Sloth: Key Stretching and Deniable Encryption using Secure Elements on Smartphones

Daniel Hugenroth
University of Cambridge

Alberto Sonnino
Mysten Labs
University College London

Sam Cutler
The Guardian

Alastair R. Beresford
University of Cambridge

Abstract

Traditional key stretching lacks a strict time guarantee due to the ease of parallelized password guessing by attackers. This paper introduces Sloth, a key stretching method leveraging the Secure Element (SE) commonly found in modern smartphones to provide a strict rate limit on password guessing. While this would be straightforward with full access to the SE, Android and iOS only provide a very limited API. Sloth utilizes the existing developer SE API and novel cryptographic constructions to build an effective rate-limit for password guessing on recent Android and iOS devices. Our approach ensures robust security even for short, randomly-generated, six-character alpha-numeric passwords against adversaries with *virtually unlimited* computing resources. Our solution is compatible with approximately 96% of iPhones and 45% of Android phones and Sloth seamlessly integrates without device or OS modifications, making it immediately usable by app developers today. We formally define the security of Sloth and evaluate its performance on various devices. Finally, we present HiddenSloth, a deniable encryption scheme, leveraging Sloth and the SE to withstand multi-snapshot adversaries.

1 Introduction

Smartphones store very sensitive information, ranging from corporate secrets to our most intimate messages and pictures. All users rely on their smartphones to protect their data when devices are lost, stolen, or seized. This is accomplished at the device level through full disk encryption. Apps with strong security requirements typically introduce an additional layer of encryption. Example apps include password managers and wallets for digital currency which store sensitive data and need to work offline.

While such apps often allow access using biometrics as a convenience, they typically require a master password as a fallback since sensor readings might be unreliable or to facilitate access by different people. In some cases, apps may intentionally avoid biometric authentication due to concerns about false positives or a desire to prevent involuntary unlocking of the secret: a user can deny knowledge of a password, whereas fingerprints can be obtained without their consent.

Passwords present a significant challenge: user-chosen passphrases are typically low-entropy, something which is particularly true on mobile devices with small on-screen keyboards. This makes apps that use passwords vulnerable to brute-force attacks. As a countermeasure, apps typically use key stretching schemes to increase the cost for an adversary. Such schemes increase the computational costs, e.g. PBKDF2, or use memory-hard functions, e.g. Argon2. Unfortunately, we have to choose high (and thus expensive) parameters in anticipation that the adversary has access to many computers.

Modern smartphones come with a built-in Secure Element (SE), a separate chip that is hardened against physical tampering and side-channel attacks, and contains its dedicated CPU, RAM, storage, and operating system. The SE communicates with the rest of the system by message passing which means that even a local adversary with kernel-level privileges and hardware access cannot break the security properties of the SE. While the smartphone vendor can deploy custom code to the SE, including to rate limit operations, app developers only have access to a limited API that provides standard cryptographic operations.

Nevertheless, in this paper, we show that it is possible to build an effective key stretching scheme with SEs as found on smartphones today by using the limited bandwidth of the chip as an intentional bottleneck. Under the assumption that the SE is secure, this places an absolute time cost on each attempt to guess a password since the adversary is forced to make all guesses via the SE, and access to additional computational power is of no benefit. This is in contrast to traditional key stretching schemes where difficult trade-offs are required. For example, an increase in the number of rounds of a key stretching scheme increases the cost of a successful attack for a password of a given strength but also increases the time taken by the app to authenticate the legitimate user.

In this paper, we also explore particularly sensitive scenarios where encryption in itself might not be enough, and we consider the case where the user may need to deny the existence of any meaningful content at all. Example scenarios include an employee crossing an international border with confidential business information, or a whistleblower whose device has been confiscated. In these situations, the user wants plausible deniability: the ability to deny the existence of information so that they cannot be pressured into revealing their password. Deniable encryption (DE) schemes allow exactly this. DE decryption functions always fail unless it is called with the correct password (if any such password exists). However, decryption failure is not distinguishable from trying to decrypt any (random) byte string that does not contain hidden information. This requires an app to initialize on installation any local state using a randomly-encrypted ciphertext. Such DE schemes can be used to build deniable storage containing hidden volumes. DE schemes inherently rely on passwords as a means of deniable authentication and our work provides an effective SE-backed DE scheme. Our DE scheme leaks no information even if an attacker is given multiple storage snapshots; e.g., access to several snapshots of the device state might be available through cloud backups.

We make the following contributions

1. We design the Sloth key stretching scheme to provide effective rate-limiting against powerful adversaries. Our scheme is practical since it uses existing APIs and does not require software or hardware modifications.

2. We evaluate these schemes on different Android and iOS phone models and show that even short passwords of six alpha-numerical characters provide strong security against the most powerful adversaries while keeping user-initiated unlock operations faster than 1 second.

3. We formally capture the security of schemes using SEs by introducing the model of a wall-time-bounded (WT) adversary with oracle access to the SE; we prove the Sloth scheme secure in this setting.

4. We design the HiddenSloth DE scheme that can withstand multi-snapshot adversaries and prove it secure.

2 Background

Hardware support for executing code in a secure context is available in most modern smartphones. Trusted Execution Environments (TEE) are the first architecture that achieved wide integration. TEE implementations, such as ARM TrustZone [33], run on the main chip, but in a privileged context so that even a compromised Kernel cannot manipulate them. However, TEEs generally come with limited protection against side-channel and physical attacks. Secure Elements (SE) are standalone components with their dedicated CPU, memory, and storage. This provides stronger isolation and protection against physical attacks.

Android allows developers to create and use keys via its API which abstracts a *Keymaster* interface to manage keys. From API level 23 (Android 6.0 M, released 2015) application developers can verify that keys are stored inside secure hardware. For phones released around 2015, this would typically mean that they are managed by a Keymaster implementation inside a TEE. However, from API level 28 (Android 9 P, released 2018), Android supports the StrongBox Keymaster implementation which must be implemented using an SE [19, 21]. Google has supported StrongBox in its flagship models since the Pixel 3 release in 2018. Other devices might have included SEs before this, but app developers would not have been able to use them.

Apple refers to SEs in their devices as a *Secure Enclave* and they have been supported in iOS devices since the iPhone 5S (released 2013) [4]. From the beginning, they have been used to protect biometric data and secure device encryption. However, they only were exposed to app developers with iOS 13 (released 2019) [7].

While TEEs and SEs allow for a much smaller implementation and attack surface, they are not impenetrable. Researchers have successfully extracted private keys from TrustZone implementations from Qualcomm [35] and Samsung [36]. Intel's SGX extension is similar in broad terms to TrustZone where there are many documented attacks [30].

Appendix A provides a survey on the availability of SEs in modern smartphones, their APIs, and their performance. We find that 96% of iPhones and 45% of Android devices provide SE access to app developers. Virtually all recent devices have at least TEE-backed key protection.

2.1 Android and iOS APIs

Code that runs on the SE is (by design) executed without any control by the operating system. Therefore, platforms do not allow developers to run custom code. Instead, they offer access to specific cryptographic operations through APIs. These APIs serialize the user request and send it to the SE where it is parsed and executed. The result is returned similarly.

We discuss the API on iOS first as it consists of very few operations. This provides a minimal attack surface, but as we will see later, it also stands in the way of efficient software implementation. At the time of writing, iOS only supports storing private EC keys of type P256 [8]. Once created inside the SE, the API provides methods to sign data, output the corresponding public key, and perform key agreement via ECDH. The API also provides methods for encrypting and decrypting data. However, the documentation is not clear on whether this happens entirely within the SE, or whether the ECDH result is shared with an AES engine outside the chip.

The Android API offers more operations and key types to developers. It uses the Keymaster API which abstracts the

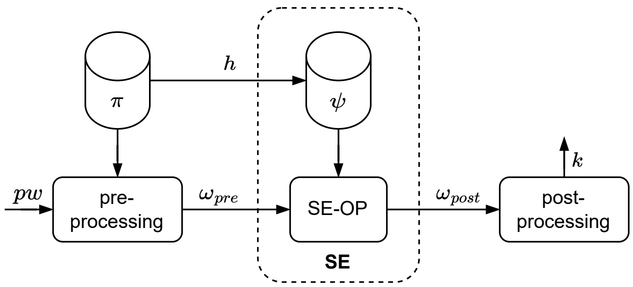

Figure 1: Overview of all Sloth schemes. The user password, pw, is first pre-processed on the device with access to device state π, producing ω_{pre}; the SE executes a cryptographic function over ω_{pre} using its internal (hidden) state ψ; the SE's output, ω_{post}, is then returned to the device; the device can then post-process ω_{post}, for example by applying a key derivation function, resulting in k as the final output.

actual key handling from the user. Device vendors implement the Keymaster HAL which serializes the API calls and communicates with the backend. This backend could be a service running in a TEE or SE. For our paper, we are interested in the StrongBox Keymaster implementation which requires using a SE [21]. It guarantees the availability of the following algorithms: RSA-2048, AES-128/256, ECDSA P-256, ECDH P-256, HMAC-SHA256, and 3DES. The availability of symmetric cryptography allows us to come up with a simpler design for Android devices. However, it also increases the complexity of the implementation that device manufacturers have to provide. Android recently added a new API `setMaxUsageCount` for OS version 12+ (Android S, API 31) that deletes the key after a given number of operations. While it would be convenient, it is implemented at the OS level[1], as StrongBox does not have sufficient persistent per key storage and thus cannot manage respective counters internally.

3 System Overview

Figure 1 provides a high-level overview common to all our Sloth schemes. Sloth distinguishes between two execution spaces: the general device space and the SE. We assume operations performed on the main device may be controlled by the adversary (e.g. using a local kernel exploit) while we assume the SE is a separate piece of hardware and remains secure (see Section 3.1). The user inputs a password, pw, in the user space. The sloth schemes start by pre-processing pw using the state $\pi \in \Pi$ where π is accessible in user space; the output of this pre-preprocessing step, ω_{pre}, is given as input to the SE. The SE maintains its own secure state, $\psi \in \Psi$, with cryptographic keys inaccessible to the device. Each key

[1] https://cs.android.com/android/platform/superproject/+/master:system/keymint/common/src/tag.rs;l=314-333;drc=ed657df7c7b329fb3d26eb0ce88af92594245ae8

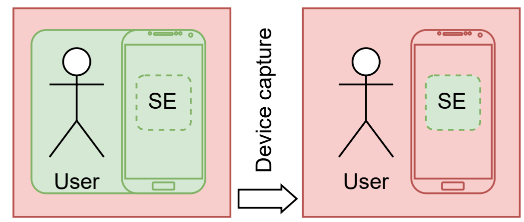

Figure 2: The user operates their smartphone securely before capture. Only the SE resists the adversary after capture.

maintained in ψ is associated with a unique (and public) key handle h. The SE accesses ψ and executes a cryptographic operation SE-OP; its output ω_{post} is returned to the user space for further post-processing to derive the final key k. The key k can then be used for authentication, full disk encryption, or deniable encryption protocols.

3.1 Threat Model

We assume an adversary can physically capture the device. For instance, they find a lost device on the street or stop the user at a border crossing. Before the capture, we assume that the device can be securely and confidentially operated by the user. After the capture, only the SE resists full-access by the adversary. We illustrate this in Figure 2.

The adversary aims to determine whether the device contains any information that has been encrypted with a user-chosen passphrase and if so recover the password, and therefore the information. We assume passwords are a sequence of characters, including passphrases. Further, we assume that if the adversary can determine that encrypted information is present on the device, they pressure the user with this evidence and obtain the correct password. In other words, a successful scheme must provide plausible deniability and prevent an adversary from distinguishing between encrypted information and random data.

The input (i.e. the user password or passphrase) to our Sloth schemes is typically not uniformly distributed. However, we assume that it has "enough randomness" to make it hard for an attacker to guess and thus can be modeled as a min-entropy probability distribution:

Definition 1 (Min-Entropy Distribution [27])**.** A probability distribution $\mathcal{X}$ has min-entropy (at least) m if for all a in the support of $\mathcal{X}$ and for random variable X drawn according to $\mathcal{X}$, $\textsc{Prob}(X = a) \leq 2^{-m}$.

The adversary can also use the secure element SE as a black box and can perform any operations via the available API (through Oracle queries which we denote O_{SE}). However, the adversary cannot extract information about the key material from the SE and they cannot clone the SE. Similar guarantees are given by Apple for their Secure Enclave [4] and

Google for StrongBox implementations [21]. These claims are taken seriously by vendors: Apple and Google award up to $250,000 [6] and $1,000,000 [20], respectively, for the discovery of relevant vulnerabilities.

We introduce an abstract definition of an SE. The SE operates on a state ψ that can only be (meaningfully) accessed by it. In practice, many SEs have limited internal storage and store an encrypted ciphertext of their state on storage managed by the OS. This encryption is performed using a secret internal key. Definition 2 introduces a generic SE as there exist SEs with different capabilities. Later sections introduce more specialized definitions for SEs with various capabilities.

Definition 2 (Secure Element)**.** A Secure Element SE operates on a hidden state $\psi \in \Psi$ using the algorithm SE.INIT and possible extension algorithms. It also has a timing function T_{ALG} that maps each algorithm ALG and its arguments to a wall time cost. SE.INIT : $\varnothing \rightarrow \Psi$ initializes an empty state Ψ with $T_{\text{SE.INIT}}() = 1$. Only the SE that initialized the state can operate on it with any of the extension algorithms.

We assume the adversary has a limited *wall time budget B* that is spent when performing oracle operations. The costs for each operation OP is defined by T_{OP} and deduced from B before it is executed. We require that the adversary ends the experiment with a non-negative time budget $B \geq 0$. We define for any algorithm (including adversary and challenger):

Definition 3 (Wall Time Algorithm)**.** A wall time (WT) algorithm is a PPT algorithm A with an initial wall time budget of $B_A \in \mathbb{N}$. During its execution it can perform the operations $\text{OP}_i(p_1, p_2, \dots)$ using the oracle O_{SE} given $\sum_i T_{\text{OP}_i}(p_1, p_2, \dots) \leq B_A$.

In comparison to standard security definitions, the strength is no longer supported by just asymptotic computational effort in $poly(\lambda)$, but also by a concrete wall time budget B. This is a strong property, as it is independent of advances in processing power or utilizing more machines.

3.2 Notations

Throughout the paper, let λ be a freely chosen but fixed security parameter. We use named tuple notation $x.y \leftarrow z$ to briefly describe the associative map operation: $X \leftarrow X \cup \{y \rightarrow z\}$. Table 3.2 summaries the symbols used in the rest of the paper.

4 The Sloth Key Stretching Scheme

We now describe the design of two concrete variants of our key stretching scheme Sloth. The variant LongSloth (§4.1) only uses symmetric operations and is simpler. However, it is incompatible with the iOS API which does not offer symmetric cryptographic operations. On the other hand, RainbowSloth (§4.2) is compatible with both Android and iOS APIs as it only requires support for a DH key exchange, but at the cost of higher complexity and additional assumptions.

Algorithms and schemes	
KDF	Key derivation function
H	Hash function
Ξ	Key stretching scheme
Δ	Deniable encryption scheme
Parameters	
λ	Security parameter for computational security
l	Length of ω_{pre} in LongSloth
n	Count of ω_{pre}s in RainbowSloth
Variables	
$\pi \in \Pi$	Storage state (space)
$\psi \in \Psi$	State (space) inside the SE
h	Key handle for the SE
ω	An intermediate secret
k	The final derived secret key

Table 1: We use these symbols for algorithms, parameters, and variables in our descriptions throughout the paper.

4.1 LongSloth: Simple Key Stretching Scheme

LongSloth exclusively relies on symmetrical operations both outside and inside the secure element SE. This enables extremely efficient implementations on Android, but cannot be implemented on iOS (see §4.2 for a variant compatible with iOS). Particularly, LongSloth operates on an SE equipped with HMAC support:

Definition 4 (SE with HMAC Support)**.** A Secure Element with HMAC support SE-WITH-HMAC is an SE that has two extension algorithms:

- SE.HMACKEYGEN : $\Psi \times H \rightarrow \Psi$ generates a new secret key k and updates $\psi \in \Psi$ under handle $h \in H$: $\psi.h \leftarrow k$ requiring time $T_{\text{SE.HMACKEYGEN}}(\psi, h) = \Theta(1)$.
- SE.HMAC : $\Psi \times H \times M \rightarrow \{0,1\}^\lambda$ takes a message $m \in M$ and a key handle $h \in H$ and outputs the $\text{HMAC}(\psi, h, m)$ result of a message $m \in M = \{0,1\}^*$ requiring time $T_{\text{SE.HMAC}}(\psi, h, m) = c_{\text{HMAC}} * |m| = \Theta(|m|)$, where c_{HMAC} is a device specific constant.

Algorithm 1 describes the main operations of LongSloth. The procedure LONGSLOTH.DERIVE (Line 9) outputs a potentially existing key. During the pre-processing step (see Figure 1), LongSloth expands a user password to a bit string ω_{pre}. This is achieved using the PWHASH operation that reads a

Algorithm 1 The LongSloth protocol with the security parameters l, λ freely chosen, but fixed.

```
1:  procedure LONGSLOTH.KEYGEN(ψ, pw, h)
2:      π ← {}
3:      π.h ← h
4:      π.salt ←$ {0,1}^λ
5:      ψ ← SE.HMACKEYGEN(ψ, h)
6:      k ← LONGSLOTH.DERIVE(π, ψ, pw)
7:      return (π, ψ, k)
8:
9:  procedure LONGSLOTH.DERIVE(π, ψ, pw)
10:     ω_pre ← PWHASH(π.salt, pw, l)
11:     ω_post ← SE.HMAC(ψ, π.h, ω_pre)
12:     k ← HKDF(ω_post)
13:     return k
```

salt value from the local storage and then hashes the user password to a variable length output. The output length $l = |\omega_{pre}|$ is a configurable parameter to limit the guess rate based on the SE throughput rate. Next, ω_{pre} is computed by the SE-OP securely inside the SE with a key selected by the key handle h. This operation is an HMAC producing ω_{post} that is then hashed into the final key output $k = \text{HKDF}(\omega_{post})$ using the hash-based key derivation function HKDF [27].

The generation of a new key k is done similarly. The procedure LONGSLOTH.KEYGEN (Line 1) first initializes a new state π with the key handle h and a fresh random salt value. The SE then generates a new HMAC key under h. It returns the updated state and the key k by calling LONGSLOTH.DERIVE.

Security intuition. LongSloth is designed to withstand an adversary with a wall time budget $B < 2^m \times l \times c_{\text{HMAC}}$. That is, the adversary would need to call 2^m times the operation SE.HMAC(x) with $|x| = l$ (see Definition 3) to try all possible user passwords. Intuitively, the security of LongSloth relies on the observation that such an adversary is unable to distinguish a key k generated by LongSloth from a random bit string. Section 6 provides a formal security analysis based on the observation that HMAC is a PRF [12].

4.2 RainbowSloth: Key Stretching for iOS

RainbowSloth relies on an SE equipped with support for Elliptic Curve Diffie-Hellman (ECDH) key exchanges (Definition 5). RainbowSloth is more complex than LongSloth but is compatible with both Android and iOS.

Definition 5 (SE with ECDH Support)**.** A Secure Element with Elliptic Curve Diffie-Hellman support SE-WITH-ECDH is a SE that has two extension algorithms:

SE.ECDHPRIVKEYGEN : $\Psi \times H \rightarrow \Psi$ generates a private EC key k and updates $\psi \in \Psi$ under handle $h \in H$: $\psi.h \leftarrow k$ requiring time $T_{\text{SE.ECDHPRIVKEYGEN}}(\psi, h) = \Theta(1)$.

SE.ECDH : $\Psi \times H \times pub \rightarrow \{0,1\}^\lambda$ takes a EC public key pub and a key handle $h \in H$ and outputs the key exchange result ECDH$(\psi.h, pub)$ requiring time $T_{\text{SE.ECDH}}(\psi, h, pub) = c_{\text{ECDH}} = \Theta(1)$, where c_{ECDH} is a device specific constant.

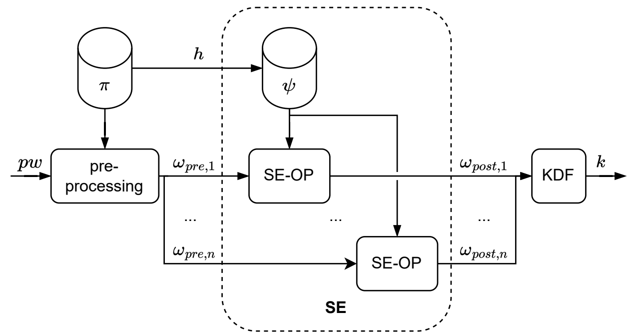

Figure 3: RainbowSloth scheme with a sequence of $\omega_{pre,i}$ inputs. The dashed area indicates the SE. Since there is no parallelism, all SE-OP are executed sequentially.

In contrast to LongSloth, RainbowSloth computes a sequence of fixed-sized public keys as its pre-secrets $\omega_{pre,i}$ (see Figure 3). The processing speed of each ECDH operation is constant since the public key size is defined by the underlying elliptic curve. Instead of increasing the input length of each operation, RainbowSloth increases the required number of SE-OP executions to achieve a given throughput limit.

Algorithm 2 describes the main operations of RainbowSloth. The procedure RAINBOWSLOTH.DERIVE decrypts a potentially existing key. It first expands the user password into multiple bit strings $\omega_{pre,i}$ with $i \in [1, n]$ calling PWHASH. Those bit strings are then formatted into P-256 public keys (Line 13). The SE uses the key handle h to select a private P-256 key and computes an ECDH operation with each $\omega_{pre,i}$ (Line 14). Since the SE has no parallelism, these operations take place one after another. The resulting values $\omega_{post,i}$, $i \in [1, n]$ are then jointly hashed into a final key k such that each contributes to all bits of k.

The generation of a new key k is done similarly. The procedure RAINBOWSLOTH.KEYGEN (Line 1) first initializes a new state π with the key handle h and a fresh random salt value. The SE then creates a new P-256 key under h. It returns all updated states and the key k by calling RAINBOWSLOTH.DERIVE as described above.

Security intuition. RainbowSloth is designed to withstand an adversary with wall time budget $B < 2^m \times n \times c_{\text{ECDH}}$ (see Definition 3); the adversary would need to call $2^m \times n$ times SE.ECDH to try all possible user passwords. Similarly to LongSloth, the security of RainbowSloth relies on the observation that the adversary is unable to distinguish a key k generated by RainbowSloth from a random bit string. Sec-

Algorithm 2 The RAINBOWSLOTH protocol with the security parameters n, λ freely chosen, but fixed.

procedure RAINBOWSLOTH.KEYGEN(ψ, pw, h)
  $\pi \leftarrow \{\}$
  $\pi.h \leftarrow h$
  $\pi.salt \xleftarrow{\$} \{0,1\}^\lambda$
  $\psi \leftarrow$ SE.ECDHPRIVKEYGEN(ψ, h)
  $k \leftarrow$ RAINBOWSLOWTH.DERIVE(π, ψ, pw)
  return (π, ψ, k)

procedure RAINBOWSLOTH.DERIVE(π, ψ, pw)
  $l \leftarrow n \times (\frac{256}{8})$
  $\omega_{pre,1} \| \cdots \| \omega_{pre,n} \leftarrow$ PWHASH($\pi.salt, pw, l$)
  for i = 1 ... n **do**
    $x \leftarrow$ REHASHTOP256($\omega_{pre,i}$)
    $\omega_{post,i} \leftarrow$ SE.ECDH($\psi, \pi.h, x$)
  $k \leftarrow$ HKDF($\omega_{post,1} \| \cdots \| \omega_{post,n}$)
  return k

tion 6 provides a formal security analysis assuming the SE runs the ECDH operations in a group where the generalized decisional Diffie-Hellman problem is hard [11].

Hashing into the P-256 Public Key Space. RainbowSloth requires an operation HASHTOP256 that maps any bit-string $seed \in \{0,1\}^*$ to a valid P-256 public key. Each P-256 public key can be represented by 256 bits for its X-coordinate and a bit that determines the Y-coordinate [17]. However, not all possible 257-bit strings refer to valid public keys, as the key space only has size 2^{256}. We designed the REHASHTOP256 algorithm and use it in our practical evaluation (Section 7).

The REHASHTOP256 algorithm (see Algorithm 3) considers the SEC-1 octal representation of P-256 curve points with point compression [17]. The representation for P-256 keys starts with a byte that is either `0x02` or `0x03` for the Y-coordinate information bit followed by 32 bytes for the X-coordinate. The algorithm repeatedly hashes the seed string and a counter to a candidate octet array. The first octet is then set as `0x02` or `0x03` based on the parity of the original value of the first octet. We try to convert each candidate array to a P-256 public key using the SEC1OCTETIMPORTP256 algorithm [17, 2.3.4]. If it returns "invalid" ($\perp$) or the point at infinity (O), the counter is incremented and the algorithm is repeated. Otherwise, a valid P-256 public key is returned. A similar approach is sketched in the Elligator paper [13, 1.4].

The runtime of this algorithm depends on finding a valid public key representation in one of its iterations. Since the output of the KDF can be considered pseudo-random, each iteration is independent and roughly half of the candidate octet arrays are invalid. We thus expect that the algorithm terminates with less than 10 iterations in $1 - 0.5^{10} \geq 99.9\%$ of all cases. Hashing with a counter instead of re-hashing the seed avoids falling into small loops that consist only of "bad

Algorithm 3 The REHASHTOP256 algorithm that maps any bit-string $seed \in \{0,1\}^*$ to a valid P-256 public key.

$counter \leftarrow 0$
while true do
  $arr \leftarrow$ KDF($seed \| counter$)$[0:32]$
  $arr[0] = 0x02 \mid (arr[0]\ \&\ 0x01)$
  $x =$ SEC1OCTETIMPORTP256(arr)
  if $x \neq \perp$ **and** $x \neq O$ **then return** x
  $counter \leftarrow counter + 1$

seeds". While long executions are unlikely, we are unaware of proofs of this algorithm's polynomial-time termination.

Alternatively, one might build construction using the Elligator Squared (E^2) [39] technique. E^2 maps an elliptic curve point to a bit string that is indistinguishable from random. Such a bit string can then be mapped back to an elliptic curve point. While it can be computed efficiently [9], E^2 bit strings require more space. A more practical concern is that we are not aware of any freely available implementation. However, if deterministic runtime is required, E^2 can be used as a drop-in replacement for REHASHTOP256.

5 The Sloth Deniable Encryption Scheme

Our generic methods to store and derive a key k can be used to build a deniable encryption (DE) scheme. We require a DE scheme to have three methods: (i) an INIT method that initializes a new storage of a given maximum size, (ii) an ENCRYPT method that stores data encrypted with a given password, and (iii) a DECRYPT method that retrieves and decrypts data from storage if there is any saved with the given password (and otherwise fails).

We present two variants of our deniable encryption scheme, 1S-HiddenSloth (Section 5.1) and MS-HiddenSloth (Section 5.2). 1S-HiddenSloth withstands an adversary capable of capturing a single snapshot of the user's device and MS-HiddenSloth withstands an adversary capable of capturing multiple snapshots at different points in time.

5.1 1S-HiddenSloth: Single-snapshot

For most real-world scenarios, the single-snapshot adversary is the most realistic threat model. An adversary gaining access to a phone through finding or stealing it has no prior snapshot or knowledge of the phone state, and hence cannot perform multi-snapshot attacks. Similarly, an adversary who confiscates a device can rarely do so covertly. If a user suspects an adversary may have taken a snapshot of their phone, they can reset the phone and thus revert to a single-snapshot scenario. Nevertheless, there may be situations where an adversary can obtain multiple copies of the phone state over time; this is addressed in Section 5.2.

Algorithm 4 The *deniable encryption against single-snapshot* DESS protocol (1S-HIDDENSLOTH) for maximum data size $s \leq 2^{31}$ and security parameter λ freely chosen, but fixed. The underlying SLOTH protocol can be instantiated with either variant.

1: **procedure** DESS.INIT(ψ, h, s)
2: $\quad pw \xleftarrow{\$} \{0,1\}^\lambda$
3: $\quad \psi, \pi, k \leftarrow$ SLOTH.KEYGEN(ψ, pw, h)
4: $\quad \pi \leftarrow$ DESS.ENCRYPT$(\pi, \psi, pw, [])$
5: $\quad$ **return** (π, ψ)
6:
7: **procedure** DESS.ENCRYPT$(\pi, \psi, pw, data)$
8: $\quad k \leftarrow$ SLOTH.DERIVE(π, ψ, pw)
9: $\quad x \leftarrow$ UINT32$(|data|) \parallel data \parallel 0^{(|\pi.data|-|data|-4)}$
10: $\quad \pi.iv \xleftarrow{\$} \{0,1\}^\lambda$
11: $\quad \pi.blob, \pi.tag \leftarrow$ AE.ENC(k, iv, x)
12: $\quad$ **return** π
13:
14: **procedure** DESS.DECRYPT(π, ψ, pw)
15: $\quad k \leftarrow$ SLOTH.DERIVE(π, ψ, pw)
16: $\quad x \leftarrow$ AE.DEC$(k, \pi.iv, \pi.blob, \pi.tag)$
17: $\quad$ **if** $x = \perp$ **then return** $\perp$
18: $\quad s' \leftarrow$ UINT32$(x[:4])$
19: $\quad$ **return** $x[4 : 4 + s']$

Algorithm 4 presents 1S-HiddenSloth, a single-snapshot resistant deniable encryption scheme. When calling the INIT procedure, the algorithm allocates a storage file *blob* and fills it with random bytes to the given size (plus an allowance for overhead). This is implemented by encrypting a zeroed payload using a randomly chosen password (Line 4). Upon encrypting new user data by calling the ENCRYPT procedure, the user provides a passphrase *pw* used to derive a cryptographic key k (using either LongSloth or RainbowSloth, see Section 4). The payload is then prefixed with a 4-bytes integer representing its length and padded to the maximum size s. The resulting byte string is encrypted using the key k, for instance using AES-GCM. For the DECRYPT operation, the key k is derived analogously and the algorithms attempt to decrypt the beginning of the file. If this operation fails, an adversary will not be able to tell whether the passphrase was wrong or there was no previous call to ENCRYPT at all.

5.2 MS-HiddenSloth: Going Multi-Snapshot

Sometimes the storage state is available to an adversary on occasions. This may happen if an app's data is backed up to the cloud to protect against device loss. Intuitively, 1S-HiddenSloth does not withstand such an adversary because changes in the ciphertext leak whether the storage has been overwritten between the device capture events.

MS-HiddenSloth overcomes this limitation and allows the adversary to capture the storage state π multiple times. However, our model restricts the adversary to only access the SE once they finally gain physical access to the device (recall Figure 2). That is, the adversary can perform SE calls only during the last device capture event. We believe this restriction is practical as smartphones are usually with the user and hence not likely available for covert access by the adversary[2].

MS-HiddenSloth requires a SE that supports (unauthenticated) symmetric encryption:

Definition 6 (SE with Symmetric Encryption)**.** A Secure Element with symmetric encryption support SE-WITH-SYMMENC is an SE that has three extension algorithms:

SE.SYMMKEYGEN : $\Psi \times H \rightarrow \Psi$ generates a new secret key k and updates $\psi \in \Psi$ under handle $h \in H$: $\psi.h \leftarrow k$ requiring time $T_{\text{SE.SYMMKEYGEN}}(\psi, h) = \Theta(1)$.

SE.SYMMENC : $\Psi \times H \times IV \times M \rightarrow C$ takes an initialization vector $iv \in IV$ and a key handle $h \in H$ and outputs the ciphertext SYMMENC.ENCRYPT$(\psi.h, iv, m) = c \in C$ for the message $m \in M$ requiring time $T_{\text{SE.SYMMENC}}(\psi, h, m) = \Theta(|m|)$.

SE.SYMMDEC : $\Psi \times H \times IV \times C \rightarrow M$ takes an initialization vector $iv \in IV$ and a key handle $h \in H$ and outputs the message SYMMENC.DECRYPT$(\psi.h, iv, c) = m \in M$ for the ciphertext $c \in C$ requiring time $T_{\text{SE.SYMMENC}}(\psi, h, c) = \Theta(|c|)$.

To protect against a multi-snapshot adversary MS-HiddenSloth wraps the storage in another layer of encryption guarded by an additional symmetric key k_{SE} held in the SE's secure state under handle h'. The outer layer is periodically re-encrypted using a new temporary key tk, which in turn is re-encrypted with a fresh k_{SE}. The re-encryption process (DEMS.RATCHET) should happen at least as often as the adversary has the opportunity to access the stored data. For instance, for a backed-up app, this would happen after every upload operation. In other cases, every app restart could be used as a trigger event. An important design feature of MS-HiddenSloth is that it does not require the user's password to execute the re-encryption process; that is, the procedure DEMS.RATCHET can be executed entirely in the background without user interaction.

MS-HiddenSloth algorithm. Algorithm 5 presents all the operations of MS-HiddenSloth. The initial state is initialized similarly to 1S-HiddenSloth but with the creation of an additional SE secret key (k_{SE}) associated with the handle h' (Line 4). To encrypt new data, MS-HiddenSloth first encrypts them as in 1S-HiddenSloth; it then generates an ephemeral key tk that is used to re-encrypt the data (Line 11). This ephemeral key is finally encrypted using the SE's secret key

[2] Also, if the adversary had such local privileged access multiple times it could simply install malware that records the keyboard and screen.

Algorithm 5 The *deniable encryption against multi-snapshot* DEMS protocol (HIDDENSLOTH) for max. size $s \leq 2^{31}$ and security parameter λ freely chosen, but fixed. The underlying SLOTH protocol can be instantiated with either variant.

1: **procedure** DEMS.INIT(ψ, h, s)
2: $\quad \pi, \psi, k \leftarrow$ DESS.INIT($\psi, h \,\|\, 0, s$)
3: $\quad \pi.h' \leftarrow h \,\|\, 1$
4: $\quad \psi \leftarrow$ SE.SYMMKEYGEN($\psi, \pi.h'$)
5: $\quad pw \xleftarrow{\$} \{0,1\}^{\lambda}$
6: $\quad \pi \leftarrow$ DEMS.ENCRYPT($\pi, \psi, pw, []$)
7: $\quad$ **return** (π, ψ)
8:
9: **procedure** DEMS.ENCRYPT($\pi, \psi, pw, data$)
10: $\quad \pi \leftarrow$ DESS.ENCRYPT($\pi, \psi, pw, data$)
11: $\quad \pi.tk \leftarrow$ AE.KEYGEN(); $\pi.tiv \xleftarrow{\$} \{0,1\}^{\lambda}$
12: $\quad \pi.blob, \pi.ttag \leftarrow$ AE.ENC($\pi.tk, \pi.tiv, \pi.blob$)
13: $\quad$ **for** $\mathcal{K} \in \{iv, tag, tk, tiv, ttag\}$ **do**
14: $\quad\quad \pi.\mathcal{K} \leftarrow$ SE.SYMMENC($\psi, \pi.h', \pi.\mathcal{K}$)
15: $\quad$ **return** π
16:
17: **procedure** DEMS.DECRYPT(π, ψ, pw)
18: $\quad$ **for** $\mathcal{K} \in \{iv, tag, tk, tiv, ttag\}$ **do**
19: $\quad\quad \pi.\mathcal{K} \leftarrow$ SE.SYMMDEC($\psi, \pi.h', \pi.\mathcal{K}$)
20: $\quad \pi.blob \leftarrow$ AE.DEC($\pi.tk, \pi.tiv, \pi.blob, \pi.ttag$)
21: $\quad$ **return** DESS.DECRYPT(π, ψ, pw)
22:
23: **procedure** DEMS.RATCHET(π, ψ)
24: $\quad$ **for** $\mathcal{K} \in \{iv, tag, tk, tiv, ttag\}$ **do**
25: $\quad\quad \pi.\mathcal{K} \leftarrow$ SE.SYMMDEC($\psi, \pi.h', \pi.\mathcal{K}$)
26: $\quad \pi.blob \leftarrow$ AE.DEC($\pi.tk, \pi.tiv, \pi.blob, \pi.ttag$)
27: $\quad \pi.tk \leftarrow$ AE.KEYGEN(); $\pi.tiv \xleftarrow{\$} \{0,1\}^{\lambda}$
28: $\quad \pi.blob, \pi.ttag \leftarrow$ AE.ENC($\pi.tk, \pi.tiv, \pi.blob$)
29: $\quad \psi \leftarrow$ SE.SYMMKEYGEN($\psi, \pi.h'$)
30: $\quad$ **for** $\mathcal{K} \in \{iv, tag, tk, tiv, ttag\}$ **do**
31: $\quad\quad \pi.\mathcal{K} \leftarrow$ SE.SYMMENC($\psi, \pi.h', \pi.\mathcal{K}$)
32: $\quad$ **return** (π, ψ)

generated at Line 4 and stored in the user space π. To decrypt existing data, MS-HiddenSloth first decrypts the ephemeral key tk (Line 19); it then uses that key to decrypt the outer encryption layer (Line 20) and finally decrypt the data (Line 21).

The re-encryption process (ratchet) works similarly to decryption followed by an encryption operation. It firsts retrieves the temporary key tk and uses it to decrypt the outer encryption layer. It then generates a new temporary key that is used to re-encrypt the data and thus create a new outer encryption layer; the new temporary key is then encrypted by the SE and persisted in the user space. For performance reasons, MS-HiddenSloth does not perform the re-encryption process over the actual data $\pi.blob$ inside the SE; it instead re-encrypts the data with a temporary key $\pi.tk$. This temporary key $\pi.tk$ and its related fields $\pi.tiv$, $\pi.ttag$ are freshly generated at every encryption (Line 11) and ratchet step (Line 27); they are only stored encrypted with the SE's secret key generated at Line 4. The indirection via $\pi.tk$ allows to leverage the faster AES engine on the main CPU without sacrificing security.

Security intuition. This extra encryption layer is sufficient to withstand a multi-snapshot adversary as the adversary will always encounter a random-looking ciphertext. When the adversary finally gains access to the SE, they can only reverse it for the last ciphertext, but not for any beforehand. A similar technique of adding a reversible encryption layer is not possible in existing schemes without an SE as the encryption key would be part of the captured storage state. An alternative would be decrypting and re-encrypting the user data at every event. This would however be impractical as it necessitates the user to input their passphrase for each of these events.

5.3 Practical Implementation Details

Sloth is generally more practical than other schemes that rely on compute and memory-hard functions. This is because their parameters are chosen from the perspective of an attacker with access to a large parallel cluster of machines that are more powerful than a smartphone.

For the regular Sloth key stretching schemes and 1S-HiddenSloth, the implementation must not use a file system or underlying storage technology that allows an attacker to discover whether and when a file has been overridden. However, since the ratchet steps for MS-HiddenSloth are assumed to be predictable from the adversary's perspective, its security guarantees hold for any form of storage.

The actual implementation into a production app also needs to consider the allocated space for the deniable encryption since it should always be the same size regardless of the actual usage. Developers have to weigh the costs of having a large encrypted file for non-users against the storage requirements of its active users. Also, developers must take care that the deniable parts of the application do not leave any other traces in the form of crash reports and logging output.

6 Security Analysis

We formally capture the security of our Sloth schemes using experiments between a challenger $\mathcal{C}$ and a wall-time bounded (WT) adversary $\mathcal{A}$ (Definition 3).

6.1 Key Stretching Security

We prove the security of LongSloth (Section 4.1) and RainbowSloth (Section 4.2). For this we formally define an SE-backed key stretching scheme and and experiments for key stretching indistinguishability and hardness. The success of a

WT adversary is controlled by the time required by the SE to execute operations $T_{\text{SE-OP}}$ (and the Sloth parameters).

Definition 7. A key stretching scheme Ξ for fixed security parameter λ consists of two algorithms:

KEYGEN : $\Psi \times \mathcal{P} \times H \rightarrow \Pi \times \Psi \times \{0,1\}^\lambda$ takes an SE state, a password, and a key handle and returns a new storage state, an updated SE state, and the derived key.

DERIVE : $\Pi \times \Psi \times \mathcal{P} \rightarrow \{0,1\}^\lambda$ takes a storage state, SE state, and a password to derive a key $k \in \{0,1\}^\lambda$.

We require for correctness that after the initialization operation $(\pi, \psi, k) \leftarrow \text{KEYGEN}(\psi', pw, h)$ and all subsequent $\text{DERIVE}(\pi, \psi, pw)$ executions return the same k.

The adversary must not learn any information about either the password or the generated key. We propose an experiment where the adversary is required to distinguish between a truly random bit string and a key generated from a password. If an WT adversary fails to do so better than with negligible probability, even when given access to the stored information and oracle access to the SE, we say that the key stretching scheme is indistinguishable.

Definition 8 (Key Stretching Indistinguishability Experiment)**.** Let $\mathcal{P}$ be a probability distribution with min-entropy m. Let $\mathcal{A}$ be a WT adversary with wall time budget B and $\mathcal{C}$ a WT challenger. Let Ξ be a key stretching scheme. Then the key stretching indistinguishability experiment $\text{KEYIND}_{\mathcal{A}_\Xi}(\lambda, \mathcal{P})$ is defined as follows:

1. The adversary $\mathcal{A}$ provides the challenger $\mathcal{C}$ with a password $pw \xleftarrow{\$} \mathcal{P}$.
2. $\mathcal{C}$ randomly samples $b \xleftarrow{\$} \{0,1\}$. Let ψ be freshly initialized and h an arbitrary, but fixed key handle.
3. $\mathcal{C}$ computes $\pi, \psi, k_0 \leftarrow \Xi.\text{KEYGEN}(\psi, pw, h)$ under λ and samples $k_1 \leftarrow \{0,1\}^\lambda$.
4. The adversary $\mathcal{A}$ receives (π, ψ, k_b).
5. $\mathcal{A}$ receives oracle access O_{SE} (WT conditions).
6. $\mathcal{A}$ outputs a bit b' and wins iff $b = b'$.
7. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

Definition 9 (Key Stretching Indistinguishability)**.** A key stretching scheme Ξ is m-entropy indistinguishable if for all WT adversaries $\mathcal{A}$, there is a negligible function NEGL:

$$Pr[\text{KEYIND}_{\mathcal{A},\Xi}(\lambda, \mathcal{P}) = 1] \leq \frac{1}{2} + \text{NEGL}(\lambda),$$

where $\mathcal{P}$ is a probability distribution with min-entropy m.

The LongSloth and RainbowSloth schemes respectively described in Section 4.1 and Section 4.2 fulfill these definitions as per the following theorems.

Theorem 1 (LongSloth Indistinguishability)**.** Let $\mathcal{P}$ be a random distribution with min-entropy m. The key stretching scheme LongSloth is m-entropy indistinguishable.

Proof. We present a full proof in Appendix B.1.1. □

Theorem 2 (RainbowSloth Indistinguishability)**.** Let $\mathcal{P}$ be a random distribution with min-entropy m. The key stretching scheme RainbowSloth is m-entropy indistinguishable.

Proof. We present a full proof in Appendix B.2.1. □

In addition to the theoretical result above, we are interested in the practical success rate of a brute-force attacker, i.e., given k find pw, with wall time budget B.

Definition 10 (Key Stretching Hardness Experiment)**.** Let $\mathcal{P}$ be a probability distribution with min-entropy m. Let $\mathcal{A}$ be a WT adversary with wall time budget B and $\mathcal{C}$ a WT challenger. Let Ξ be a key stretching scheme. Then the key stretching hardness experiment $\text{KEYHARD}_{\mathcal{A}_\Xi}(\lambda, \mathcal{P})$ is defined below:

1. $\mathcal{C}$ samples a password $pw \xleftarrow{\$} \mathcal{P}$. Let ψ be freshly initialized and h an arbitrary (but fixed) key handle.
2. $\mathcal{C}$ computes $k = \Xi.\text{KEYGEN}(\psi, pw, h)$.
3. $\mathcal{A}$ receives (π, ψ, k).
4. $\mathcal{A}$ receives oracle access O_{SE} (WT conditions).
5. $\mathcal{A}$ outputs a pw' and wins iff $k = \Xi.\text{DERIVE}(\psi, pw', h)$.
6. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

Definition 11 (Key Stretching Hardness)**.** A key stretching scheme Ξ is σ-hard if for all WT adversaries $\mathcal{A}$ with wall-time budget B,

$$Pr[\text{KEYHARD}_{\mathcal{A},\Xi}(\lambda, \mathcal{P}) = 1] \leq \frac{B}{\sigma \cdot 2^m},$$

where $\mathcal{P}$ is a probability distribution with min-entropy m.

Theorem 3 (LongSloth Hardness)**.** Let $\mathcal{P}$ be a random distribution with min-entropy m. Then the key stretching scheme LongSloth with parameter l is $(l \cdot c_{\text{HMAC}})$-hard.

Proof. We present a full proof in Appendix B.1.2. □

Theorem 4 (RainbowSloth Hardness)**.** Let $\mathcal{P}$ be a random distribution with min-entropy m. Then the key stretching scheme RainbowSloth with parameter n is $(n \cdot c_{\text{ECDH}})$-hard.

Proof. We present a full proof in Appendix B.2.2. □

6.2 Deniable Encryption Security

Similarly to key stretching, we discuss the security of deniable encryption by giving formal definition, describing security experiments, and then showing that an adversary has at most a negligible advantage. We only discuss the multi-snapshot case and prove the security of MS-HiddenSloth. A similar (and simpler) reasoning applies to 1S-HiddenSloth.

Definition 12 (SE with MS Deniable Encryption Support)**.** A SE-backed deniable encryption scheme Δ for fixed security parameter λ and maximum data size $s \in \mathbb{N}$ consists of three algorithms:

INIT : $\Psi \times H \times \mathbb{N} \to \Pi \times \Psi$ takes a SE state, a key handle, and storage size s and returns a new storage state and SE state.

ENCRYPT : $\Pi \times \Psi \times \mathcal{P} \times \{0,1\}^s \to \Pi$ updates a storage state for given SE state and a password so that it now stores the given data.

DECRYPT : $\Pi \times \Psi \times \mathcal{P} \to \{0,1\}^s \cup \{\bot\}$ tries to decrypt from the storage state given the password. If successful, the previously stored data is returned, otherwise $\bot$.

RATCHET : $\Pi \times \Psi \to \Pi \times \Psi$ updates the storage and SE state by decrypting and re-encrypting the stored data.

Definition 13 (MS Deniable Encryption Indistinguishability Experiment)**.** Let λ be a fixed security parameter and s the maximum data size. Let $\mathcal{P}$ be a probability distribution with min-entropy m. Let $\mathcal{A}$ be a WT adversary with wall time budget B and $\mathcal{C}$ a WT challenger. Let Δ be a multi-snapshot deniable encryption scheme. Then the multi-snapshot deniable encryption indistinguishability experiment DE-MS-IND$_{\mathcal{A},\Delta}(\lambda, \mathcal{P})$ is defined as follows:

1. The challenger C randomly samples $b \xleftarrow{\$} \{0,1\}$ and $pw \xleftarrow{\$} \mathcal{P}$. Let ψ_0, ψ_1 be freshly initialized and h an arbitrary, but fixed key handle.
2. C computes $\pi_0, \psi_0 \leftarrow \Delta.\text{INIT}(\psi_0, h, s)$ and $\pi_1, \psi_1 \leftarrow \Delta.\text{INIT}(\psi_1, h, s)$ under λ.
3. A sends any $\tilde{b} \in \{0,1\}$ and $m \in \{0,1\}^s$ to C.
4. C executes $\pi_{\tilde{b}} \leftarrow \Delta_{MS}.\text{ENCRYPT}(\pi_{\tilde{b}}, \psi_{\tilde{b}}, pw, m)$, then C sends $\pi_{\tilde{b}}$ to A.
5. A can repeat execution from step 3 several times (under WT conditions).
6. C randomly samples $b \xleftarrow{\$} \{0,1\}$, computes $\pi_b, \psi_b \leftarrow \Delta.\text{RATCHET}(\pi_b, \psi_b)$, and provides A with (π_b, ψ_b).
7. A receives oracle access O_{SE} under the WT conditions.
8. A outputs a bit b' and wins iff $b = b'$.
9. The experiment returns 1 iff A wins, otherwise 0.

Definition 14 (MS Deniable Encryption Indistinguishability)**.** A multi-snapshot deniable encryption scheme Δ is B-time m-entropy secure if for all WT adversaries $\mathcal{A}$ with time budget B, there is a function NEGL such that

$$Pr[\text{DE-MS-IND}_{\mathcal{A},\Delta}(\lambda, \mathcal{P}) = 1] \leq \frac{1}{2} + \text{NEGL}(\lambda),$$

where $\mathcal{P}$ is a probability distribution with min-entropy m.

Theorem 5. (MS-HiddenSloth Indistinguishability) The multi-snapshot deniable encryption scheme MS-HiddenSloth is B-time m-entropy secure.

Proof. We present a full proof in Appendix B.3. □

Similarly to the key stretching scheme, we analyze the practical success rate of a brute-force attacker.

Definition 15 (Deniable Encryption Hardness Experiment)**.** Let $\mathcal{P}$ be a probability distribution with min-entropy m. Let $\mathcal{A}$ be a WT adversary with wall time budget B and $\mathcal{C}$ a WT challenger. Let Δ be a key stretching scheme. Then the deniable encryption hardness experiment DEHARD$_{\mathcal{A}_\Delta}(\lambda, \mathcal{P})$ is defined as follows:

1. $\mathcal{A}$ provides $\mathcal{C}$ with $data$ with $|data| > 0$.
2. $\mathcal{C}$ samples a password $pw \xleftarrow{\$} \mathcal{P}$. Let ψ be freshly initialized and h an arbitrary (but fixed) key handle. Let $\pi, \psi = \Delta.\text{INIT}(\psi, h, |data|)$.
3. $\mathcal{C}$ computes $\pi = \Delta.\text{ENCRYPT}(\pi, \psi, pw, data)$.
4. $\mathcal{A}$ receives (π, ψ).
5. $\mathcal{A}$ receives oracle access O_{SE} (WT conditions).
6. $\mathcal{A}$ outputs a pw' and wins iff $data = \Delta.\text{DECRYPT}(\pi, \psi, pw')$.
7. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

Definition 16 (Deniable Encryption Hardness)**.** A deniable encryption scheme Δ is σ-hard if for all WT adversaries $\mathcal{A}$ with wall-time budget B,

$$Pr[\text{DEHARD}_{\mathcal{A},\Delta}(\lambda, \mathcal{P}) = 1] \leq \frac{B}{\sigma \cdot 2^m},$$

where $\mathcal{P}$ is a probability distribution with min-entropy m.

Theorem 6 (MS-HiddenSloth Hardness)**.** Let $\mathcal{P}$ be a random distribution with min-entropy m. Then the deniable encryption scheme MS-HiddenSloth instantiated with a σ-hard key stretching scheme is σ-hard.

Proof. We present a full proof in Appendix B.3.2. □

7 Evaluation

We measure the performance of SEs in Android and iOS devices (§7.1), choose practical parameters for our schemes (§7.2), and test full implementations of LongSloth and RainbowSloth (§7.3) as well as HiddenSloth (§7.4).

7.1 Performance Characteristics of SEs

For iOS devices we measure the duration of the Secure Enclave's ECDH operations on the recent iPhones from XR to 14. These cover the Apple chips A12, A13, A14, and A15. We wrote a benchmark app that first creates a secret P-256 key within the SE. It then creates a new random public key and performs ECDH with the private key within the SE. We repeat the ECDH step 1,000 times with the new public keys and measure the elapsed time. The results (Figure 4) show that similar chips have similar run times, e.g., A15 for iPhone 13 and iPhone 14. All measurements are between 6 ms and

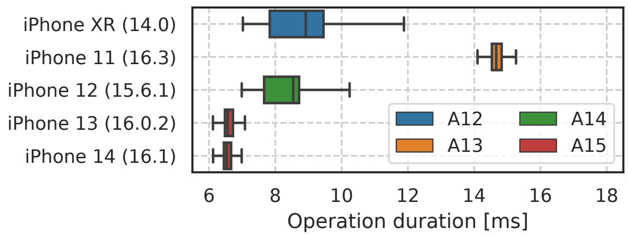

Figure 4: The duration of the ECDH operation on iOS for different phones (iOS version in brackets). The colors indicate the chip generation used in the respective phone.

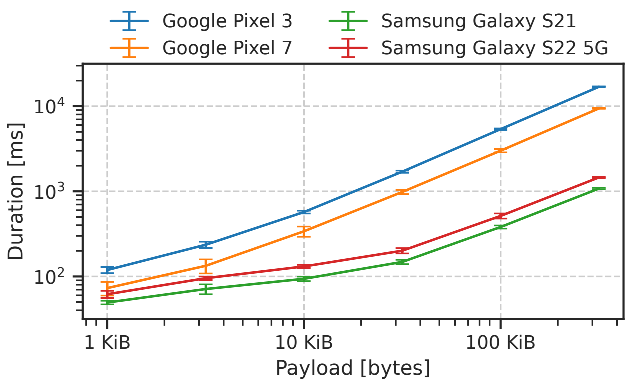

Figure 5: The duration of a HMAC operations on Android phones with StrongBox support. Both axes are log-scale.

16 ms with little variance per model. Surprisingly, the A13 chip has lower throughput than its predecessor. However, A13 is also the first one with a mathematically verified public key implementation [4] which might point to a difference in the underlying algorithm.

For Android devices, we measure the duration of HMAC executions for inputs of varying lengths on multiple devices. We wrote a benchmark tool that first creates a secret key within the SE. It then generates random byte arrays as inputs, passes these as input to the SE, and receives the computed HMAC value as a result. We repeat this step 10 times for each device and input length to measure the elapsed time. The results are shown in Figure 5. Intuitively, the measured time increases with input length in an almost linear manner. There are minor inflection points that differ between the devices. In our experiments, the Samsung phones also have a 10× higher bandwidth compared to the Google devices. For our parameter choice, this will result in a larger l value for those devices. Notably, for the same device and input length, the variance is very small. Once the parameters have been established for each device, their impact is predictable and dependable.

	Entropy	50 years	100 years
WordList (3 words)	38.8	3.4 ms	6.7 ms
WordList (4 words)	51.7	<0.1 ms	<0.1 ms
AlphaNum (5 chars)	29.8	1.7 s	3.4 s
AlphaNum (6 chars)	35.7	27.8 ms	55.5 ms
AlphaNum (7 chars)	41.7	0.4 ms	0.9 ms
PIN (9 digits)	29.9	1.6 s	3.2 s

Table 2: Overview of password configurations, their entropy (bits), and the required $t_{individual}$ for given security margins.

7.2 Parameter choice

For our evaluation, we target a security level of $T_{total} = 100$ years. We treat the acceptable password complexity, such as the alphabet size and the number of characters, as input parameters. Other than classic alphanumerical[3] passwords, we also consider passphrases that are based on the EFF word list [16] that contains 7776 words and PINs consisting of only digits. Let $|A|$ be the alphabet size and $|pw|$ the password length, then the entropy for a configuration is $e = \log_2(|A|^{|pw|})$. We want that an attacker's worst-case to match the targeted security level, i.e., if they brute-force the entire space, then they require at least T_{total}. Let T_{total} be the targeted security level, then for a password configuration with entropy e each password verification should take at least $t_{individual} = T_{total} \times 2^{-e}$ (see Table 7.2).

The password configurations that we show in this table are shorter than those used in web applications nowadays. This is because for Sloth the parameter choice does not need to conservatively assume an attacker with highly parallel computing resources. Short passwords, and in particular passphrases generated from word lists, are more memorable and can be generated by the application for the user — and thus avoid the problem of users picking weak passwords or reusing the same one for different services.

In our main evaluation (§7.3) we will examine two configurations: 3-word passphrases from the EFF word list ($c1$) and 6-character alphanumerical passwords ($c2$). In both cases, we multiply the minimal $t_{individual}$ (as per Table 7.2) by a **safety factor of ×10** for conservative parameter choice. Therefore: $t_{c1} = 67\,ms$ and $t_{c2} = 555\,ms$. This safety factor can account for the attacker over-clocking the SE and overhead by the operating system when communicating with the chip that an attacker might be able to "optimize away".

With the required minimum times t_{c1}, t_{c2} for the individual operations, we determine l for LongSloth on Android and n for RainbowSloth on iOS. On Android, we use the measurements from Figure 5 to fit a second-degree polynomial where

[3] Case-sensitive letters `a-zA-Z` and digits `0-9`, hence $|A| = 76$.

	3 words $t_{c1} = 67$ ms	6 characters $t_{c2} = 555$ ms
LongSloth (parameter: **l**)		
Google Pixel 3	1,500	11,600
Google Pixel 7	4,500	24,200
Samsung Galaxy S21	10,700	178,000
Samsung Galaxy S22	2,200	145,100
RainbowSloth (parameter: **n**)		
iPhone 11	6	43
iPhone 12	10	76
iPhone 13	12	95

Table 3: Parameter choices for given password configuration and its $t_{individual}$ augmented by a safety factor of 10×.

we set the y-values to the smallest measurement for a given size minus 2 standard deviations. We then pick the l value at the intersection with the desired duration and round to the next multiple of 100. On iOS we pick the 10th percentile value t_{p10} of our measurements as a conservative worst-case (i.e. fastest) duration and then compute $n_c = \lceil \frac{t_c}{t_{p10}} \rceil$ for $c \in \{c1, c2\}$. The resulting values are summarized in Table 3. If an app cannot find existing parameters for a new device type, it can perform a similar method on its first start to self-calibrate.

7.3 LongSloth and RainbowSloth

We implemented LongSloth on Android and RainbowSloth on iOS. All code including documentation and analysis scripts is available under an MIT license.[4] Where possible we use the existing cryptography APIs of the platform, with AES-GCM for symmetric authenticated encryption, and HKDF-SHA256 as the KDF. For PwHash we use the third-party LibSodium library which is available on both platforms and implements the memory-hard password hashing algorithm Argon2 [14]. For the evaluation, we choose the recommended OWASP parameters for Argon2id with 19 MiB of memory and an iteration count of 2 [31]. Since one could opt for another PwHash implementation, we exclude its runtime (50 ms) from our results for LongSloth and RainbowSloth.

For our evaluation, we are interested in the duration of the Sloth.derive operations, as these determine the time costs for an attacker. We use the parameters as chosen in Section 7.2 including the safety factor and execute each operation 10 times. The results are shown in Figure 6 for LongSloth on Android RainbowSloth on iOS. In all cases, the measured total durations comfortably exceed the threshold times t_{c1} and t_{c2}. This confirms that with our parameter choice, the

[4] https://github.com/lambdapioneer/sloth

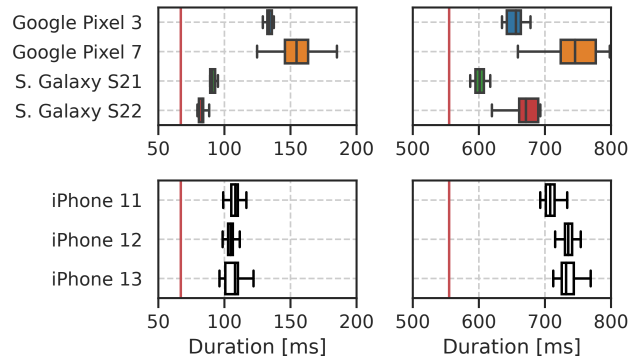

Figure 6: The duration of the LongSloth.Derive operation on Android (top) RainbowSloth.Derive operation on iOS (bottom). For both we evaluated the configurations $c1$ (left) and $c2$ (right) from Table 3 for different phones. The red lines indicate the threshold times t_{c1} and t_{c2}.

algorithm meets its minimum timing promise and hence key stretching security. The variance for the individual configurations is small which can allow for reducing the safety factor.

7.4 HiddenSloth

For the deniable encryption scheme HiddenSloth we evaluate its Ratchet methods for varying maximum data sizes s. This parameter s spans from a small storage size that might wrap text-only configurations (1 MiB) to larger ones that can store long chat histories including media (100 MiB). In these experiments, we use the faster t_{c1} for the underlying Sloth schemes. As HiddenSloth requires an SE-with-SymmEnc, we evaluate it only on Android devices. Our results are shown in Figure 7. The measured durations are similar for all test devices and range from around 700 ms for 1 MiB to about 2 s for storage with 100 MiB capacity. We note that the Ratchet step does not require the password and thus can be executed in the background without any user interaction. As such, even large storage sizes have no user-visible impact and we suggest scheduling it when the device is idle and charging.

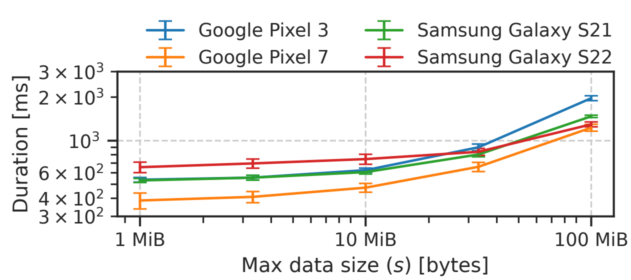

Figure 7: Duration of the HiddenSloth.Ratchet step for different phones (various max size s). Both axes are log-scale.

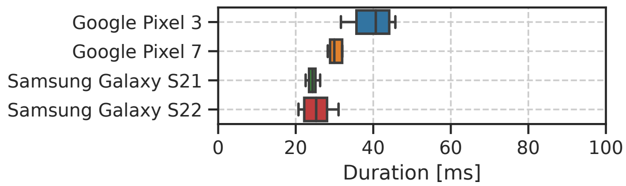

Figure 8: Duration of HIDDENSLOTH random-access decryption without authentication for 1 MiB blocks.

Decryption speed is critical for most application use-cases as it dictates the speed by which stored data, e.g. photos, load. Apps can optimize this step by caching the derived key k and unwrapped $\pi.\mathcal{K}$ in memory after the user has entered their password. For this we implemented authenticated encryption using an encrypt-then-mac regime with AES-CTR and HMAC. This then allows random access to individual blocks of the ciphertext. Our results in Figure 8 show that reading a 1 MiB block generally takes less than 50 ms. The app should verify the authentication tag of the entire storage before performing decryption operations.

7.5 Limitations

Sloth remains vulnerable to physical attacks during usage such as shoulder surfing [10], smudge attacks [3], and side-channel attacks that monitor keyboard entry [37]. However, this is compatible with our threat model and solutions for these are orthogonal to our scheme. In addition, if an app developer allows user-chosen passphrases, then our min-entropy assumption (Section 3.1) is not valid. Therefore, we designed Sloth such that randomly-chosen passphrases are feasible. Since we rely on a hardware-backed secret, device loss means that the secret cannot be recovered. However, apps can introduce complementary backup procedures[5].

8 Related Work

The terminology of key stretching which we also use for our paper was coined in 1999 by Kelsey et al. [26]. Examples of modern password hashing functions include PBKDF2 [23], scrypt [32], and Argon2 [14]. All share the property that an attacker can increase brute-force speed by using more computers. Blocki et al. [15] show in an economic analysis of offline password cracking that key stretching alone does not provide sufficient protection.

Password-authenticated key agreement (PAKE) protocols, such as OPAQUE [22], use interactive protocols with a server to rate-limit password guesses. However, such protocols are not suitable when there is unreliable Internet connectivity. Further, the required network communication leaves traces that rule out their usage for deniable encryption schemes. This also holds for related protocols such as DupLESS [25].

The VeraCrypt project [34] is the most-popular encryption software that offers support plausible deniable encryption for regular computers. It can also boot into different operating systems depending on the entered password when unlocking the machine. Our work could be used to further strengthen password authentication against brute-force attacks. To our knowledge, StegFS [29] was the first practical design of a plausibly-deniable encrypted file system.

For mobile devices, MobiFlage [38] describes a design for providing deniable encryption on Android resisting single-snapshot adversaries. The MobiCeal [18] paper describes an implementation that protects against multi-snapshot adversaries by obfuscating access patterns using “dummy writes”. However, both MobiFlage and MobiCeal require changes to the operating system which renders them impractical for use by developers who wish to support users of standard handsets. The work by Liao et al. [28] describes a theoretical design where some computations for the deniable encryption scheme are run inside an ARM TrustZone environment. Likewise, this design would require significant changes to the operating system and the ability to run custom code within the TEE. Also, none of the above protect against brute-force attacks.

Our work is inspired by a deniable encryption scheme sketched in CoverDrop [1] and shares the idea of storing a separate key inside an SE. However, CoverDrop does not allow for generic key stretching and it is less efficient, as it requires the entire ciphertext to grow to guarantee lower guessing rates. Furthermore, the approach described in CoverDrop does not work on iOS due to the lack of AES-GCM support in Apple’s SE. The paper also does not offer security proofs nor practical evaluations for a representative set of devices.

9 Conclusion

Sloth is family of novel practical key stretching and deniable encryption schemes which leverages the limited throughput of the SE to provide strong security guarantees and favorable parameter choices for standard smartphones. Sloth works without any changes to the operating system and allows for shorter passphrases as the adversary cannot speed up brute-force attacks by using multiple computers. Our survey shows that SEs are more widely available and more practical than is generally assumed. We presented a formal model of SEs which capture their timing characteristics. Instead of asymptotic analysis, we work with absolute duration to provide definite time bounds. This precision allows picking smaller parameters that improve efficiency and usability for the user. We believe that SEs will play a critical role in the security of mobile applications and services, and hope to inspire more research in the area of hardware-assisted security on smartphones.

[5]For example, by adding an additional layer of indirection where the derived secret k encrypts another secret k' that is then used, e.g., for HIDDENSLOTH. The k' can be temporarily decrypted, transferred to another device, and then re-encrypted there using a new derived secret k_2.

Acknowledgements

We would like to thank the anonymous reviewers from previous submissions for their valuable feedback. Daniel Hugenroth is supported by Nokia Bell Labs. This work is partially funded by Mysten Labs.

References

[1] Mansoor Ahmed-Rengers, Diana A Vasile, Daniel Hugenroth, Alastair R Beresford, and Ross Anderson. CoverDrop: Blowing the whistle through a news app. *Proceedings on Privacy Enhancing Technologies*, 2022(2):47–67, 2022.

[2] Amazon. AWS device farm. https://aws.amazon.com/device-farm, 2023.

[3] MD Amruth and K Praveen. Android smudge attack prevention techniques. In *Intelligent Systems Technologies and Applications: Volume 2*, pages 23–31. Springer, 2016.

[4] Apple Inc. Apple platform security - Secure Enclave, 2021. https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web.

[5] Apple Inc. App Store - iOS and iPadOS usage, 2022. https://web.archive.org/web/20230130101543/https://developer.apple.com/support/app-store/,.

[6] Apple Inc. Apple security bounty, 2022. https://security.apple.com/bounty/.

[7] Apple Inc. Apple developer documentation - SecureEnclave, 2023. https://developer.apple.com/documentation/cryptokit/secureenclave.

[8] Apple Inc. Apple developer documentation - SecureEnclave.P256, 2023. https://developer.apple.com/documentation/cryptokit/secureenclave/p256.

[9] Diego F Aranha, Pierre-Alain Fouque, Chen Qian, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Binary elligator squared. In *International Conference on Selected Areas in Cryptography*, pages 20–37. Springer, 2014.

[10] Adam J Aviv, John T Davin, Flynn Wolf, and Ravi Kuber. Towards baselines for shoulder surfing on mobile authentication. In *33rd Annual Computer Security Applications Conference*, pages 486–498, 2017.

[11] Feng Bao, Robert H Deng, and Huafei Zhu. Variations of diffie-hellman problem. In *Information and Communications Security: 5th International Conference, ICICS 2003, Huhehaote, China, October 10-13, 2003. Proceedings 5*, pages 301–312. Springer, 2003.

[12] Mihir Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Crypto*, volume 4117, pages 602–619. Springer, 2006.

[13] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 967–980, 2013.

[14] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[15] Jeremiah Blocki, Benjamin Harsha, and Samson Zhou. On the economics of offline password cracking. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 853–871. IEEE, 2018.

[16] Joseph Bonneau. Deep dive: EFF's new wordlists for random passphrases. *Electronic Frontier Foundation (EFF)*, 2016. Accessed September 2023.

[17] Daniel R. L. Brown. SEC 1: Elliptic Curve Cryptography. Standard, Certicom Research, May 2009. Version 2.0.

[18] 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 *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 454–465. IEEE, 2018.

[19] Google Inc. Hardware Security Best Practices, 2022. https://source.android.com/docs/security/best-practices/hardware.

[20] Google Inc. Android and Google devices security reward program rules, 2022. https://bughunters.google.com/about/rules/6171833274204160/android-and-google-devices-security-reward-program-rules.

[21] Google Inc. Android Keystore system - hardware security module, 2022. https://developer.android.com/training/articles/keystore#HardwareSecurityModule.

[22] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: an asymmetric pake protocol secure against pre-computation attacks. In *Advances in Cryptology–EUROCRYPT*, pages 456–486. Springer, 2018.

[23] Burt Kaliski and A Rusch. RFC 8018: PKCS# 5: Password-based cryptography specification version 2.1, 2017.

[24] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[25] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *22nd USENIX Security Symposium (USENIX Security '13)*, pages 179–194, 2013.

[26] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Information Security: First International Workshop, ISW'97 Tatsunokuchi, Ishikawa, Japan September 17–19, 1997 Proceedings*, pages 121–134. Springer, 2005.

[27] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, volume 6223, pages 631–648. Springer, 2010.

[28] 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.

[29] Andrew D McDonald and Markus G Kuhn. StegFS: A steganographic file system for linux. In *Information Hiding: Third International Workshop, IH'99, Dresden, Germany, September 29-October 1, 1999 Proceedings 3*, pages 463–477. Springer, 2000.

[30] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

[31] Open Worldwide Application Security Project (OWASP). Password storage cheat sheet, 2021. Accessed September 2023.

[32] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. https://www.rfc-editor.org/rfc/rfc7914.html, 2016.

[33] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.

[34] VeraCrypt project. VeraCrypt - free open source disk encryption with strong security for the paranoid, 2023. https://www.veracrypt.fr/en/Home.html.

[35] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

A Survey of SE Availability

The availability of SE functionality for end-user apps is determined by hardware support (i.e. does the device have an SE) as well as platform support for the respective API. We estimate market share for both Apple's iOS devices and Google's Android ecosystem.

A.1 SE Support on Apple devices

As discussed in Section 2, Apple first added Secure Enclaves to their iPhone 5S in 2013 and an API was added in iOS 13 which was released in 2019. Since the iPhone 5S only supported iOS version until 12, all devices with iOS 13 (platform support) also contain a Secure Enclave (hardware support). This is supported by the fact that there is no API to check for the presence of a Secure Enclave. The App Store statistics for May 2022 show that 82% of all iPhones use iOS 15 and 14% use iOS 14 [5]. No data is given for iOS 13 or before. Therefore, at least 96% of all iPhones devices expose SE functionality to developers and are compatible with our Sloth scheme.

A.2 SE Support on Android devices

The situation for the Android ecosystem is more complex as devices are manufactured by different vendors with different hardware as well as changes to the operating system. This can lead to situations where Android devices have an SE, but do not offer API access, or where a device's API is compatible, but has no SE. We use the overall distribution of active Android versions as an upper boundary for platform support. We then execute test code on dozens of real devices to determine hardware support.

For the platform support we use the API Version Distribution that is shown in the "New Project" wizard in the Android Studio IDE. Figure 9 shows the API distribution that is shown when creating new project in the Android Studio IDE when it was last updated on January 6th, 2023. Figure 10 shows the same data for the previous update in August 4th, 2022. Figure 9 shows that 97.2% of devices run API level 23 (Android M) or higher and hence provide the API to check whether a key is backed by a TEE or SE. Furthermore, 81.2% of devices run API level 28 (Android P) or higher and hence provide the StrongBox API that can enforce storage in an SE.

For hardware support, we use the AWS Device Farm [2] which allows remote access to real devices in a datacenter. We compile a test application and upload it for execution on all selected device types. Our test code performs multiple checks. First, we create different key types and then check via `KeyInfo#isInsideSecureHardware` if it is stored in secure hardware. A positive answer indicates that the device has a TEE *or* SE. Then, we read the `PackageManager` feature flag for StrongBox support:

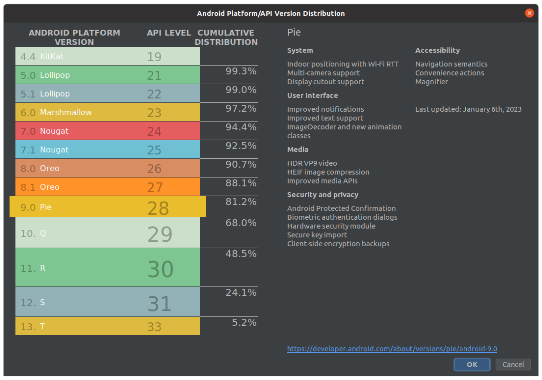

Figure 9: Distribution of Android version as shown in the Android Studio IDE for January 6th, 2023.

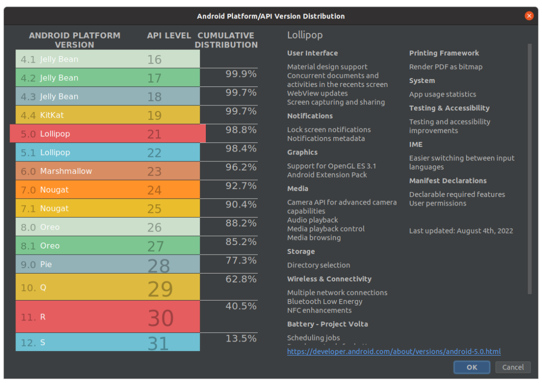

Figure 10: Distribution of Android version as shown in the Android Studio IDE for August 4th, 2022.

FEATURE_STRONGBOX_KEYSTORE. Finally, we create different key types with the setIsStrongBoxBacked property that enforces storing them in an SE and checking for failures. Figure 11 and Table 4 summarize our results.

These results show that the vast majority of devices released after 2020 provide access to an SE. And all of those have at least TEE support. The earliest device with SE support is the Google Pixel 3 which was released 2018. The release year and Android OS version strongly correlate meaning that newer devices indeed run more recent OS versions. In our sample, all devices with OS version 12+ (Android S, API 31) offer an SE. However, based on the available API distribution (Figure 9), only 14% of active handhelds run this OS version. For an estimate of the overall availability, we take the relative presence of SEs for each OS version and weigh it based on the API distribution. This yields an estimate of 45% devices supporting SE globally for data from January 2023 (up from 39% in August 2022). However, this value will differ between countries and we believe it will continue to improve as more device features, such as contactless payments, rely on SEs.

Note that the percentages are cumulative and in order to get

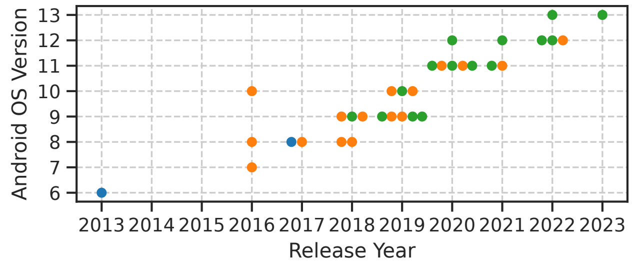

Figure 11: Swarm plot of all surveyed Android devices. We use green • for StrongBox support, orange • for TEE support, and blue • for no support for hardware-backed keys at all.

the actual percentage for a specific version number, e.g. API 9 in January 2023, we have to calculate $81.2\% - 68.0\% = 13.2\%$. For calculating the estimated share of supported devices we first compute the prevalence of SE support per API level using Table 4. This yields the following data: API 9 (50.0%) API 10 (25.0%) API 11 (57.1%) API 12 (80.0%).

We then calculate for January 2023 as follows: $13.2\% \times 50.0\% + 19.5\% \times 25.0\% + 24.4\% \times 57.1\% + 24.1\% \times 80.0\% \approx \mathbf{44.7\%}$. And for August 2022 likewise: $14.5\% \times 50.0\% + 22.3\% \times 25.0\% + 27.0\% \times 57.1\% + 13.5\% \times 80.0\% \approx \mathbf{39.1\%}$.

B Security Proofs

We prove the security of LongSloth (Section 4.1), RainbowSlot (Section 4.2), and HiddenSloth (Section 5.2).

B.1 Security proofs for LongSloth

B.1.1 LongSloth Indistinguishability

Proof. The LongSloth protocol Ξ runs on a SE with HMAC support SE-WITH-HMAC (Definition 4); let SE-OP be the HMAC protocol run by the SE. We prove Theorem 1 by reduction. Let's assume there exists an efficient adversary $\mathcal{A}_\Xi$ against Ξ; we build an adversary $\mathcal{A}$ against SE-OP. $\mathcal{A}$ interacts with a SE-OP-challenger $\mathcal{C}$ and simulates a Ξ-challenger to $\mathcal{A}_\Xi$.

Definition 17 recalls the IND-HMAC$_\mathcal{A}(\lambda)$ experiment played by $\mathcal{A}$ and $\mathcal{C}$. Bellare proved that HMAC is a PRF under the sole assumption that its underlying compression function is a PRF [12]. As in the original paper by Bellare, it is convenient to consider a PRF-adversary $\mathcal{A}$ that takes inputs (Section 3.2 of [12]). Figure 12 illustrates how to leverage $\mathcal{A}_\Xi$ to build an efficient adversary $\mathcal{A}$ breaking the IND-HMAC security of SE-OP (Definition 18).

Definition 17 (IND-HMAC Experiment [12])**.** Let λ be a fixed security parameter. Let $\mathcal{A}$ be a WT adversary with wall

Device Model	OS	Release	TEE	SE
ASUS Nexus 7 - 2nd Gen (WiFi)	6	2013		
Google Pixel	7	2016	✓	
Google Pixel 2	8	2017	✓	
Google Pixel 3	9	2018	✓	✓
Google Pixel 4 (Unlocked)	10	2019	✓	✓
Google Pixel 4a	11	2020	✓	✓
Google Pixel 5 (Unlocked)	12	2020	✓	✓
Google Pixel 6 (Unlocked)	12	2021	✓	✓
Google Pixel 7	13	2022	✓	✓
LG Stylo 5	9	2019	✓	
OnePlus 8T	11	2020	✓	
Samsung A51	10	2019	✓	
Samsung Galaxy A10s	10	2019	✓	
Samsung Galaxy A13 5G	11	2021	✓	
Samsung Galaxy A40	9	2019	✓	
Samsung Galaxy A7	8	2016	✓	
Samsung Galaxy A71	11	2020	✓	
Samsung Galaxy J7 (2018)	8	2018	✓	
Samsung Galaxy Note 10	9	2019	✓	✓
Samsung Galaxy Note20	11	2020	✓	✓
Samsung Galaxy S10	9	2019	✓	✓
Samsung Galaxy S21 Ultra	11	2021	✓	✓
Samsung Galaxy S22 5G	12	2022	✓	✓
Samsung Galaxy S23	13	2023	✓	✓
Samsung Galaxy S8 (T-Mobile)	8	2017		
Samsung Galaxy S9 (Unlocked)	9	2018	✓	
Samsung Galaxy Tab A 10.1	10	2016	✓	
Samsung Galaxy Tab S4	8	2018	✓	
Samsung Galaxy Tab S6 (WiFi)	9	2019	✓	✓
Samsung Galaxy Tab S7	11	2020	✓	✓
Samsung Galaxy Tab S8	12	2022	✓	✓
Sony Xperia XZ3	9	2018	✓	
Xiaomi 12 Pro	12	2022	✓	

Table 4: All 33 surveyed devices from the AWS device farm. The *Device Model* is the name provided by the AWS API. The *OS Version* referes to the tested OS version and the device might be available with different versions.

time budget B and $\mathcal{C}$ a WT challenger. The SE-OP indistinguishability experiment IND-HMAC$_{\mathcal{A}}(\lambda)$ is defined as follows:

1. Let the state ψ be freshly initialized and h an arbitrary (but fixed) key handle. $\mathcal{C}$ randomly samples a secret key $k \leftarrow$ SE.HMACKEYGEN(ψ, h) and sets $\psi.h \leftarrow k$.
2. $\mathcal{A}$ receives oracle access SE.HMAC under the WT conditions (Definition 3).
3. $\mathcal{A}$ submits two chosen plaintexts $(m_0, m_1) \in M^2$ to $\mathcal{C}$.
4. $\mathcal{C}$ randomly samples $b \xleftarrow{\$} \{0,1\}$, and provides $\mathcal{A}$ with $c_0 \leftarrow$ SE.HMAC(ψ, h, m) if $b = 0$ or $c_1 \xleftarrow{\$} \{0,1\}^\lambda$ otherwise.
5. $\mathcal{A}$ outputs a bit b' and wins iff $b = b'$.
6. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

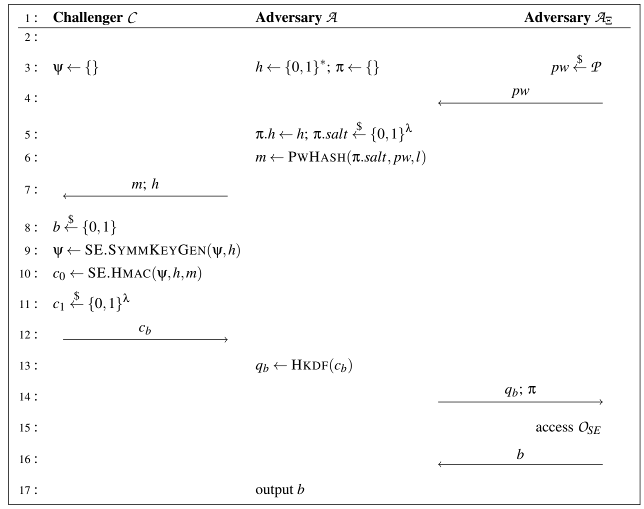

Figure 12: LongSloth security reduction. $\mathcal{A}$ leverages the efficient adversary $\mathcal{A}_\Xi$ to play against $\mathcal{C}$ and break the IND-HMAC security of SE-OP.

Definition 18 (IND-HMAC Security [12])**.** A SE-OP protocol is IND-HMAC secure if for all WT adversaries $\mathcal{A}$ with time budget B, there is a function NEGL such that for all λ,

$$Pr[\text{IND-HMAC}_{\mathcal{A}}(\lambda) = 1] \leq \frac{1}{2} + \text{NEGL}(\lambda)$$

The challenger $\mathcal{C}$ starts by initializing its internal state ψ with a secret key and the adversary $\mathcal{A}_\Xi$ selects a m-entropy secure password pw. $\mathcal{A}_\Xi$ sends pw to $\mathcal{A}$ who uses it to generate a message for the challenger $\mathcal{C}$; it sets $m \leftarrow$ PWHASH$(salt, pw, l)$ (where $salt$ is a random salt). Challenger $\mathcal{C}$ samples a random bit. If $b = 0$ it provides $\mathcal{A}$ with the HMAC of m, i.e., $c_0 \leftarrow$ HMAC(ψ, h, m); otherwise it samples a random bit string $c_1 \xleftarrow{\$} \{0,1\}^\lambda$ of the same size as the HMAC output. In order to guess whether c_b is the output of a HMAC or a random bit string, the adversary $\mathcal{A}$ provides $\mathcal{A}_\Xi$ with the stretched key $q_b \leftarrow$ HKDF(c_b). $\mathcal{A}_\Xi$ determines whether q_b originated from its password pw or a random source. To this purpose, $\mathcal{A}_\Xi$ can access the oracle O_{SE} under the WT conditions (see Definition 3). It finally returns $b = 0$ if it believes q_b originated from pw and $b = 1$ otherwise. Finally, the adversary $\mathcal{A}$ deduces that $\mathcal{C}$ computed the HMAC of m if $b = 0$ and that it sampled a random bit string if $b = 1$.

We observe that $\mathcal{A}$ wins the IND-HMAC$_{\mathcal{A}}(\lambda)$ experiment with the same probability as $\mathcal{A}_\Xi$ wins KEYIND$_{\mathcal{A}_\Xi}(\lambda, \mathcal{P})$. As a result, the existence of an efficient adversary $\mathcal{A}_\Xi$ winning KEYIND$_{\mathcal{A}_\Xi}$ with probability $p > 1/2 + \text{NEGL}$ implies the existence of an efficient adversary $\mathcal{A}$ winning IND-HMAC$_{\mathcal{A}}(\lambda)$ with the same probability p. This directly violates the assumption that SE-OP runs a IND-HMAC secure HMAC algorithm, hence a contradiction. □

B.1.2 LongSloth Hardness

Proof. We constructively proof Theorem 3 by deriving the success rate of $\mathcal{A}$ for a given wall time budget. For this we first note that k only allows $\mathcal{A}$ to verify its guesses, but provides no helpful information otherwise. This is because the pre-image resistance of HKDF implies $\mathcal{A}$ does not learn any information about ω_{post} (and thus any of the previous state) from k. Hence, k provides $\mathcal{A}$ with no advantage when choosing pw' candidates.

Second, we show that $\mathcal{A}$ has to consider each pw' candidate independently. For this we observe that the output $\omega_{pre} = \text{PWHASH}(\pi.salt, pw')$ is indistinguishable from random as per our assumptions, i.e. the input of SE.HMAC is independent for each pw. We note that the salt sampled as per Ξ.KEYGEN rules out any pre-computations by $\mathcal{A}$.

Third, we show that $\mathcal{A}$ has to pay the full costs $(l * c_{\text{HMAC}})$ for each of their guesses. This follows from the *existential unforgeability under adaptive chosen-messages* [24, p.113] that we can assume for SE.HMAC. In particular, $\mathcal{A}$ does not gain any information from submitting a prefix of ω_{pre}. With the pre-image resistance of HKDF, this implies that $\mathcal{A}$ must perform all steps of the Ξ.DERIVE(ψ, pw', h) using oracle O_{SE}. Hence, one password guess reduces $(l * c_{\text{HMAC}})$ units from the adversary's budget B.

Fourth, we derive the probability of success for a set password guesses. Let X be a random variable drawn from the distribution $\mathcal{P}$ by the challenger to determine the password pw. Assume to the advantage of $\mathcal{A}$ that they can efficiently sample a set G of passwords from $\mathcal{P}$ with each $g_i \in G$ independently drawn from $\mathcal{P}$ as random variable Y with $g_i \neq g_j \forall i,j$. Then the probability that one of the guesses allows $\mathcal{A}$ to derive k is: $Pr[\mathcal{A}\ wins] = \sum_{pw \in \mathcal{P}} \sum_{g \in \mathcal{G}} Pr[X = pw] \cdot Pr[Y = pw'] = \sum_{pw \in \mathcal{P}} Pr[X = pw] \cdot \sum_{g \in \mathcal{G}} Pr[Y = pw'] = |G| \cdot Pr[Y = pw'] \leq \frac{|G|}{2^m}$ Based on our arguments above, the maximum size of G that the adversary can verify given budget B is $|G| = \frac{B}{(l \cdot c_{\text{HMAC}})}$. Substituting in the previous equation yields: $Pr[\text{KEYHARD}_{A,\Xi}(\lambda, \mathcal{P}) = 1] \leq \frac{B}{(l \cdot c_{\text{HMAC}})} \cdot \frac{1}{2^m}$. □

B.2 Security proofs for RainbowSloth

B.2.1 RainbowSloth Indistinguishability

Proof. The RainbowSloth protocol Ξ runs on a SE with symmetric encryption support SE-WITH-ECDH (Definition 5); let SE-OP be the DDH secure Diffie-Hellman key exchange run by the SE. We prove Theorem 2 by reduction. Let's assume there exists an efficient adversary $\mathcal{A}_\Xi$ against Ξ; we build an adversary $\mathcal{A}$ against SE-OP. $\mathcal{A}$ interacts with a SE-OP-challenger $\mathcal{C}$ and simulates a Ξ-challenger to $\mathcal{A}_\Xi$.

Challenger $\mathcal{C}$		**Adversary $\mathcal{A}$**		**Adversary $\mathcal{A}_\Xi$**
		$h \leftarrow \{0,1\}^*$; $\pi \leftarrow \{\}$		$pw \xleftarrow{\$} \mathcal{P}$
			$\xleftarrow{pw}$	
		$\pi.h \leftarrow h$; $\pi.salt \xleftarrow{\$} \{0,1\}^\lambda$		
		$\omega_{user} \leftarrow \text{PWHASH}(\pi.salt, pw, l)$		
		$\bar{g} \leftarrow \text{HASHTOP256}(\text{KDF}(\omega_{user}))$		
	$\xleftarrow{\bar{g}}$			
$b \xleftarrow{\$} \{0,1\}$; $r \xleftarrow{\$} \mathbb{Z}_q$				
$\bar{u_0} \leftarrow (g_1^r, \ldots, g_k^r) \in \mathbb{G}^k$				
$\bar{u_1} \xleftarrow{\$} \in \mathbb{G}^k$				
	$\xrightarrow{\bar{u_b}}$			
		$q_b \leftarrow \text{HKDF}(\bar{u_b})$		
			$\xrightarrow{q_b;\ \pi}$	
				access O_{SE}
			$\xleftarrow{b}$	
		output b		

Figure 13: RainbowSloth security reduction. $\mathcal{A}$ leverages the efficient adversary $\mathcal{A}_\Xi$ to play against $\mathcal{C}$ and break the generalized DDH assumption.

Definition 19 recalls the (generalized) decisional Diffie-Hellman (DDH) [11] experiment played by $\mathcal{A}$ and $\mathcal{C}$. Figure 13 illustrates how to leverage $\mathcal{A}_\Xi$ to build an adversary $\mathcal{A}$ breaking the DDH assumption (Definition 20).

Definition 19 (Generalized DDH [11])**.** Let λ be a fixed security parameter. Let n be an integer and $\mathbb{G}$ a large cyclic group of prime order q. Let $\mathcal{A}$ be a WT adversary with wall time budget B and $\mathcal{C}$ a WT challenger. The generalized DDH indistinguishability experiment $\text{DDH}_{\mathcal{A}}(\lambda)$ is defined as follows:

1. $\mathcal{A}$ provides $\mathcal{C}$ with $(g_1, \ldots, g_n) \in \mathbb{G}^n$.
2. $\mathcal{C}$ randomly samples $b \xleftarrow{\$} \{0,1\}$. If $b = 0$ it randomly samples $r \xleftarrow{\$} \mathbb{Z}_q$ and sets $\bar{u_0} = (g_1^r, \ldots, g_n^r) \in \mathbb{G}^n$; otherwise it randomly samples $\bar{u_1} = (u_1, \ldots, u_n) \in \mathbb{G}^n$. It then provides $\mathcal{A}$ with $\bar{u_b}$.
3. $\mathcal{A}$ outputs a bit b' and wins iff $b = b'$.
4. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

Definition 20 (Generalized DDH [11])**.** The generalized decisional Diffie-Hellman assumption holds in $\mathbb{G}$ if for any n, there is a function NEGL such that,

$$Pr[\text{DDH}_{\mathcal{A}} = 1] \leq \frac{1}{2} + \text{NEGL}(\lambda)$$

Figure 13 shows how to leverage the RainbowSloth adversary $\mathcal{A}_\Xi$ to build an efficient adversary breaking the the generalized DDH assumption (Definition 20).

□

B.2.2 RainbowSloth Hardness

Proof. The proof for RainbowSloth is analogous to the one for LongSloth (see §B.1.2) with the cost of the critical operation exchanged for $n \cdot c_{\text{ECDH}}$ as per Definition 5. □

B.3 Security proof of HiddenSloth

Both 1S-HiddenSloth and MS-HiddenSloth run on top of a key stretching scheme such as LongSloth (Section 4.1) and RainbowSloth (Section 4.2); as a result, they run on a SE with either HMAC or ECDH support. As mentioned in Section 5.1 and Section 5.2, any HiddenSloth protocol Δ requires an authenticated IND-CPA secure stream cipher AE running in the user space (i.e., outside of the SE).

B.3.1 MS-HiddenSloth Indistinguishability

Proof. We prove Theorem 5 by reduction. Let's assume there exists an efficient adversary $\mathcal{A}_\Delta$ against Δ; we build an adversary $\mathcal{A}$ against AE. $\mathcal{A}$ interacts with a AE-challenger $\mathcal{C}$ and simulates a Δ-challenger to $\mathcal{A}_\Delta$. Definition 21 recalls the IND-CPA experiment played by $\mathcal{A}$ and $\mathcal{C}$; Definition 22 recalls the definition of AE's IND-CPA security.

Definition 21 (AE IND-CPA Experiment)**.** Let λ be a fixed security parameter. Let $\mathcal{A}$ be a WT adversary with wall time budget B_λ and $\mathcal{C}$ a WT challenger. The AE indistinguishability experiment IND-CPA$_\mathcal{A}(\lambda)$ is defined as follows:

1. $\mathcal{C}$ randomly samples a secret key $k \leftarrow$ AE.KEYGEN.
2. $\mathcal{A}$ receives oracle access AE.ENC under the WT conditions (Definition 3).
3. $\mathcal{A}$ submits two chosen plaintexts $(m_0, m_1) \in M^2$ to $\mathcal{C}$.
4. $\mathcal{C}$ randomly samples $b \xleftarrow{\$} \{0,1\}$ and $iv \xleftarrow{\$} IV$, and provides $\mathcal{A}$ with $c_b, t_b \leftarrow \text{AE.ENC}(k, iv, m_b)$.
5. $\mathcal{A}$ receives oracle access AE.ENC under the WT conditions.
6. $\mathcal{A}$ outputs a bit b' and wins iff $b = b'$.
7. The experiment returns 1 iff $\mathcal{A}$ wins, otherwise 0.

Definition 22 (AE IND-CPA)**.** A stream cipher AE is IND-CPA secure if for all WT adversaries $\mathcal{A}$ with time budget B, there is a function NEGL such that for all λ,

$$Pr[\text{IND-CPA}_\mathcal{A}(\lambda) = 1] \leq \frac{1}{2} + \text{NEGL}(\lambda)$$

It is convenient to grant $\mathcal{A}$ access to a SE oracle allowing to perfectly simulate SE symmetric encryption operations *SE.SymmEnc* as performed by a SE with symmetric encryption support SE-WITH-SYMMENC (Definition 6). This oracle is described in Algorithm 6 and is initialized calling O.INIT before starting the experiment.

Figure 14 illustrates how to leverage the MS-HiddenSloth adversary $\mathcal{A}_\Delta$ to build an efficient adversary $\mathcal{A}$ breaking the IND-CPA security of AE (Definition 22). Figure 14 generates a random key k calling AE.KEYGEN. The experiment should ideally generate k through a Sloth key stretching algorithm but Theorems 1 and 2 state that $\mathcal{A}_\Delta$ cannot distinguish a key generate by a Sloth algorithm from a purely random key.

During the preparation phase (lines 7 to 21 of Figure 14), the adversary $\mathcal{A}_\Delta$ provides $\mathcal{A}$ with a message $\tilde{m} \leftarrow M$ and

Algorithm 6 Oracle simulating *SE.SymmEnc* operations as performed by a SE-WITH-SYMMENC.

```
procedure O.INIT()
    ψ ← {}
procedure O.ENCRYPT(h′, m)
    ψ ← SE.SYMMKEYGEN(ψ, h′)
    iv ←$ IV
    return SE.SYMMENC(ψ, h′, iv, m)
```

a bit $\tilde{b}$. $\mathcal{A}$ makes use of it oracle access to AE.ENC (step 2 of Definition 21) by forwarding $\tilde{m}$ to the challenger $\mathcal{C}$. $\mathcal{C}$ encrypts $\tilde{m}$ and replies with the corresponding iv iv, ciphertext $\tilde{c}$, and tag $\tilde{t}$. $\mathcal{A}$ stores the queried message as $s_{\tilde{b}} \leftarrow \tilde{m}$; it will later use it to simulate the state of the protocol Δ to $\mathcal{A}_\Delta$. It then sample a fresh key tk to re-encrypt $\tilde{c}$, which simulates the outer encryption layer of Algorithm 5. $\mathcal{A}$ makes use of its SE encryption oracle (defined in Algorithm 6) to encrypt tk, $\tilde{t}$, and iv. $\mathcal{A}$ now holds all information to craft a state π as if generated by a Δ-challenger. The preparation phase repeats a number of times chosen by $\mathcal{A}_\Delta$ (under WT conditions).

$\mathcal{A}$ eventually prepares two messages for the challenger $\mathcal{C}$: $m_0 \leftarrow s_0$ and $m_1 \leftarrow s_1$. The values s_0 and s_1 retain the latest messages queried by $\mathcal{A}_\Delta$ during the preparation phase for $\tilde{b} = 0$ and $\tilde{b} = 1$, respectively. The challenger $\mathcal{C}$ encrypts either m_0 or m_1 based on a random bit b and provides $\mathcal{A}$ with the corresponding ciphertext and tag $c_b, t_b \leftarrow AE.Enc(k, iv, m_b)$ and iv iv. At this point, c_b is the encryption of the latest pair $(\tilde{m}, b)$ queried by $\mathcal{A}_\Delta$ during the preparation phase (at Line 7). $\mathcal{A}$ finally re-encrypt those information to simulate the outer encryption layer similarly to the preparation phase.

$\mathcal{A}_\Delta$ accesses the oracle O_{SE} and eventually determines whether π contains the encryption of the latest pair $(\tilde{m}_0, 0)$ or $(\tilde{m}_1, 1)$ it submitted during the preparation phase. It eventually returns $b = 0$ if it believes π contains the encryption of $\tilde{m}_0$ and $b = 1$ otherwise. Finally, the adversary $\mathcal{A}$ deduces that $\mathcal{C}$ encrypted m_0 if $b = 0$ and m_1 if $b = 1$.

We observe that $\mathcal{A}$ wins the IND-CPA$_\mathcal{A}(\lambda)$ experiment with the same probability as $\mathcal{A}_\Delta$ wins experiment DE-MS-IND$_{\mathcal{A}_\Delta}(\lambda, \mathcal{P})$. As a result, the existence of an efficient adversary $\mathcal{A}_\Delta$ winning experiment DE-MS-IND$_{\mathcal{A}_\Delta}(\lambda, \mathcal{P})$ with probability $p > 1/2 + \text{NEGL}$ implies the existence of an efficient adversary $\mathcal{A}$ winning IND-CPA$_\mathcal{A}(\lambda)$ with the same probability p. This directly violates the assumption that AE is IND-CPA secure, hence a contradiction. □

B.3.2 MS-HiddenSloth Hardness

Proof. We prove Theorem 6 by showing that the probability that adversary $\mathcal{A}$ wins the deniable encryption hardness experiment for protocol Δ is no bigger than its probability of winning the hardness experiment against its underlying key

Challenger $\mathcal{C}$		Adversary $\mathcal{A}$		Adversary $\mathcal{A}_\Delta$
$k \leftarrow \text{AE.KeyGen}()$		$s_0 \leftarrow 0;\ s_1 \leftarrow 0$		
		$h' \leftarrow \{0,1\}^*\|\|1$		
				$\tilde{m} \leftarrow M;\ \tilde{b} \leftarrow \{0,1\}$
			$\xleftarrow{\tilde{m};\ \tilde{b}}$	
		$s_{\tilde{b}} \leftarrow \tilde{m}$		
	$\xleftarrow{\tilde{m}}$			
$iv \xleftarrow{\$} IV$				
$\tilde{c},\tilde{t} \leftarrow \text{AE.Enc}(k,iv,\tilde{m})$				
	$\xrightarrow{\tilde{c};\ \tilde{t};\ iv}$			
		$tk \leftarrow \text{AE.KeyGen}()$		
		$\pi \leftarrow \{\};\ tiv \xleftarrow{\$} IV$		
		$\pi.blob, \pi.ttag \leftarrow \text{AE.Enc}(tk,tiv,\tilde{c})$		
		$\pi.tiv \leftarrow tiv;\ \mathcal{K} \leftarrow [tk,t_b,iv]$		
		$\pi.\mathcal{K} \leftarrow \text{O.Encrypt}(h',\mathcal{K})$		
			$\xrightarrow{\pi}$	
		$m_0 \leftarrow s_0;\ m_1 \leftarrow s_1$		
	$\xleftarrow{m_0;m_1}$			
$b \xleftarrow{\$} \{0,1\};\ iv \xleftarrow{\$} IV$				
$c_b,t_b \leftarrow \text{AE.Enc}(k,iv,m_b)$				
	$\xrightarrow{c_b;\ t_b;\ iv}$			
		$tk \leftarrow \text{AE.KeyGen}()$		
		$\pi \leftarrow \{\};\ tiv \xleftarrow{\$} IV$		
		$\pi.blob, \pi.ttag \leftarrow AE.Enc(tk,tiv,c_b)$		
		$\pi.tiv \leftarrow tiv;\ \mathcal{K} \leftarrow [tk,t_b,iv]$		
		$\pi.\mathcal{K} \leftarrow \text{O.Encrypt}(h',\mathcal{K})$		
			$\xrightarrow{\pi}$	
				access O_{SE}
			$\xleftarrow{b}$	
		output b		

Figure 14: MS-HiddenSloth security reduction. $\mathcal{A}$ leverages the efficient adversary $\mathcal{A}_\Delta$ to play against $\mathcal{C}$ and break the IND-CPA security of AE.

stretching scheme Ξ. That is, $Pr[\text{KeyHard}_{\mathcal{A},\Delta}(\lambda,\mathcal{P}) = 1] \leq Pr[\text{KeyHard}_{\mathcal{A},\Xi}(\lambda,\mathcal{P}) = 1]$.

First, $\mathcal{A}$ removes the outer encryption layer calling SE.SymmDec and AE.Dec (Line 25 and Line 26 of Algorithm 5). As a result, $\pi.blob$ contains the encryption of *data*, encrypted using a key derived from *pw*. This operations requires a one-time cost of $T_{\text{SE.SymmEnc}}$ from the adversary's budget B and does not need to be repeated for each guess.

Second, we observe that π (and $\pi.blob$ in particular) only allows $\mathcal{A}$ to verify their guesses, but provides no helpful information otherwise. This observation follows from the IND-CPA resistance of the underlying AE encryption scheme. Hence, knowing π provides $\mathcal{A}$ with no advantage when choosing pw' candidates.

Third, we show that $\mathcal{A}$ has to pay the full cost c_Ξ required to access the SE and run the underling key stretching scheme Ξ used by $\Delta.\text{Decrypt}$. $\mathcal{A}$ must find a key k such that $\text{AE.Dec}(k, \pi.iv, \pi.blob, \pi.tag)$ returns *data* (Line 16 of Algorithm 4). Assuming AE is a permutation-based cipher, there is a single k satisfying this property. Let's assume (to the advantage of the adversary) that $\mathcal{A}$ can brute-force AE.Dec to recover k (since it does not require access to the SE and thus does not cost $\mathcal{A}$'s budget). $\mathcal{A}$ must now win the key stretching hardness experiment presented in Definition 10 against a Ξ-challenger. Assuming Ξ is σ_Ξ-hard, Definition 11 indicates $\mathcal{A}$ pays budget $c_\Xi = \sigma_\Xi$ for each of their guesses.

Finally, since the adversary pays a one-time cost $T_{\text{SE.SymmEnc}}$ and a cost of σ_Ξ for each of their guesses, the overall cost of guessing *pw* is $\sigma > \sigma_\Xi$ per guess. It follows that $\frac{B}{\sigma \cdot 2^m} < \frac{B}{\sigma_\Xi \cdot 2^m}$, and thus $Pr[\text{DeHard}_{\mathcal{A},\Delta}(\lambda,\mathcal{P}) = 1] \leq Pr[\text{KeyHard}_{\mathcal{A},\Xi}(\lambda,\mathcal{P}) = 1]$. $\square$