Abstract

Keyloggers are a prominent class of malicious software that surreptitiously logs all the user activity. Traditional approaches aim to eradicate this threat by either preventing or detecting their deployment. In this paper, we take a new perspective to this problem: we explore the possibility of tolerating the presence of a keylogger, while making no assumption on the keylogger internals or the system state. The key idea is to confine the user keystrokes in a noisy event channel flooded with artificially generated activity. Our technique allows legitimate applications to transparently recover the original user keystrokes, while any deployed keylogger is exposed to a stream of data statistically indistinguishable from random noise. We evaluate our solution in realistic settings and prove the soundness of our noise model. We also verify that the overhead introduced is acceptable and has no significant impact on the user experience.

1 Introduction

Privacy-breaching malware is a class of malicious software that discloses sensitive user data to third parties. Keyloggers are one of the most serious examples of this class, given their ability to surreptitiously log all the user activities [6]. Despite significant research and commercial effort, keyloggers still pose an important threat to users. This threat is further exacerbated by the surge of user-space keyloggers [4, 9], which are easy to implement and can be executed with no special permission. For this reason, our work explicitly focuses on this class of keyloggers. Current approaches aim to counter this threat by either preventing their deployment or detecting their execution [9, 1, 2, 14, 5].

To the user, however, all these approaches are merely informative, which means that they do not offer a practical solution in case of positive detection. Users of the Windows operating system are well-aware of the challenges posed by removing a piece of malicious code: the subverted OS facilities are generally so numerous that Anti-virus programs regularly bail out of the removal phase and instead point the user to security bulletins with step-by-step removal instructions. In addition, many are the cases in which users have insufficient permissions to perform a complete removal of the malicious application. For example, many companies provide their employees with only non-administrative user accounts. The situation is even more problematic for users temporarily accessing untrusted machines, e.g., Internet cafés. In this scenario, the user is literally entrusting his private data to strangers who may or may not honor his trust.

To address these concerns, a number of commercial solutions [13, 11] have been recently proposed. The general idea is to encrypt the keystrokes before they leave the kernel and decrypt them upon arrival at the intended user application. This approach has two fundamental limitations. First, it requires a new kernel module which can only be installed with privileged rights. Second, it does not attempt to hide or disguise the typing dynamics of the user. Unfortunately, keystroke dynamics have been proved to be sufficiently accurate to crack passwords, with surprisingly good results in case of dictionary-based attacks for the English language [15].

In this paper we introduce NoisyKey, the first unprivileged and statistically sound approach to tolerate the presence of a user-space keylogger. Unlike previous methods [13, 11], we confine the user keystrokes in a noisy event channel by artificially generating dummy keystroke data. Our technique exposes the noisy keystroke stream only to keyloggers, while allowing legitimate applications to transparently recover the original data. The generation of dummy keystrokes is backed by a privacy model that ensures that the resulting stream is indistinguishable from random noise. The key idea is to adopt a predetermined reference keystroke distribution and adaptively generate dummy keystroke data such that the combination of the user activity and the generated noise always matches the reference distribution.
The result is that details on the original user activity are no longer exposed to the adversary. We prototyped our technique in a lightweight library that does not require any privilege to be deployed. To verify the effectiveness, we evaluated our prototype against a real dataset of user inputs [7]. Our experiments show that our technique successfully eliminates any evidence of the original user behavior from the overall keystroke distribution, and only impacts marginally the user experience.

2 Our Approach

The key idea is to transparently flood with dummy data the event channel used to deliver the user keystrokes to the intended application. If the generated noise is indistinguishable from user activity, any malicious application peeking on the same channel will only eavesdrop a random stream of data, with no means to recover the original keystrokes.

On Windows, a single keystroke leaving the kernel is first delivered to the process csrss.exe. Its task is to reroute the user input to the intended application. At destination, the keystroke is handled by the GUI application thread, which eventually calls user32.d11 to update the user interface. Figure 1 depicts all the components involved in the process and highlights those that can potentially be subverted by user-space keyloggers. We have verified this claim by analyzing all the samples available at [12] and found no exception. Our solution is implemented in the library noisykey.d11. Although it does not mingle with the internals of the event channel, it does control its ends (a) and (b). On one side, it injects well-crafted noise in the form of dummy keystrokes. On the other, it removes the noise before it reaches the graphical routines included in user32.d11. Unlike alternative approaches establishing a separate event channel, our approach is entirely unprivileged. Also, deploying our solution does not require the user to recompile or restart the application, but, as we later explain, is completely online. To preserve keyboard shortcut functionalities, our solution explicitly handles well-known hotkey modifiers (i.e., CTRL, ALT, WIN, and SHIFT). Dead-keys and user-defined shortcuts, in turn, can be explicitly white-listed by the user. Finally, to minimize the impact on the performance, noisykey.d11 automatically interrupts its activities when the target application is not on-focus.

Architecture. The architecture of our solution, depicted in Figure 2, comprises four different components: the Noise Factory, the Normalizer, the Injector Thread, and the Silencer Thread. The Noise Factory is a repository of dummy keystroke sequences. These sequences have to be well-forged in order to mimic human-like keystroke dynamics. The Normalizer acts as middle-man between the Noise Factory and the Injector Thread. Its goal is to generate context-aware noise, shaping the dummy data according to the user activity. The importance of context awareness is immediately evident when we consider the case of a user typing a credit card number. A context-agnostic noise may not include any digits at all, allowing a context-aware attacker to easily recover the original data. The last two components are two loosely synchronized threads designed to inject the dummy keystrokes into the event channel and subsequently exfiltrate the original user activity. The Injector Thread completes an iteration every \( I_\text{f} \) ms and injects the dummy keystrokes using the API \texttt{keybd_event()}.

Normalizer

Keystroke Model

We model a keystroke using the following factors: (i) scancode (i.e., the code associated to the \texttt{keydown} event), (ii) typing timestamp (i.e., the timestamp of the \texttt{keydown} event), (iii) user activity found.
event), and (iii) hold time (the time between the keydown and the keyup event). Note that a typable symbol does not necessarily correspond to a visible character. For instance, the capital letter A is obtained by the combination of scan codes \( \text{SHIFT} + a \). Our model merges these combinations in a single scan code defined over the alphabet of typable symbols \( \Sigma \). To model the typing timestamp and the hold time, we assume a discretized time model, i.e., \( \mathbb{T} = \{ t_0, \ldots, t_n \} \). More formally:

**Definition 1.** A keystroke is a triple, \( (k, t, h) \) represented by the scan code \( k \in \Sigma \), the typing timestamp \( t \in \mathbb{T} \), and the hold time \( h \in \mathbb{T} \).

User-generated keystrokes are generally issued in logically related sequences, e.g., passwords, usernames, and credit card numbers. Thus:

**Definition 2.** A keystroke sequence \( S \) is a set of \( n \) triples, \( S = \{ (k_0, t_0, h_0), \ldots, (k_n, t_n, h_n) \} \), where \( 0 \leq i \leq n \).

To reason over the typing dynamics of a keystroke sequence, we adopt a probabilistic approach, with timing information modeled by random variables. In particular, given a generic keystroke sequence \( S \), we model the number of keystrokes pressed over a time interval \( T_{i,j} = \{ t \mid t_i \leq t \leq t_j \} \) using the random variable \( X_S(T_{i,j}) \). This allows us to quantitatively model the typing dynamics of any given keystroke sequence. To qualitatively model the typing dynamics, we resort to a second random variable, \( Y_{S,T_{i,j}}(k) \), which, given \( S \) and \( T_{i,j} \), measures the number of keystrokes with scan code \( k \) issued in the time interval. The random variable \( Z_{S,T_{i,j}}(k,h) \), finally, measures the number of keystrokes with scan code \( k \) and hold time \( h \) issued in the time interval. More formally:

**Definition 3.** Let \( S \) be a keystroke sequence of length \( n \), and \( T_{i,j} \) a generic time interval. Then:

The function \( f_S : \Sigma \times \mathbb{T} \times \mathbb{T} \rightarrow \{0,1\} \) determines if a keystroke, as identified by its triple, is part of the keystroke sequence \( S \), where the \( \in \) operator treats nil arguments as wildcards:

\[
f_S(k,t,h) = \begin{cases} 1 & \text{if } (k,t,h) \in S \\ 0 & \text{otherwise} \end{cases}
\]

The random variable \( X_S(T_{i,j}) \) counts the number of keystrokes issued in \( T_{i,j} \):

\[
X_S(T_{i,j}) = \sum_{t_j \in T_{i,j}} f_S(nil,t_l,nil)
\]

The random variable \( Y_{S,T_{i,j}}(k) \) counts the number of keystrokes with scan code \( k \) in \( T_{i,j} \):

\[
Y_{S,T_{i,j}}(k) = \sum_{t_j \in T_{i,j}} f_S(k,t_l,nil)
\]

The random variable \( Z_{S,T_{i,j}}(k,h) \) counts the number of keystrokes with scan code \( k \) and hold time \( t \) in \( T_{i,j} \):

\[
Z_{S,T_{i,j}}(k,h) = \sum_{t_j \in T_{i,j}} f_S(k,t_l,h)
\]

### 4 Privacy Model

Our privacy model is based on the work of Pfitzmann et al. [10]. Their work defines a consolidated terminology for privacy properties in the context of distributed systems where senders and receivers (i.e., actors), are assumed to exchange messages (i.e., items of interest). Our setting is slightly simplified, as we do not have to consider different actors. Under the assumption that the adversary has no way to tell real and dummy keystrokes apart, we consider just one sender, the keyboard, and one receiver, the application. Our goal is to generate dummy events so that the original keystrokes and their dynamics are no longer explicitly exposed to the adversary.

In messaging contexts, the highest grade of privacy is named undetectability, namely that an adversary has no ability to identify a real message among other (dummy) messages. This definition swiftly translates to keystrokes. We say that a keystroke sequence (i.e., the user activity) is undetectable if an adversary has no ability to discriminate it from other dummy keystroke sequences. Following the findings in [8], we define the undetectability property in terms of behavioral similarities between the random variables that describe the sensitive items of interests, i.e., the keystroke sequences.

**Privacy-definition 1.** Given a statistical test \( \mathcal{T} \), two random variables, \( R_1 \) and \( R_2 \), are said to be \( \alpha \)-undetectable with respect to each other, i.e., \( R_1 \approx_\alpha R_2 \), if the null hypothesis that their two datasets are from different distributions is rejected by \( \mathcal{T} \) with confidence \( 1 - \alpha \).

**Privacy-definition 2.** Given a time interval \( T_{i,j} \), two keystroke sequences \( S_1, S_2 \) are \( \alpha \)-undetectable with respect to each other, i.e., \( S_1 \approx_\alpha S_2 \), if all their random variables are \( \alpha \)-undetectable given any scan code \( k \in \Sigma \), and hold time \( h \in \mathbb{T} \):

\[
X_{S_1}(T_{i,j}) \approx_\alpha X_{S_2}(T_{i,j}) \\
Y_{S_1,T_{i,j}}(k) \approx_\alpha Y_{S_2,T_{i,j}}(k) \\
Z_{S_1,T_{i,j}}(k,h) \approx_\alpha Z_{S_2,T_{i,j}}(k,h)
\]

In the ideal case of constant human activity, i.e., random variables with a steady underlying distribution, our goal would be to inject dummy keystroke sequences that yield identical random variables. Unfortunately, this is hardly a realistic assumption, given the complexity of the typing dynamics of a typical user, which are known to exhibit high variability over time. Reasons range from...
adaptation to new environments to variations in the emotional state of the user [7]. These observations suggest that a robust injection strategy must adapt to the real user activity in real time. For example, intense user activity should result in lower injection rates for the dummy keystroke sequences. The injection should be also context-aware, e.g., a user typing his credit card number should result in lower frequencies of dummy numeric scancodes. To meet these goals, our strategy is to keep the overall behavior steady, with dummy keystroke sequences tuned according to the user activity over time.

Privacy-definition 3. Let $S_r$ be a user-issued keystroke sequence, and $S_d$ a dummy keystroke sequence injected. $S_r$ is per se $\alpha$-undetectable if $S_r \cup S_d \approx \alpha S_{ref}$, where $S_{ref}$ is the reference keystroke sequence.

It is of great importance to choose a suitable $S_{ref}$ by accurately tuning its random variables. For instance, the rate of keystrokes per time interval $X_{S_{ref}}(T_{i,j})$ must be selected orders of magnitude greater than the rate found in a typical user-issued keystroke sequence $X_S(T_{i,j})$. Failing to meet this requirement would break the $\alpha$-undetectability property and potentially allow an adversary to recover the original user-issued keystrokes. Likewise, $Y_{S_{ref},T_{i,j}}(k)$ must agree with the subset of scancodes used by the user and also provide higher frequencies for every possible scancode $\in S_r$. Failure to do so would again break the $\alpha$-undetectability property and potentially allow a context-aware adversary to recover the original user-issued keystroke sequences defined over a limited set of scancodes, e.g., credit card numbers. Similar concerns apply to $Z_{S_{ref},T_{i,j}}(k,h)$.

We now introduce the $S_{ref}$ used in our evaluation. The 3 random variables are selected with uniform probability distributions, with ranges chosen on a per-variable basis:

\[ X_{S_{ref}}(T_{i,j}) \sim \text{Uniform}_{X}(0,400) \]
\[ Y_{S_{ref},T_{i,j}}(k) \sim \text{Uniform}_{Y}(L(\Sigma),U(\Sigma)) \]
\[ Z_{S_{ref},T_{i,j}}(k,h) \sim \text{Uniform}_{Z}(0,2000) \]

Based on our findings, further explained in Section 5, we set the maximum keystroke rate for Uniform$_X$ to 400 keystrokes. The range of Uniform$_Y$ reflects, in turn, the idea that each scancode $k \in \Sigma$ should have an equal probability of occurrence in the reference keystroke sequence $S_{ref}$ (L and U are, respectively, the lower and the upper bound). Choosing a proper range for Uniform$_Z$ proved to be more challenging, as the maximum user hold time cannot be easily estimated in advance. Our strategy is to resort to the maximum hold time found in the dataset published by Killourhy et al in [7]. We believe this dataset to be authoritative and fairly comprehensive, collecting more than 20000 keystrokes timings typed by more than 50 different subjects. In conclusion, we point out that choosing a proper $S_{ref}$ is merely a parameter of our model, and can thus be tuned according to domain-specific requirements at deployment time.

5 Evaluation

We implemented our prototype in a dynamic linked library written in C++ and evaluated it in Windows 7 SP1. All the experiments were performed on a machine with an Intel Core i7 processor and 4GB of RAM. While we tested many different user applications (e.g., Firefox, Thunderbird, Notepad) without incurring any compatibility issue, we adopted Firefox as our application of choice due to its widespread adoption.

Preliminaries. To validate our technique against real user activity, we implemented a user keystroke sequence simulator. Using the patterns from the dataset published by Killourhy et al. in [7], we simulated the activity of 51 different subjects typing the password .tie5Roan1 . 400 times. We split the evaluation in two different parts. First, we select a single keystroke sequence and investigate the performance impact of our solution in terms of additional CPU load and latency perceived by the user. Subsequently, we verify the effectiveness of our technique by ascertaining whether the simulated user activity is $\alpha$-undetectable regardless of the typing subject.

For each experiment we set $S_r$ from the keystroke timings of the selected subject and we initialize the Noise Factory with the reference keystroke sequence (i.e., $S_d = S_{ref}$), ready to be adapted at runtime by the Normalizer. During each run we continuously assess whether the assumption $S_r \cup S_d \approx \alpha S_{ref}$ holds. To this end, we break down each run into two different phases: (i) a first phase simulating the absence of user activity; (ii) a second phase loading the keystroke sequence simulator with $S_r$ and instructing the Normalizer to adapt the sequence of dummy keystrokes $S_d$. Finally, in order to assess whether $S_r$ is $\alpha$-undetectable, we apply the Privacy-definition 1 by instantiating the statistical test $\mathcal{T}$ with a Pearson $\chi^2$ two samples test with significance $\alpha = 0.01$—hence ascertaining whether $S_r$ is 0.01-undetectable. To satisfy Privacy-definition 2, we formulate the following hypotheses test for each random variable $R \in \{X,Y,Z\}$:

\[ H_0 : R_{S_r} \cup R_{S_d} \sim R_{S_{ref}} \]
\[ H_1 : R_{S_r} \cup R_{S_d} \not\sim R_{S_{ref}} \]

The test accepts the null hypothesis ($H_0$) with confidence $1 - \alpha$ if the values generated by both groups of random variables are consistent with a single probabilistic distribution. We calculate the outcome of the test by deriving the frequencies of the values and determining the resulting $p$-value via the same methodology adopted in [8].
Performance. As discussed in Section 2, the injection cycle of the Injector Thread can be configured by tuning the time interval between consecutive runs \( I_t \). Small time intervals yield a more prolific noise generation, but can also degrade performance. The performance impact is also influenced by the number of dummy keystrokes injected at each run. For these reasons, it is crucial to carefully choose both the value of \( I_t \) and the parameters of Uniform\(_X\). The maximum distribution value \( k_{\text{max}} \), in particular, should be tuned to achieve an optimal privacy-performance tradeoff. To evaluate the performance hit perceived by the user, we focus on two different measurements: the CPU load and the keystroke latency, i.e., the delay between the physical generation of a keystroke and the final keydown event in the intended application. Figure 3a shows the results for \( I_t = 10 \text{ms} \), and \( k_{\text{max}} \) varying between 1 and 1450 keystrokes/run. The average keystroke latency remains steady at 250ms for \( k_{\text{max}} \leq 900 \). On the other hand, the CPU load increases linearly, breaking the 50% boundary when the distribution Uniform\(_X\) is config-

![Graph](image-url)
ured to inject a maximum of 900 keystrokes/run.

Figure 3b visually depicts the outcome of the $\chi^2$ tests performed on all the random variables. Results are averaged over all the subjects in the dataset. The time interval associated to each random variable was set to $T_{i,j} = 100\text{ms}$, but we obtained similar results with other values. Each random variable was tested twice, yielding two different sets of bars. In the first row, we report the outcome of evaluating $S_i \cup S_d \approx S_{ef}$. The second row further verifies that the exhibited behavior matches the intended distribution by testing the timings produced by $S_i \cup S_d$ against the underlying distribution. We performed as many $\chi^2$ tests as the number of time intervals in which the random variables are defined. The results are aggregated in a single bar depicting the percentage of success for each value of $k_{\text{max}}$. The experiment shows that all the random variables are negatively affected by low values of $k_{\text{max}}$. The reason is that, for those values, the intensity of the user activity dominates that of the dummy keystroke sequences. However, setting $k_{\text{max}} = 200$ is sufficient to obtain a 100% success rate for all the random variables. The same value in Figure 3a yields a CPU load of less than 10% and a perceived latency of 240ms, values that are typically sufficient in real-time interactions [3].

The second batch of experiments in Figure 3c and 3d depicts the effect of varying $I_i$ ($k_{\text{max}} = 400$). Since the Injector Thread is queried every time some keystrokes are issued to the user application, we expect low $I_i$ values to yield low keystroke latencies and high CPU load. Figure 3c confirms this intuition. A reasonable trade-off is found at 50ms, as both latency and CPU load are still within acceptable values, i.e., 250ms and 21% respectively. For these values, Figure 3d shows that $S_i$ is 0.01-undetectable in all cases with the notable exception of the variable $Y_{S_i \cup S_d} (k)$, which yields low success rates for $I_i \geq 100$. This demonstrates that the exhibited distribution of scancodes is highly dependent on the overall keystroke rate, which in turn decreases for higher values of $I_i$. However, we note that an injection cycle $I_i = 50\text{ms}$ is sufficient to obtain a 100% success rate.

**Effectiveness.** Similar percentages can be observed regardless of the subject. Figure 4 shows the results of our statistical tests for all the 51 subjects in the dataset. In almost all the cases, our technique was able to make the keystroke sequences 0.01-undetectable, a value realistically sufficient to safeguard the privacy of the user.

## 6 Conclusion

Existing malware countermeasures are generally focused on preventing or detecting malicious software instances. In this paper, we take a new perspective to the problem. We present NoisyKey, a technique that allows the user to live together with a keylogging malware without putting his privacy at stake. The key idea is to confine the user private data in a noisy event channel flooded with artificially generated keystroke activity. Our technique transparently allows legitimate applications to recover the original data, while exposing the keylogger to the original noisy stream. Our evaluation shows that the resulting stream of data is statistically undetectable from arbitrary stream of data. We also implemented our technique in a lightweight library, and tested it on modern operating systems and applications. Our work shows a new interesting paradigm in dealing with malicious software, and we believe our strategy to have possible applications in other domains and classes of malware.

## References


