw flash, it may observe various special traces of hidden sensitive data, compromising their deniability [26], [41].

The PDS systems were later designed for computing devices using flash storage media. Several of them [21], [22], [33], [39], [64], [73] still deploy PDS on the block layer, taking care of various security and usability issues for mobile environments. However, they all suffer from deniability compromises when the adversary can have access to the internal flash memory and extract traces of hidden sensitive data which are invisible to the block layer. Directly integrating the PDS with the raw flash memory can eliminate the aforementioned compromise. Such designs include DEFY [55], DEFTL [41], INFUSE [24], PEARL [25], CrossPDE [27], which leverage the unique nature of flash storage media to achieve deniability.

We have observed a few limitations present in those existing PDS system designs: First, they are strongly coupled with a specific type of storage medium and cannot be used in a different type of storage medium. Specifically, the HDD-based PDS systems cannot be deployed on the flash media due to the potential deniability compromises and, the flash-based PDS systems cannot be deployed on the HDDs due to the requirement of flash memory hardware support. Second, they

Manuscript received 08-Apr-2023; revised 06-Oct-2023. This work was supported by the US National Science Foundation under Grant CNS-1928349, CNS-2225424, and DGE-2043022 (Corresponding author: Bo Chen.)

The authors are with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA (e-mail: niusenc@mtu.edu; bchen@mtu.edu).

The final version of the paper is available in IEEE Xplore [3].

mostly do not handle deniability compromises in the internal memory. The past processing of the hidden sensitive data essentially leaves traces in the internal memory and, which may be identified by the adversary via memory forensics, leading to deniability compromises.

Overcoming the aforementioned limitations is of essential importance for PDS system designs because: 1) A PDS system design which heavily relies on the underlying storage medium would be cumbersome in real world. The user may change the external storage device of his/her computer over time, e.g., upgrading an HDD to an SSD. If a storage hardware-dependent PDS system is used, the user needs to find a replacement of his/her PDS system after changing the external storage device, a challenging task for those who lack computer skills. 2) For a long time, the compromise of the PDS system in the internal memory has been ignored as the focus was on the deniability guarantee on the external storage device. Yet, a simple solution would be restarting the computing device to eliminate the traces of the hidden sensitive data in the volatile RAM, but it would not be effective due to the retention effect[1] of the RAM [35], [36]. We need an effective solution which can prevent the compromise of the PDS in the internal memory.

In this work, we overcome the aforementioned limitations by designing HiPDS, a storage Hardware-independent Plausibly Deniable Storage system which can simultaneously ensure deniability regardless of the underlying storage hardware being used and, meanwhile, effectively eliminate the deniability compromises present in the internal memory. The design of HiPDS is based on two key insights:

Insight #1: adapting chameleon hash to plausibly hide sensitive data on the external storage device. A chameleon hash is associated with both a public and a private key and, without having access to the private key, a chameleon hash function works just like a traditional collision-resistant hash function (this implies that we can deny the chameleon hash function as a regular collision-resistant hash function, so that the adversary does not even know the existence of the private key). However, when having access to the private key, collisions of the hash function can be efficiently found. It is therefore possible to hide the sensitive data as one of the collisions for the given cover data and the hash value, if we choose the *right* Chameleon hash construction and *carefully* adapt the construction. Specifically, the user only stores the cover data and the corresponding hash value in the external storage and, the sensitive data are hidden among the cover data and the hash value. Without knowing the private key, the adversary cannot figure out the existence of the hidden sensitive data (this implies that it is also impossible for the adversary to figure out the plaintext of the sensitive data). Upon being coerced, the user can simply disclose the public key (i.e., the *decoy key*) and claim that the public key is used to compute the hash value for the cover data; using the public key, the adversary can only verify whether the hash value is corresponding to the cover data, without being able to be aware of the existence of the hidden sensitive data (and the private key as well). On the contrary, the user can efficiently extract the sensitive data from the cover data and the hash value using the private key.

A salient advantage of using the chameleon hash is that it hides the sensitive data among the cover data, without concerns about any potential deniability compromises in the storage hardware [41]. This is because the sensitive data remain hidden in the cover data along the entire storage data path. In other words, the design remains secure independent of the underlying storage hardware. In addition, the design remains secure even if the adversary can have access to the external storage multiple times over time, as every access of the hidden sensitive data can be plausibly denied as that of the cover data.

We have faced a few challenges when leveraging the chameleon hash for PDS. First, most existing chameleon hash constructions cannot be immediately used to hide sensitive data as they compress the input from a large field to generate a condense hash value from a small field, making it infeasible to re-generate the original input from the hash value (i.e., extracting the sensitive data turns impossible). To address this challenge, we have identified the discrete log-based chameleon Hash construction (DCH) proposed by Krawczyk et al. [43] which does not compress the input when generating the hash value. Second, the DCH cannot be immediately used to hide the sensitive data with plausible deniability, as the adversary can compromise the deniability by identifying the existence of a redundant random number, which is unfortunately necessary for extracting the hidden sensitive data via chameleon hash. To address this challenge, we have Adapted the DCH (ADCH) construction so that the sensitive data can be hidden and extracted without storing the redundant random number. Third, for the ADCH, it is computationally expensive to find a suitable random number r'. A brute-force search has a computational complexity of $O(q)$, where q is a large prime number. To address this challenge, we have designed a novel algorithm which can efficiently compute r' in constant time.

Insight #2: leveraging trusted execution environment (TEE) to prevent deniability compromises in the memory. The adversary may have access to the memory and identify the past existence of the hidden sensitive data. To avoid such a compromise, we can keep track of any traces of the hidden sensitive data in the memory, and purge the traces upon finishing processing them. This immediate solution however, could be cumbersome as traces of the hidden sensitive data may be here and there in the memory and, keeping track of and purging them itself would be challenging. Our solution is: 1) We process the hidden sensitive data in a secure memory region isolated by the TEE. The TEE can ensure that even if the adversary can compromise the OS and perform forensic analysis over the memory, it cannot identify any traces of the hidden sensitive in the secure memory. By doing this, we can reduce our efforts to safeguard only the sensitive data and private key during their entry into the TEE. 2) To ensure that the use of TEE is deniable, the TEE for the deniable storage system should only be launched when processing hidden sensitive data and, once the processing is finished, the TEE should be terminated. In this way, the adversary cannot interact with the TEE online to compromise the deniability and, the prior running of the TEE can be plausibly denied as running TEE-based

[1]If data are staying long in the memory, they may be "remenbered" by the memory cells.

confidential computing tasks (e.g., private web query [51], secure payment [70]) instead of a deniable storage system.

Note that the TEE and the Chameleon hash are completely orthogonal in our design. However, the TEE and the other exiting PDS systems [8], [14], [19], [21], [22], [41], [50], [64], [73] are not necessarily orthogonal, because: most of them encrypt the sensitive data; after having encrypted the sensitive data in the TEE, the TEE (e.g., the SGX) typically needs to rely on the insecure operating system to write the encrypted sensitive data to the external storage; therefore, the untrusted OS can observe the encrypted ciphertext and suspect something sensitive has been hidden inside. This deniability compromise is not present in our design, as the sensitive data have been encoded into the cover data using the Chameleon hash once they get out of the TEE and, the cover data are clearly non-sensitive and hence not suspicious.

Contributions. Our contributions are summarized below:

- We have designed HiPDS, the first secure plausibly deniable storage system which is deployable in any storage hardware. HiPDS leverages both the cryptography (i.e, chameleon hash) and the TEE to protect the hidden sensitive in both the external storage and the internal memory.
- To the best of our knowledge, HiPDS is the first work which adapts chameleon hash for plausible information hiding, an independent research contribution.
- We have analyzed the security of HiPDS. In addition, we have implemented a prototype of HiPDS and experimentally evaluated its performance.

II. Background

Plausibly deniable storage (PDS) system. A PDS system can be built using a hidden volume technique [67], in which the sensitive data are stored in a hidden volume, and the entire hidden volume will be encrypted using a true key and stored at the end of the disk starting from a secret offset. The adversary is not able to identify the existence of the hidden volume which is indistinguishable from the randomness filled in the entire disk initially. The PDS system can be also built using a steganographic file system [8], [50], in which the sensitive data are stored hidden at random locations of the disk and are indifferentiable from the randomness filled to the disk initially.

Chameleon hash. As a non-standard type of collision resistant hash function, a chameleon hash function [43] (or a trapdoor hash function) is associated with a pair of public and private keys. The private key is also called the trapdoor. The chameleon hash function typically has some interesting properties: 1) Anyone who knows the public key can compute the hash function; 2) Without knowing the trapdoor, the hash function is similar to a traditional hash function that is collision resistant; 3) Anyone who knows the trapdoor can efficiently find collisions for every given input.

In some sense, without knowing the trapdoor, the chameleon hash function is equivalent to a regular collision resistant hash function (i.e., *the hash function construction itself will not lead to the deniability compromise*). However, by knowing the trapdoor, the collisions can be efficiently computed. Due to the nice properties of chameleon hash, it has been used extensively in blockchain redaction [11], verifiable image revision [71], E-cheque protocol [66], etc. In this work, we have initiated the study of leveraging chameleon hash for plausible information hiding against coercive attacks.

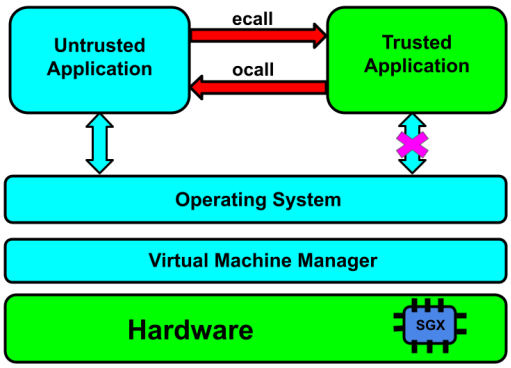

Fig. 1. Intel SGX.

Trusted execution environment. A trusted execution environment (TEE) has been designed to support secure computation in modern computing devices [59]. A TEE often relies on the hardware to create a secure area which is isolated from the normal operating system, so that the computation running in the this area remains protected even if the OS is compromised. Popular TEE implementations for personal computers include Intel software guard extensions (SGX) and AMD secure encrypted virtualization (SEV)/secure memory encryption (SME)/transparent SME (TSME). We will use Intel SGX as an example to illustrate how a TEE works. SGX is a set of security related instructions that are embedded in the modern Intel CPUs [31]. Its structure is shown in Figure 1. It allows defining specific areas in memory as secret areas known as enclaves. The content and the code in an enclave is protected and cannot be accessed by any other processes outside the enclave, including the processes with higher privileges. Each enclave is encrypted by the CPU and, therefore, the CPU can protect the code from being examined by an adversary. An SGX project typically contains two components, an untrusted component which runs in the untrusted world, as well as a trusted component which runs in the trusted world protected by the processor. Applications running in the untrusted world can invoke pre-defined functions inside the enclave, and the invocations are ECALLs (or enclave calls). The enclave can also call pre-defined functions in the untrusted world, and the invocations are OCALLs (or outside calls).

III. Models and Assumptions

System model. We consider a personal computer (e.g., a desktop, a laptop) which is used to store and to process sensitive data. The processor of the computer is equipped with a trusted execution environment (TEE), e.g., Intel SGX, or AMD SEV/SME/TSME. Note that: 1) equipping TEE itself will not cause any deniability issues as the TEE has been widely deployed in computers [2], and 2) using TEE itself will not cause any deniability issues either, as the TEE has

been used broadly for confidential computing applications [7] (i.e., using TEE does not imply the existence of a deniable storage system). The external storage of the computer can be either a magnetic storage like hard disk drive (HDD) or a flash storage device like a solid state drive (SSD).

Adversarial model. We consider a *computationally bounded* adversary. The adversary is *coercive* [18], i.e., it can capture both the victim user and his/her computer at some point, and force him/her to reveal the sensitive data. The coercion methods can be threatening the victim via legal penalties or performing a rubber-hose cryptanalysis [15]. The adversary is assumed to be able to capture the victim and his/her computer multiple times [14] over time, i.e., a *multi-snapshot* adversary. For example, a user may travel with a laptop but lose the possession of the laptop when crossing the checkpoints of the airports. After capturing a victim computer each time, the adversary can play with the victim computer; in addition, it can obtain snapshots of both the external storage and the internal memory of the victim computer, performing forensic analysis over those snapshots to compromise the deniability; further, the adversary can also compare the snapshots captured at different points of time, aiming to compromise the deniability. The multiple snapshots captured at different points of time indeed can help the adversary to compromise the deniability as justified below:

Case 1: attacking a PDS system based on the hidden volume technique via multiple snapshots. The hidden volume-based PDS system [21], [41], [64] creates two volumes, a public and a hidden volume (Figure 2). The entire disk is filled with random data initially. The hidden volume is hidden at the second half of the disk and remains indifferentiable from the random data. From the view of the adversary, the second half of the disk is empty. If the user updates data in the hidden volume, by comparing the multiple snapshots (e.g., snapshot I and II) captured at different points of time, the adversary will be aware that the "empty" space has been changed which is suspicious (Figure 2). In this way, the deniability of the PDS will be compromised.

Case 2: attacking the PDS system based on steganography. In a steganographic file system-based PDS [8], [50], sensitive data are hidden in the dummy file blocks. Those dummy file blocks should not be modified from the adversary's view. If the user updates sensitive data, the adversary will observe changes in the dummy file blocks by comparing the snapshots captured at different points of times, compromising the deniability.

Assumptions. We rely on a few assumptions: 1) TEE is secure. This assumption is common for TEE-based secure applications [57]. Although various attacks have been found against TEE [45], [46], [53], [61]–[63], [63], [68], enhancing TEE security has been taken care actively by orthogonal works [16], [34], [74] and is not the focus of this work. 2) Data corruption attacks are out of the scope of this work, and we will briefly discuss mitigating strategies in Sec. V-B. 3) At the time when the adversary physically captures the victim device, the user is not processing hidden sensitive data. Otherwise, the adversary can trivially retrieve the sensitive information. This assumption implies that *TEE is not used to process hidden sensitive data when the adversary is accessing the victim device*, i.e., the adversary cannot compromise the deniability by simply observing the running of the TEE or playing with the TEE. 4) We aim to deny the existence of sensitive data and, therefore, the adversary has no prior knowledge on both the existence of the sensitive data and the plaintext of the sensitive data. In other words, a known-plaintext attack [49] on the hidden sensitive data is infeasible under our adversarial model. 5) Similar to the prior works [25], [41], [64], we assume that the capability of HiPDS is widespread and its available itself would not be a red flag. For example, HiPDS is part of a standard operation system distribution, and its existence does not imply the use of a plausibly deniable storage system [25].

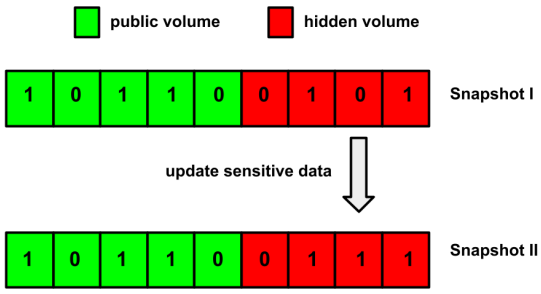

Fig. 2. Attacking a hidden volume-based PDS system via multiple snapshots.

IV. HiPDS: A Storage Hardware-independent Plausibly Deniable Storage System

A. Adapting Chameleon Hash to Plausibly Hiding Information on An External Storage Device

Not all the chameleon hash constructions can be used to hide information, as most of them [10], [11], [17], [72] compress the large input to a condense hash value, rendering the sensitive information hidden in the input irrecoverable. Fortunately, we have identified one chameleon hash construction which may be used for the hiding purpose, namely, the Discrete log-based Chameleon Hash construction (DCH) proposed by Krawczyk et al. [43]. DCH works as follows:

Let p and q be two large prime numbers such that $p = kq + 1$, where k is an integer. Let g be an element of order q in Z_p^*. The private key x can be chosen as a random number from Z_q^*, and the public key y can be computed as $y = g^x \bmod p$. p, q, and g are also part of the public key. Given a message m from Z_q^*, the chameleon hash function is constructed as: $\text{CHAM-HASH}_y(m, r) = g^m y^r \bmod p$, where r is a random number from Z_q^*. It is clear that by knowing the public key y, we can efficiently compute the hash value of any given message $m \in Z_q^*$ for the chosen random number $r \in Z_q^*$. In addition, DCH satisfies three security properties [43] as a chameleon hash:

1) *Uniformity*. All message m induce the same probability distribution on $\text{CHAM-HASH}_y(m, r)$, for r chosen uniformly at random. In particular, by knowing the public key y, the random number r, as well as the hash value $g^m y^r \bmod p$, it is computationally hard to compute m due to the hardness of computing discrete logarithms.

2) *Collision resistance*: Without knowing the private key x, it is computationally infeasible to find a collision message $m' \in Z_q^*$ and a random number $r' \in Z_q^*$ such that CHAM-HASH$_y(m, r)$ = CHAM-HASH$_y(m', r')$. The collision resistance property of DCH (without knowing x) is based on the hardness of computing discrete logarithms.
3) *Trapdoor collisions*: By knowing the private key x, we can efficiently find a collision message $m' \in Z_q^*$ and a random number $r' \in Z_q^*$ such that CHAM-HASH$_y(m, r)$ = CHAM-HASH$_y(m', r')$, by solving the equation $m + xr = m' + xr' \bmod q$.

Mostly, without the knowledge of private key x, the DCH behaves like a regular collision resistant hash function (security property 1 and 2). In addition, the construction of DCH hashing is a pretty common construction based on the hardness of computing discrete logarithms [9], [32], i.e., the format of the construction does not automatically indicate a chameleon hash construction.

Immediately using DCH for plausible information hiding. To hide sensitive data, an immediate approach is to directly use DCH, with private key x and public (decoy) key y, p, q, and g, as follows: Given cover data $m \in Z_q^*$, we pick a random number r from Z_q^*, and compute a hash value h as: h=$g^m y^r \bmod p$. To hide sensitive data $m' \in Z_q^*$, we utilize the private key x and compute the corresponding random number $r' \in Z_q^*$ by solving the equation $m + xr = m' + xr' \bmod q$. We then remove m' and store m, r, h, and r' to the external storage. m' can be computed at any time when needed by solving the equation $m + xr = m' + xr' \bmod q$ using m, r, r' if x is known. When the adversary comes and coerces the victim for the secret, the victim can disclose the decoy key y. The adversary can use y to verify whether or not h is the hash value of the cover data m with random number r. In addition, *the adversary will notice the existence of r' on the external storage and suspect that sensitive data are stored hidden, as the victim cannot plausibly explain the existence of r'*.

Adapting DCH (ADCH) for plausible information hiding. The immediate DCH is insecure due to the existence of r', which is unfortunately needed for recovering the sensitive data m'. Note that r' cannot be computed on the fly, and cannot be stored encrypted as encryption itself is not plausibly deniable. Here m' is hidden as one of the collisions of the cover data m. *For a hash function, the role of the original message and the collision message seem to be exchangeable*. In other words, given the collision message, the original message is also one of its collisions. Based on this observation, we adapt the DCH for plausible information hiding by swapping the cover data and the sensitive data, and the resulted scheme is called ADCH. Specifically, we treat the sensitive data as m from Z_q^*, and generate a random number r from Z_q^* using a pseudo-random function (PRF) with the secret key x. Note that the input to the PRF can be generated on the fly using known information like the file path and the block index. Using m and r, we can compute a hash value h as: h=$g^m y^r \bmod p$. We treat the cover data as m', which is from Z_q^*. Using the private key x, we can compute the random number r' from Z_q^* by solving the equation $m + xr = m' + xr' \bmod q$. We then remove m and store m', r', and h, to the external storage. *Upon being coerced by the adversary*, the victim can simply disclose the public key y and, the adversary uses y to verify that h is the correct hash value of the cover data m' with random number r' (note that CHAM-HASH$_y(m', r')$ is equal to CHAM-HASH$_y(m, r)$). The adversary however will not be able to notice anything abnormal on the external storage device, hence will not be able to identify the existence of the secret key x and the hidden sensitive data m. A side-by-side comparison between DCH and our ADCH is shown in Figure 3. On the contrary, the victim can restore m at any time when needed by solving the equation $m + xr = m' + xr' \bmod q$ by knowing m', r', and x, and *generating r on the fly using a pseudo-random function*.

B. Design Overview

We have designed HiPDS, the first storage Hardware-independent Plausibly Deniable Storage system. To defend against the coercive attack, we encode the sensitive data into the non-sensitive cover data, so that every single read and write of the sensitive data on the external storage device can be plausibly denied as the read and write of the cover data. This turns to be possible by leveraging our newly designed ADCH (Sec. IV-A). Specifically, to **write** the sensitive data to the external storage, we will *encode* them to the cover data using ADCH. The resulted random numbers and hash values, together with the cover data, will be committed to the external storage. To **read** the sensitive data from the external storage, HiPDS will read the corresponding cover data, together with the random numbers and the hash values, and *decode* them to obtain the original sensitive data. By analyzing the external storage, the adversary can obtain the cover data, the random numbers and the hash values. The adversary cannot derive any knowledge about the hidden sensitive data from the cover data. In addition, each hash value can be plausibly denied as the hash value computed over the corresponding cover data and the random number thanks to the nice property of chameleon hash. Upon being coerced, the victim can disclose the public (decoy) key and, using the public key, the adversary can verify whether the hash value is corresponding to the cover data and the random number. Note that hacking [41] into the internal hardware of the external storage will not give the adversary any advantage of compromising the deniability, as the sensitive data remain hidden in the cover data even in the raw disk image.

To prevent the hidden sensitive data from being leaked in the internal memory, both the encode and the decode process will be run in a trusted execution environment (TEE), a secure area of the main processor. Each time upon finishing reading/writing the sensitive data, the TEE (e.g., an SGX enclave and the code running inside) should be completely destroyed so that the adversary cannot have access to a TEE previously running for encoding/decoding hidden sensitive data. Note that the sensitive data and the private key should not be present in the untrusted memory. Therefore, they are recommended to be input from or output to secure I/O devices [47], [56], [65], [69]. Optionally, if they are from the untrusted memory, they

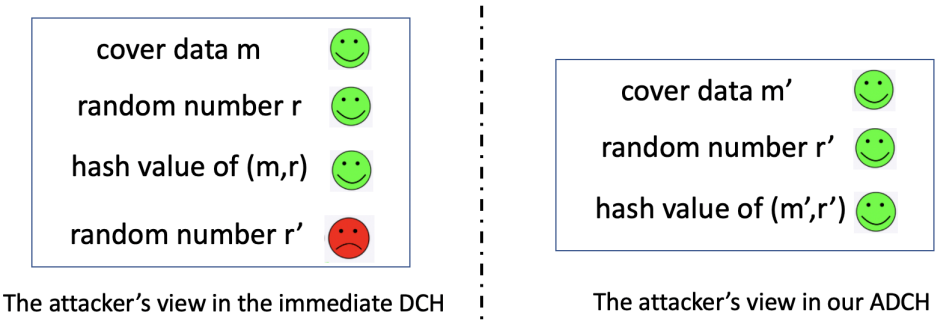

Fig. 3. A comparison between the immediate DCH and our ADCH when the adversary has access to the external storage.

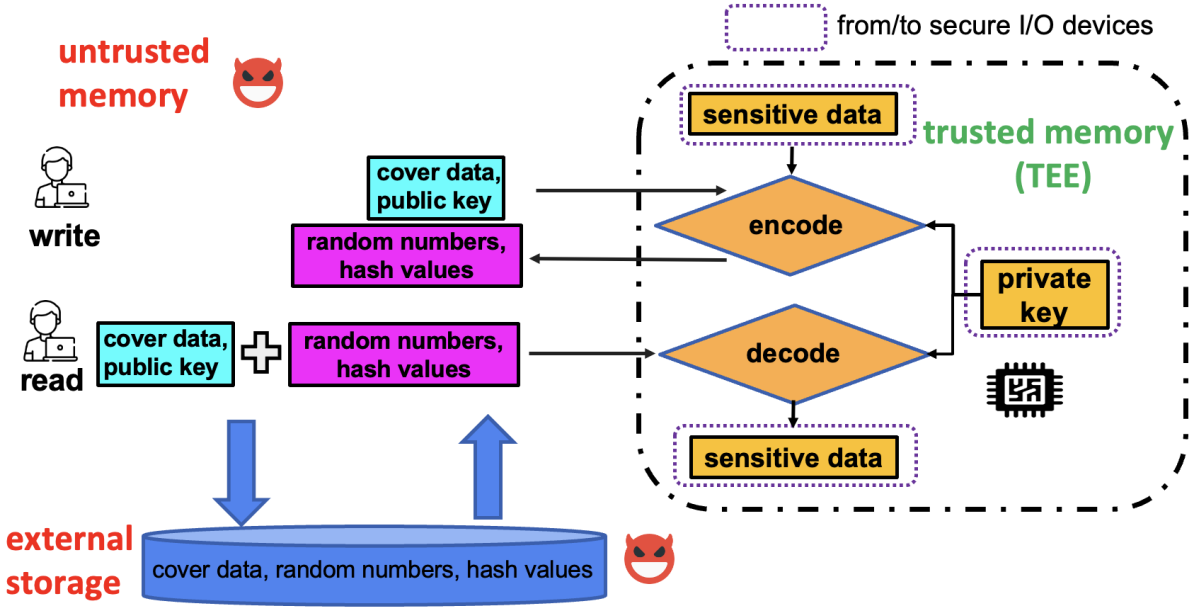

Fig. 4. An overview of HiPDS. Both the untrusted memory and the external storage can be compromised by the adversary, which belong to the *untrusted world*. The trusted memory cannot be compromised by the adversary, belonging to the *trusted world*. The external storage can consist of any type of storage media including hard disk drives and solid state drives.

should be sanitized immediately from the untrusted memory once loaded into the TEE. An overview of our HiPDS design is shown in Figure 4, with details elaborated in Sec. IV-C.

C. Design Details

Setup of HiPDS. κ is a security parameter. We pick two large prime numbers p and q, such that $p = kq + 1$, where k is a positive integer. We also pick an element g of order q from Z_p^*. x is a number randomly selected from Z_q^*, and y is computed as g^x mod p. k' is randomly picked from $\{0,1\}^\kappa$. The public key is (p, q, g, y) while the private key is (x, k'). We also define a pseudo-random function (PRF) f: $\{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^{log_2(q)}$. The public (decoy) key is committed to the external storage of the computing device, while the private (secret) key is only available to the device when reading/writing sensitive data.

Each time to read/write hidden sensitive data, HiPDS will launch a trusted execution environment (TEE) and the private key should be sent to the TEE. The write operation will work with the encoding process running in the TEE, while the read operation will work with the decoding process running in the TEE. After finishing reading/writing the hidden sensitive data, the TEE (as well as the private key) should be destroyed from the device. This is necessary; otherwise, the adversary may be able to "play with" the TEE online to compromise the deniability. Considering reading/writing the sensitive data is a rare event, the overhead introduced by launching/destroying the TEE (e.g., an SGX enclave) would not be significant.

Encode. The encode process is described in Algorithm 1. It will encode a sensitive message m into a cover message m', based on both the public key (p, q, g, y) and the private key (x, k'). It will first generate a random number for the sensitive message m, by applying PRF f with key k over the metadata of the cover message m'. Note that the metadata of the cover message should be unique and stable. For example, if the cover message is a file block, the metadata can be: the host id || file path || the index of the block in the file. In other words, for the same cover message at the same file location, we will

always be able to re-generate the same random number r. After knowing m', m, r, and x, we will be able to determine r'. With m' and r', and y, we will be able to compute the hash value h for (m', r').

Input: cover message m', sensitive message m, private key (x, k'), public key (p, q, g, y)
Output: random number r', hash value h
1. $r = f_{k'}(metadata\ of\ cover\ message\ m')$
2. Determine r' such that $m + xr = m' + xr' \bmod q$
3. $h = g^{m'} y^{r'} \bmod p$
return r', h

Algorithm 1: Encode

Decode. The decode process is described in Algorithm 2. This process will decode the sensitive message m from the cover message m', based on the public key (p, q, g, y), the private key (x, k'), and the random number r' associating with m'. It will first re-generate the random number for the sensitive message m, by applying f with key k over the metadata of the cover message m'. By knowing m', r', r, and x, we can then easily compute the sensitive message m.

Input: cover message m', random number r', private key (x, k'), public key (p, q, g, y)
Output: sensitive message m
1. $r = f_{k'}(meta\ data\ of\ cover\ message\ m')$
2. $m = m' - xr + xr' \bmod q$
return m

Algorithm 2: Decode

Write hidden sensitive data. The write operation allows the user to write a sensitive file to the external storage. The sensitive file is first loaded into the TEE. To process the sensitive file, we view it as a collection of file blocks, each of which is n-byte. This implies that our design can support fine-grained access of the hidden sensitive data in a deniable manner. To encode this sensitive file, a cover file needs to be read from the untrusted world outside the TEE. Note that ideally, the cover file should be of the same size of the sensitive file (refer to Sec. V-B for handling the case in which the cover file has a different size than the sensitive file). The cover file is also viewed as a collection of n-byte file blocks. HiPDS will read the first block from the cover file into the TEE, encoding the first block of the sensitive file into the cover block using the Encode algorithm. The resulted random number and hash value will be returned to the untrusted world. The aforementioned process will be repeated until all the blocks of the sensitive file have been encoded. We can buffer multiple blocks from the cover file and send them as a batch to the TEE for processing. Finally, the entire cover file, the collection of random values, as well as the collection of the hash values will be committed to the external storage.

Read hidden sensitive data. The read operation allows the user to read a sensitive file from the external storage. Similarly, the cover file is viewed as a collection of n-byte blocks. HiPDS will read each block from the cover file into the TEE, together with its corresponding random number and the hash value, and decode the cover block using the Decode algorithm, obtaining the sensitive file block. Note that we can also buffer multiple blocks from the cover file and send them as a batch to the TEE for processing.

D. Optimizing The Encoding Process

There are a few steps in the encoding process (Algorithm 1) which are time-consuming, and we optimize them as follows. **Efficiently computing the random number** r'. In Step 2 of encoding (Algorithm 1), we need to determine a suitable random number r'. This could be achieved using a brute-force searching by initializing r' with 0, and increasing it by 1 until Equation 1 holds.

$$m + xr = m' + xr'\ mod\ q \tag{1}$$

This brute-force searching has a computational complexity of $O(q)$, which is extremely expensive as q is a large prime number (e.g., 128 bits or larger). To efficiently compute r', we first derive Equation 2 from the Equation 1:

$$\begin{gathered} xr' + Nq = m + xr - m' \\ where\ r' \in Z_q^*\ and\ N\ is\ an\ integer. \end{gathered} \tag{2}$$

Equation 2 is a linear equation with two unknowns r' and N. To solve this linear equation, we need to first check whether it has integer solutions. Equation 2 has integer solutions if and only if the greatest common divisor of x and q (denoted as $gcd(x, q)$) can be divided by $m + xr - m'$ [28], [58]. As q is a large prime number and x is from Z_q^*, $gcd(x, q)$ is 1, which is always divisible by $m + xr - m'$. We therefore can confirm that Equation 2 always has integer solutions. To efficiently find out a pair of (r', N) which can satisfy this equitation, we can leverage Extended Euclidean Algorithm [13] as follows: We first find a solution of Equation 3 via Extended Euclidean Algorithm, and the solution is assumed as (r'_0, N_0). Then, the Equation 3 can be converted to Equation 4, which is equivalent to Equation 5. As $gcd(x, q)$ is 1, we can further get Equation 6. By comparing Equation 2 and 6, we can infer that $(r'_0(m+xr-m'), N_0(m+xr-m'))$ is a solution of Equation 2. Therefore, r' can be computed as $(r'_0 \cdot (m + xr - m'))\ mod\ q$. In other words, we only need to calculate r'_0 once during the initialization, and for each encoding, we can compute r' efficiently in $O(1)$ time, instead of $O(q)$ in the brute-force searching. In addition, the solution for the unknowns r' and N in Equation 2 can be further generalized in Equation 7, where t is an integer. Therefore, r' can be an arbitrary value which satisfies the condition in Equation 7, rather than a fixed value.

$$xr' + Nq = gcd(x, q) \tag{3}$$

$$xr'_0 + N_0 q = gcd(x, q) \tag{4}$$

$$\frac{xr'_0(m + xr - m')}{gcd(x, q)} + \frac{N_0 q(m + xr - m')}{gcd(x, q)} = m + xr - m' \tag{5}$$

$$x(r'_0(m+xr-m')) + (N_0(m+xr-m'))q = m+xr-m' \tag{6}$$

$$\begin{aligned} r' &= r'_0(m + xr - m') + q * t \\ N &= N_0(m + xr - m') - x * t \end{aligned} \tag{7}$$

V. Analysis and Discussion

A. Security Analysis

We consider an adversary (Sec. III) which can coerce the victim for the secrets at multiple checkpoints (i.e., multi-snapshot coercive adversary). In the following, we show that HiPDS is secure against the multi-snapshot adversary which can have access to the external storage and the memory multiple times over time. Note that the "cover data" refer to both cover files and data blocks in a cover file in HiPDS.

Plausibly denying the existence of sensitive data in the external storage. The only relation between the cover data and the sensitive data is, the sensitive data is one of the collisions of the cover data in the hash function. However, without knowing the private key, it is computationally infeasible for the adversary to find out the collisions as the discrete log-based chameleon hash function (DCH) is collision resistant (Sec. IV-A). In addition, without knowing the private key, the DCH is similar to a regular keyed hash function (Sec. IV-A) and, the adversary is not able to confirm that the hash function used in HiPDS is a chameleon hash function (i.e., the existence of this hash function itself does not raise a red flag). Upon being coerced, the victim only discloses the public key and, using the public key, the adversary can verify the key is really used to compute the hash value for the cover data (together with its corresponding random number). It cannot find out the existence of the private key, hence the existence of the hidden sensitive data. Therefore, HiPDS can mitigate the *coercive adversary* which can have access to the external storage. Having access to the external storage at multiple checkpoints does not offer the adversary any advantage in compromising the deniability as every read/write of the sensitive data on the external storage device can be plausibly explained as reading/writing the cover data.

Plausibly denying sensitive data in the memory. Sensitive data are only processed in the secure memory region protected at the hardware level by the TEE. This implies that even if the adversary can obtain the root privilege, it cannot hack into this secure memory region and hence cannot obtain any information (as well as the existence) of the sensitive data by performing forensic analysis over the memory if the TEE is secure. The existence of such a secure memory region can be plausibly explained, as the TEE can be used for any confidential computing tasks like secure payment and does not indicate the existence of a deniable storage system (Sec. III). The adversary cannot capture the victim when he/she is right processing the hidden sensitive data (Sec. III). Since the TEE is destroyed immediately each time after finishing processing the sensitive data, the adversary will not be able to capture a TEE which is right processing the hidden sensitive data, and hence cannot "play with" the TEE online in order to compromise the deniability. Having access to the memory multiple times at different checkpoints will not provide the adversary any advantage of compromising the deniability because: the adversary can at most find out the secure memory region changes which can be plausibly explained as running a different confidential computing task in the TEE before the checkpoint.

B. Discussion

About the cover data. To ensure plausible deniability, the cover data should be *meaningful*. In other words, in view of the adversary, the cover data are regular data which are processed by the user using the TEE due to some security concerns. We can obtain this type of "real" cover files from public search engines, public repository, or private non-sensitive files discarded by the user. For instance, the user can generate hundreds of English words, and use each English word to query the mainstream search engines (e.g., google, bing), collecting texts from the search results. The texts can be combined further to generate the file content [42]. In addition, to hide a sensitive file, we need to use a cover file which is at least of the same size as the sensitive file. If the size of the cover file is larger than the sensitive file, we can use the first block of the cover file to keep track of those cover file blocks encoded with the sensitive data. For example, if we plan to use cover file block 2 to 11, we can encode 10 into cover file block 1 via chameleon hash; upon decoding, after having decoded cover file block 1, we can obtain 10, and know that cover file block 2 to 11 should be decoded. To ensure plausible deniability, we should also process the remaining file blocks in the cover file which do not have sensitive data encoded, generating their hash values upon writing using arbitrary random numbers. Lastly, to read/delete sensitive data, the user needs to first locate the corresponding cover file. Memorizing the mappings between the sensitive files and the cover files could be difficult for the user. A strategy would be carefully naming the cover file so that it can facilitate memorization without being alert by the adversary.

Updating the sensitive data. To update the sensitive data, we can read the corresponding cover file into the TEE, decode it and obtain the sensitive file. We then update the sensitive file. Re-encoding the updated sensitive file to the original cover file may be problematic, because: 1) If the size of the sensitive file grows, the original cover file may not be large enough to encode the sensitive file again. 2) Re-encoding the updated sensitive file into the same exact cover file implies that the corresponding random numbers, all or partially, may need to be changed. This would be aware by the adversary. A better solution would be reading a new cover file and encoding the updated sensitive file to it; in addition, the old cover file should be deleted. This solution would be fine if the sensitive file has been changed significantly, but it would not be efficient if only a few file blocks of the sensitive file are updated. For this case, we can re-encode the updated file blocks of the sensitive file to the original cover file, by updating the corresponding blocks in the cover file, as well as the corresponding random numbers and hash values, claiming that the cover file has been updated and hence some random numbers/hash values need to be updated as well.

About associating a key and a hash with every file block. Associating a key and a hash with every data block is not unique in our design. This allows efficiently checking integrity of the file, e.g., if the file is large, the verifier can only check a random subset of file blocks and detect whether the file has been corrupted or not with high probability. This has been

used broadly in the storage systems [9], [32]. Therefore, this is deniable.

About protecting the code of HiPDS. The presence of a PDS system in the software stack does not directly indicate that the user is hiding information [25], [41], [64], considering that the PDS system itself may be widespread and becomes a piece of standard software in the OS [25], [64]. If the code itself is really a concern for the user, we recommend the user to load HiPDS each time when processing hidden sensitive data, and to remove it after finishing the processing. The binary as well as the configuration files of HiPDS are only a few Megabytes in size, which would not create a significant burden on the user considering that reading/writing hidden sensitive data is typically a rare event.

Managing the private key. Each time when processing hidden sensitive data, the private key (x, k') needs to be provided by the user. Note that the private key should be loaded into the TEE securely via two options: 1) through a secret I/O device, or 2) through an insure I/O device, but its traces need to be sanitized in the untrusted memory after it is loaded. Memorizing a long key would be difficult for the user, and one improvement would be leveraging PBKDF2 (Password Based Key Derivation Function 2 [52]) so that the user can generate the key on the fly using his/her secret password [64].

About secure I/O devices. Upon processing hidden sensitive data, a secure I/O device is recommended to load the sensitive data/private key to the TEE, or to obtain output from the TEE. These include a secure keyboard [56], [65], a secure network interface card [56], etc. Since we assume that the adversary cannot capture the victim when processing hidden sensitive data (Sec. III), the adversary will not be able to capture the victim when using the secure I/O devices for PDS. In addition, the existence of the secure I/O devices itself will not cause any deniability issues as they are not necessarily used for deniable storage system. Note that the secure I/O devices are not mandatory components of HiPDS. Users optionally can use regular I/O devices, but need to ensure that traces of hidden sensitive data/private key should be sanitized upon completion of the processing of the sensitive data.

Mitigating data corruption attacks. HiPDS cannot resist against the data corruption attacks, as the adversary can easily destroy all the user data upon capturing the user device. A suggestion would be to regularly make copies of the sensitive data on different computer devices as a precautionary measure.

Implementing HiPDS **under different TEEs or no TEE**. The purpose of using TEE is to isolate the processing of the hidden sensitive data in the memory, so that the adversary is not able to compromise this process by analyzing the memory. In the following, we discuss two cases: 1) The TEE is available in the victim device. The TEE has been available in main-stream processors. Although Intel has deprecated SGX in the 11th and 12th generation of its core desktop processors, it continues to support SGX in its Xeon processors [6]. In addition, AMD supports SEV/SME/TSME in its Ryzen, RyzenPro and EPYC processors [1], [4]. HiPDS can be broadly deployed on computing devices using the aforementioned processors. 2) The TEE is not available in the victim device. In this case, other secure hardware technologies, e.g., the trusted execution technology (TXT) [40] which enables the provision of an isolated execution environment, can be leveraged. Additionally, software-based trusted execution environments [38], [44] can be also utilized. They may not offer the same hardware-level security guarantees as trusted hardware, but they can provide a certain degree of protection for sensitive operations.

VI. Implementation and Evaluation

A. Implementation

We have developed a prototype of HiPDS. The TEE used in our experiments is SGX enclave. OpenSSL-1.1.1m [5] was integrated into SGX to facilitate our development. We chose κ as 160, and initiated the PRF f using HMAC-SHA1 with a 160-bit random key k'. The output of the HMAC-SHA1 was tuned to ensure that the generated random number r is from Z_q^*.

The prime numbers p and q must satisfy $p = kq+1$, where k is a positive integer. Finding out suitable large prime numbers p and q which can satisfy this condition is not straightforward. For simplicity, we fixed k as 2. This allows us to utilize the OpenSSL API $BN\_generate\_prime\_ex$ to generate a safe large prime number p such that $(p-1)/2$ is also prime, i.e., $q = (p-1)/2$ is guaranteed as prime. We will discuss how the changes of the k value might affect the performance (Sec. VI-B). We also determined a suitable g value which is an element of order q in Z_p^*. In our experiments, we fixed g as 2 for simplicity. We will also discuss how the changes of g might effect the performance (Sec. VI-B). The private key x is a number randomly selected from Z_q^*, and the public key y is computed as $g^x \bmod p$. To ensure each file block is from Z_q^*, the size of each file block is always 1 bit less than $log_2(q)$.

HiPDS supports two operations on sensitive data, namely, write and read. We created two ECALL functions in the SGX enclave for the write and the read operation. Upon writing sensitive data, the cover file is divided into a collection of blocks and sent to the enclave through ECALL for encoding the sensitive data. After the encoding is finished, the enclave will return hash values and random numbers for the corresponding cover file blocks to the untrusted world. Upon reading sensitive data, the cover file is viewed as a collection of blocks, which will be entirely or partially read from the external storage (together with the corresponding random numbers and hash values), and sent to the enclave via ECALL for decoding to obtain the sensitive data.

Experimental setup. We mainly ran our prototype in a local Lenovo ThinkPad P50 laptop equipped with Intel Core i7-6700HQ processors (2 Quad-core processors, totally 8 cores), 8GB memory, 500GB HDD and Ubuntu 18.04.6 LTS operating system. The processors support SGX. The computational time is averaged over 5 runs. When encoding/decoding a sensitive file, we batched a few file blocks before invoking the ECALL, with buffer size 5MB.

B. Evaluation

Evaluating the setup. We first evaluated the time needed for setting up HiPDS. As we have fixed k and g as 2, we can

q	129-bit	193-bit	257-bit
Time (s)	0.001289	0.001313	0.001366

TABLE I
THE TIME NEEDED TO SET UP HiPDS WHEN THE SIZE OF FINITE FIELD VARIES.

	without optimization	with optimization
time (s)	3.34	0.000021

TABLE II
THE TIME NEEDED FOR ENCODING ONE DATA BLOCK W/O THE OPTIMIZATION (AVERAGED FROM THE ACCUMULATED TIME NEEDED TO PROCESS MULTIPLE DATA BLOCKS UNDER THE SAME SETTING).

vary q (hence p is varied) which determines the field size of Z_q^*. The time needed to set up HiPDS is shown in Table I. We can observe that a larger q will result in slightly larger setup time, which is reasonable as determining the parameters in a larger field typically takes more computation.

Performance of reading/writing sensitive data. The major overhead for reading/writing sensitive data comes from decoding/encoding the data in the enclave. Therefore, we evaluated the time needed for encoding/decoding a sensitive file when the file size varies, e.g., 100MB, 300MB, 500MB, and 700MB. During the experiments, we fully utilized the multiple cores of the processors equipped with the laptop, by launching 8 SGX enclaves simultaneously [31], which work together to process a given file in parallel, such that each enclave processes an approximately equal portion of file blocks. The experimental results are shown in Figure 5, Figure 6 and Figure 7 for different q values. We can observe that: 1) A larger size of block size has a lower computational time. This is because the major computation cost comes from the calculation of the hash value and generating random number for each data block. A smaller size of data block will have a more data blocks, which will need more time to process. 2) The computational time needed for encoding/decoding a sensitive file increases linearly when the file size increases. This is because, a larger file consists of more file blocks, which requires more processing time. 3) The computational time needed for decoding a file is much less than that for encoding the file. The reason is, the encoding process involves computing $g^{m'}$ in Z_p^*, which is more expensive in the large prime field. 4) The throughput for encoding is approximately 1.48MB/s, while the throughput for decoding is approximately 10.02MB/s. The normal disk throughout in our laptop is 100+MB/s (measured via benchmark tool Bonnie++). The throughout slowdown is significant, but the throughput is certainly acceptable in practice, especially considering that the hidden sensitive data is typically small in size [22] and, accessing the hidden sensitive data is an infrequent event.

Performance comparison before and after the optimization. We experimentally justified the effectiveness of our optimization in encoding. We collected the time needed to encode each data block (32 bytes) w/o using our optimization technique. The result is shown in Table II, which confirms that, 1) determining a suitable random number r' via a brute-force searching in a large field is very slow, and 2) our optimization can significantly reduce the time needed for determining r' by eliminating the brute-force searching, hence significantly reducing the time needed for encoding a block.

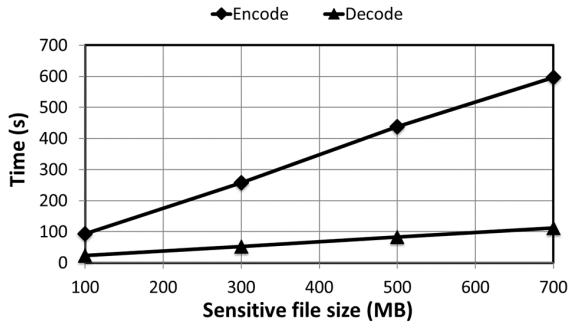

Fig. 5. The time needed for encoding/decoding a sensitive file in the SGX enclave (q = 129 bits).

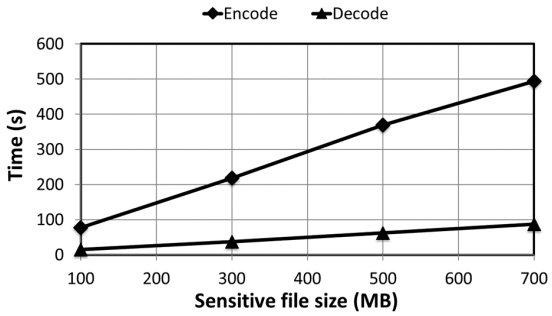

Fig. 6. The time needed for encoding/decoding a sensitive file in the SGX enclave (q = 193 bits).

Comparing HiPDS with other PDS systems. We compared HiPDS with other PDS systems (HIVE [14], DataLair [19], PD-DM [23], and PEARL [25]) which can defend against a multi-snapshot adversary. The comparison is shown in Table III. We can observe that: 1) HiPDS does not affect the I/O throughout on the public non-sensitive data. On the contrary, the other PDS systems significantly affect the I/O throughput on the public non-sensitive data (varying from 40% to 99% decrease in the throughput). 2) HiPDS significantly decreases the I/O throughput on the hidden sensitive data. However, such a degradation is not unique for HiPDS, as the

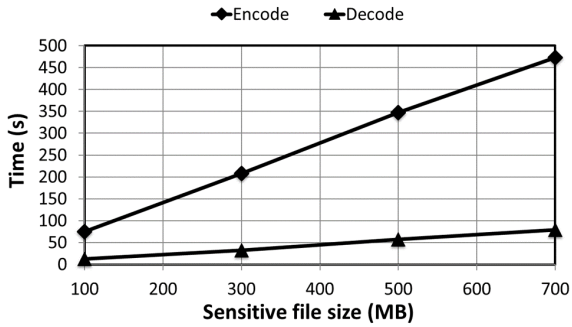

Fig. 7. The time needed for encoding/decoding a sensitive file in the SGX enclave (q = 257 bits).

	Technique	Public data	Hidden data
HIVE [14]	ORAM	0.44%	0.44%
DataLair [19]	ORAM	0.95% - 37.24%	1.38% - 2.66%
PD-DM [23]	Dummy write	1.53% - 17.93% (HDD) 1.28% - 44.21% (SSD)	1.65% - 12.33% (HDD) 1.42% - 43.11% (SSD)
PEARL [25]	WOM code	60.00%	10.00% - 20.00%
HiPDS	Chameleon hash	100.00%	2.00% - 9.12%

entry

TABLE III

PERFORMANCE COMPARISON AMONG PDS SYSTEMS AGAINST MULTI-SNAPSHOT ADVERSARIES. FOR PUBLIC/HIDDEN DATA, EACH ENTRY DESCRIBES A RATIO BETWEEN THE THROUGHPUT OF THE PDS SYSTEM AND THE THROUGHPUT OF A NORMAL SYSTEM WITHOUT DENIABILITY.

	storage required
steganographic file system [8], [50]	approximately 3×
hidden volume [21], [41], [64]	approximately 2×
ORAM [14], [19]	at least 2×
WOM code [25]	approximately 5×
HiPDS (our design)	approximately 3×

TABLE IV

THE STORAGE OVERHEAD FOR DIFFERENT PDS TECHNIQUES. NOTE THAT THE VALUE IS NOT EXACT AS IT HAS BEEN ESTIMATED BASED ON SOME TYPICAL CASES.

other PDS systems also suffer from such a degradation. This performance degradation mostly comes from the additional operations needed to hide the access on the hidden sensitive data in order to mitigate the strong multi-snapshot adversary. Although HiPDS cannot improve the I/O throughout on the hidden data, it is still advantageous, as it is storage hardware-independent, and does not affect the I/O throughput on the public data.

The storage overhead for hiding sensitive data. To plausibly deny the existence of sensitive data, we need to store the cover data, the random number for the cover data, as well as the hash value. In other words, to store a sensitive file with size $|S|$, the storage is $3|S|$. We argue that the high storage overhead for a PDS system is pretty common due to the nature of plausible hiding (evidence can be found in Table IV). For example, the steganographic file system [8], [50] needs to maintain multiple replicas of the sensitive data to mitigate data lost, e.g., to achieve 0.005% data loss rate, 3 copies of the sensitive data need to be maintained in the disk; the hidden volume technique [21], [41], [64] needs to fill the entire disk with randomness, and only half of the storage can be used to stored sensitive data; the WOM code-based technique [25] needs to use 5 bits to hide 1 bit when using (3,5) WOM code; the ORAM-based solutions [14], [19] require at least 100% extra storage as half of the disk is required to be free [23].

The impact of k and g on the performance of HiPDS. The aforementioned experimental results were obtained by fixing both the k and g as 2. When k is increased, for fixed q, p will be larger. In other words, the hash value will be computed in a larger field, and the computation for the encoding would be increased. However, the computation required for the decoding will not be effected, as q is fixed. When g is increased, for fixed q and p, the computation required for computing the hash value will be larger, hence the computation for the encoding would be larger. However, the computation required for the decoding would not be affected, as q is fixed.

VII. RELATED WORK

Steganographic file system-based PDSes. Steganographic file systems [8], [50] hided the secret data among the randomness initially filled in the entire disk. To avoid data loss, a few copies of the hidden sensitive data should be be maintained across the disk. Such a PDS design can only defend against a single-snapshot adversary, as the multi-snapshot adversary can compare different snapshots and identify the update on the sensitive data hidden in the randomness.

Hidden volume-based PDSes. TrueCrypt [67] and VeraCrypt [29] introduced a hidden volume which can be used to stored hidden sensitive data. The hidden volume is encrypted using a true key and placed to the end of the disk which has been filled entirely with random data initially. Both TrueCrypt and the VeraCrypt were designed for HDDs and can only defend against the single-snapshot adversary. However, they are vulnerable to the multi-snapshot adversary who can compare snapshots and be aware of the data dynamics on the hidden volume. Mobiflage [64], MobiHydra [73], Mobipluto [20], [21] extended the hidden volume technique to Android devices. However, they are deployed on the block layer and may suffer from the single-snapshot adversary who can have access to underlying flash memory. DEFTL [41] fixed the aforementioned deniability compromises by incorporating the hidden volume into the flash translation layer, eliminating any low-layer traces which may cause deniability compromises. CrossPDE [27] extends DEFTL by separating PDS functionality and placing them in different layers of a storage system to achieve security, efficiency and usability. Both the DEFTL and the CrossPDE can defend against the single-snapshot adversary for mobile devices. Still, they cannot defend against the multi-snapshot adversary due to the use of the hidden volume technique.

ORAM-based PDSs. To obfuscate the writes of the hidden sensitive data, HIVE [14] and DataLair [19] incorporated write-only oblivious random access machine (ORAM). They can defend against the multi-snapshot adversary as they hide the pattern of access when writing the hidden sensitive data. PD-DM [23] improved [14], [19] by replacing randomization of ORAM via a canonical form which allows most of writes to be performed sequentially.

Flash hardware-based PDSes. PEARL [25] leveraged the write-once memory (WOM) code to hide sensitive data in the flash memory. It can defend against the multi-snapshot adversary by encoding hidden bits in the same physical locations as the public bits, utilizing the write-once memory which fits the nature of flash memory. Another work, INFUSE [24], attempted to defend against the multi-snapshot adversary by hiding the sensitive data into the hardware side-channel of flash memory. INFUSE requires that the flash memory hardware has the ability to program and operate the same cell as an SLC or an MLC, which could significantly limit its applications.

Other PDSes. All the aforementioned PDSes only concern on the deniability compromises in the external storage. The PDS designed by Liao et al. [48] is the sole one which concerns on the deniability compromises in the internal memory using the

TEE. However, their work is specifically designed for mobile devices which use flash memory as external storage and is therefore not storage hardware independent. In addition, it is unclear how the TrustZone secure world can have direct access to the external storage in their design and, going through the untrusted OS to access the external storage still suffers from deniability compromises.

VIII. Conclusion

In this work, we have designed a storage hardware-independent plausibly deniable storage system to defend against a multi-snapshot adversary. By leveraging chameleon hash, HiPDS can plausibly hide the sensitive data via the non-sensitive cover data. In this way, every access of hidden sensitive data can be plausibly denied as that of the cover data. TEE is also leveraged to plausibly hide the sensitive data in the memory. Security analysis and experimental evaluation show that HiPDS can ensure plausible deniability with an acceptable computational overhead.

References

[1] AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/.

[2] GitHub - ayeks/SGX-hardware: This is a list of hardware which supports Intel SGX - Software Guard Extensions. https://github.com/ayeks/SGX-hardware.

[3] HiPDS: A Storage Hardware-Independent Plausibly Deniable Storage System. https://ieeexplore.ieee.org/abstract/document/10336827.

[4] Memory encryption: AMD SME, TSME and SEV. https://mricher.fr/post/amd-memory-encryption/.

[5] Openssl 1.1.1 series release notes. https://www.openssl.org/news/openssl-1.1.1-notes.html.

[6] Rising to the Challenge - Data Security with Intel Confidential Computing. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141.

[7] What is Confidential Computing? — TechRepublic. https://www.techrepublic.com/article/confidential-computing/.

[8] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *International Workshop on Information Hiding*, pages 73–82. Springer, 1998.

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM CCS*, pages 598–609, 2007.

[10] Giuseppe Ateniese and Breno de Medeiros. Identity-based chameleon hash and applications. In *International Conference on Financial Cryptography*, pages 164–180. Springer, 2004.

[11] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton Andrade. Redactable blockchain–or–rewriting history in bitcoin and friends. In *2017 IEEE European symposium on security and privacy*, pages 111–126. IEEE, 2017.

[12] Austen Barker, Staunton Sample, Yash Gupta, Anastasia McTaggart, Ethan L Miller, and Darrell DE Long. Artifice: A deniable steganographic file system. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*, 2019.

[13] Matt Bishop. Introduction to computer security. 2005.

[14] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM CCS*, pages 203–214. ACM, 2014.

[15] Hristo Bojinov, Daniel Sanchez, Paul Reber, Dan Boneh, and Patrick Lincoln. Neuroscience meets cryptography: designing crypto primitives secure against rubber hose attacks. In *21st USENIX Security Symposium*, pages 129–141, 2012.

[16] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Proceedings of 2019 ACSAC*, pages 788–800, 2019.

[17] Jan Camenisch, David Derler, Stephan Krenn, Henrich C Pöhls, Kai Samelin, and Daniel Slamanig. Chameleon-hashes with ephemeral trapdoors. In *IACR International Workshop on Public Key Cryptography*, pages 152–182. Springer, 2017.

[18] Rein Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Proceedings of CRYPTO*, pages 90–104. Springer, 1997.

[19] Anrin Chakraborti, Chen Chen, and Radu Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *Proceedings on Privacy Enhancing Technologies*, 3:175–193, 2017.

[20] Bing Chang, Yao Cheng, Bo Chen, Fengwei Zhang, Wen-Tao Zhu, Yingjiu Li, and Zhan Wang. User-friendly deniable storage for mobile devices. *computers & security*, 72:163–174, 2018.

[21] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 381–390. ACM, 2015.

[22] Bing Chang, Fengwei Zhang, Bo Chen, Yingjiu Li, Wen-Tao Zhu, Yangguang Tian, Zhan Wang, and Albert Ching. Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In *Proceedings of DSN*, pages 454–465. IEEE, 2018.

[23] Chen Chen, Anrin Chakraborti, and Radu Sion. Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. *Proc. Priv. Enhancing Technol.*, 2019(1):153–171, 2019.

[24] Chen Chen, Anrin Chakraborti, and Radu Sion. Infuse: Invisible plausibly-deniable file system for nand flash. *Proc. Priv. Enhancing Technol.*, 2020(4):239–254, 2020.

[25] Chen Chen, Anrin Chakraborti, and Radu Sion. {PEARL}: Plausibly deniable flash translation layer using {WOM} coding. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[26] Niusen Chen, Bo Chen, and Weisong Shi. The block-based mobile pde systems are not secure-experimental attacks. In *Applied Cryptography in Computer and Communications: Second EAI International Conference, AC3 2022, Virtual Event, May 14-15, 2022, Proceedings*, pages 139–152. Springer, 2022.

[27] Niusen Chen, Bo Chen, and Weisong Shi. A cross-layer plausibly deniable encryption system for mobile devices. In *Security and Privacy in Communication Networks: 18th EAI International Conference, SecureComm 2022, Virtual Event, October 2022, Proceedings*, pages 150–169. Springer, 2023.

[28] Tsu-Wu J Chou and George E Collins. Algorithms for the solution of systems of linear diophantine equations. *SIAM Journal on computing*, 11(4):687–708, 1982.

[29] CodePlex. Veracrypt ssd. https://veracrypt.codeplex.com/, 2017.

[30] United States Congress. Health Insurance Portability and Accountability Act. Retrieved March 18, 2023, from http://www.hhs.gov/ocr/privacy/index.html, 1996.

[31] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[32] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):1–29, 2015.

[33] Wendi Feng, Chuanchang Liu, Zehua Guo, Thar Baker, Gang Wang, Meng Wang, Bo Cheng, and Junliang Chen. Mobigyges: A mobile hidden volume for preventing data loss, improving storage utilization, and avoiding device reboot. *Future Generation Computer Systems*, 2020.

[34] Lukas Giner, Andreas Kogler, Claudio Canella, Michael Schwarz, and Daniel Gruss. Repurposing segmentation as a practical lvi-null mitigation in sgx. In *USENIX Security Symposium*, 2022.

[35] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, pages 77–89, 1996.

[36] Peter Gutmann. Data remanence in semiconductor devices. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[37] Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 317–326, 2010.

[38] Seung-Kyun Han and Jinsoo Jang. Mytee: Own the trusted execution environment on embedded devices. In *NDSS*, 2023.

[39] Shuangxi Hong, Chuanchang Liu, Bingfei Ren, Yuze Huang, and Junliang Chen. Personal privacy protection framework based on hidden technology for smartphones. *IEEE Access*, 5:6515–6526, 2017.

[40] Intel. Intel trusted execution technology. https://i.dell.com/sites/content/business/smb/en/Documents/Trusted-Execution-Technology.pdf.

[41] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 24th ACM conference on Computer and communications security*. ACM, 2017.

[42] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. {UNVEIL}: A {Large-Scale}, automated approach to detecting ransomware. In *25th USENIX security symposium (USENIX Security 16)*, pages 757–772, 2016.

[43] Hugo Krawczyk and Tal Rabin. Chameleon hashing and signatures. 1998.

[44] Unsung Lee and Chanik Park. Softee: Software-based trusted execution environment for user applications. *IEEE Access*, 8:121874–121888, 2020.

[45] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking" security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2937–2950, 2021.

[46] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. {CIPHERLEAKS}: Breaking constant-time cryptography on {AMD}{SEV} via the ciphertext side channel. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 717–732, 2021.

[47] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. Establishing trusted i/o paths for sgx client systems with aurora. *IEEE Transactions on Information Forensics and Security*, 15:1589–1600, 2019.

[48] Jinghui Liao, Bo Chen, and Weisong Shi. Trustzone enhanced plausibly deniable encryption system for mobile devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 441–447. IEEE, 2021.

[49] Mitsuru Matsui and Atsuhiro Yamagishi. A new method for known plaintext attack of feal cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 81–91. Springer, 1992.

[50] Andrew D McDonald and Markus G Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 463–477. Springer, 2000.

[51] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. X-search: revisiting private web search using intel sgx. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 198–208, 2017.

[52] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. Pkcs# 5: Password-based cryptography specification version 2.1. Technical report, 2017.

[53] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE S&P*, pages 1466–1482. IEEE, 2020.

[54] HweeHwa Pang, K-L Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Proceedings of 2003 ICDE*, pages 657–667. IEEE, 2003.

[55] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *NDSS*, 2015.

[56] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. Bastion-sgx: Bluetooth and architectural support for trusted i/o on sgx. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, 2018.

[57] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick PC Lee, and Xiaosong Zhang. Accelerating encrypted deduplication via {SGX}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 957–971, 2021.

[58] UN Roy, RP Sah, AK Sah, and SK Sourabh. Linear diophantine equation: Solution and applications. *International Journal of Mathematics Trends and Technology (IJMTT)*, 65(1), 2019.

[59] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[60] U.S. Senator Paul Sarbanes and U.S. Representative Michael G. Oxley. Sarbanes-Oxley Act. Retrieved March 18, 2023, from https://www.govinfo.gov/content/pkg/COMPS-1883/pdf/COMPS-1883.pdf, 2002.

[61] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel sgx. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 177–196. Springer, 2019.

[62] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[63] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 318–331. IEEE, 2019.

[64] Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27*, 2013.

[65] Florin-Alexandru Stancu, Dumitru Cristian Trancă, and Mihai Chiroiu. Tio-secure input/output for intel sgx enclaves. In *2019 International Workshop on Secure Internet of Things (SIOT)*, pages 1–9. IEEE, 2019.

[66] Ying Sun, Jiwen Chai, Huihui Liang, Jianbing Ni, and Yong Yu. A secure and efficient e-cheque protocol from chameleon hash function. In *5th International Conference on Intelligent Networking and Collaborative Systems*. IEEE, 2013.

[67] TrueCrypt. Free open source on-the-fly disk encryption software.version 7.1a. *Project website: http://www.truecrypt.org/*, 2012.

[68] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

[69] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In *Proceedings of the seventh ACM on conference on data and application security and privacy*, pages 261–268, 2017.

[70] Yankai Xie, Chi Zhang, Lingbo Wei, Qingtao Wang, and Zhe Yang. A secure and efficient bitcoin payment channel using intel sgx. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[71] Junpeng Xu, Haixia Chen, Xu Yang, Wei Wu, and Yongcheng Song. Verifiable image revision from chameleon hashes. *Cybersecurity*, 4(1):1–13, 2021.

[72] Shengmin Xu, Jianting Ning, Jinhua Ma, Guowen Xu, Jiaming Yuan, and Robert H Deng. Revocable policy-based chameleon hash. In *European Symposium on Research in Computer Security*, pages 327–347. Springer, 2021.

[73] Xingjie Yu, Bo Chen, Zhan Wang, Bing Chang, Wen Tao Zhu, and Jiwu Jing. Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *Proceedings of ISC 2014*, pages 555–567, 2014.

[74] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.