Timing Side-Channel Attacks on USB Devices Using eBPF

Gianmarco Lusvardi[1], Luca Ferretti[1] and Mauro Andreolini[1]

[1]University of Modena and Reggio Emilia

Abstract

Timing side-channel attacks allow to infer information processed by an algorithm from its execution times. The impact of these attacks in cryptography has been proven in several real-world scenarios, such as the compromise of private keys associated to digital signature systems deployed in local or remote systems. Collecting precise timings is paramount to maximize attacks effectiveness, that is, to increase the probability of recovering the private keys. In this paper, we investigate the feasibility of collecting high-precision timings of cryptographic algorithms executed on embedded devices by leveraging a normal personal computer. We focus on digital signatures and on devices accessed via USB, such as smart cards and authentication tokens. To this end, we study the possibility of measuring signatures timings by using the extended Berkeley Packet Filter (eBPF), a technology that allows to develop programs which are executed in kernel space, to develop a program to measure timings related to data exchanges between the host and USB devices directly in kernel space. We evaluate the effectiveness of the approach by crafting a testbed based on a vulnerable smart card. We collect and compare measurements taken both in user and in kernel space, and we analyze their efficacy when used as inputs to a known mathematical heuristic used for recovering the private keys. Experimental results show that the proposed approach increases the precision of collected timings and the probability of running a successful attack in most cases.

Keywords

Timing Attack, eBPF, USB, Digital Signatures, Smart Card

1. Introduction

When any kind of program is run on a computer, in addition to the desired output it also produces side-effects on the physical environment, such as running time or power consumption. Measurements of such effects when they depend on some secret parameter may allow an adversary to recover the secret through a so-called *side-channel attack*. Side-channel attacks have been extensively studied in literature and may be based on many types of information, algorithms, platforms and interfaces [1, 2, 3, 4, 5, 6].

We focus on *timing side-channel attacks* on cryptographic schemes, where adversaries measure execution times of algorithms which may depend on a secret key. Although the need for implementing such schemes with algorithms that are time-constant with regards to secret information is well-known in cryptographic engineering, vulnerabilities may still raise due to human errors (e.g., inexperienced developers) or to subtle optimizations operated by compilers [7]. To the aim of detecting and exploiting timing side-channel vulnerabilities, collecting precise timings is paramount to maximize attacks effectiveness [1].

In this paper, we investigate the feasibility of collecting high-precision timings of cryptographic algorithms executed on embedded devices by leveraging a normal personal computer, without any special equipment. We focus on digital signatures and on devices accessed via USB, such as smart cards and authentication tokens. In order to perform a successful timing side-channel attack, we need a very precise clock that can measure the running time of the targeted implementation in order to filter out as much measurement noise as possible and pick up any difference in time taken by the program with different inputs. The USB protocol may introduce quantization in timing measurements, which can pose a serious problem for such attacks, because it has the effect of reducing the precision of the clock used for measuring.

ITASEC 2024: The Italian Conference on CyberSecurity, April 08–12, 2024, Salerno, IT

gianmarco.lusvardi@unimore.it (G. Lusvardi); luca.ferretti@unimore.it (L. Ferretti); mauro.andreolini@unimore.it (M. Andreolini)

0009-0003-5341-0257 (G. Lusvardi); 0000-0001-5824-2297 (L. Ferretti); 0000-0002-7408-6906 (M. Andreolini)

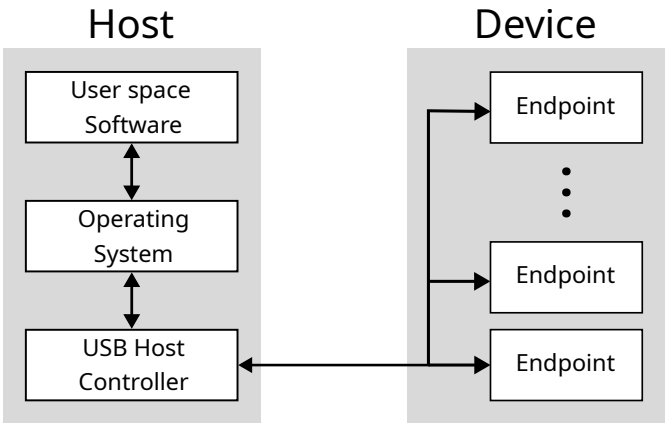

Figure 1: USB 2.0 architecture and data flow

Our contribution is twofold. First, we study the possibility of measuring signatures timings by using extended Berkeley Packet Filter (eBPF), a technology that allows to develop programs which are executed in kernel space and are typically associated to system events, and develop a program that measures timings related to data exchanges between the host and USB devices directly in kernel space. Measuring timings in kernel space allows us to reduce the measurement noise introduced by process scheduling and buffering.

Second, we design an experimental testbed to evaluate the efficacy of the measurements. We leverage a USB Armory Mk-II device [8], which is a security-focused Single-Board Computer (SBC) provided with USB device emulation features, flashed with the GoKey firmware [9] supporting the OpenPGP protocol [10], that we modified to make it vulnerable to a known timing side-channel attack [1]. The testbed allows us to compare timings at different abstraction layers and analyze how noise introduced at multiple levels affect the efficacy of a known heuristic used for recovering the private key. Experiments show that not only it is feasible to detect timing differences despite the time quantization introduced by the host-centric nature of USB, but also that they are exploitable for performing a successful attack. Moreover, measuring signature timings in kernel space by using eBPF improves the precision of the timing measurements, allowing an adversary to increase the attack success probability with less signatures.

The paper is organized as follows. Section 2 describes base knowledge on the USB protocol, eBPF and lattice attacks for key recovery. Section 3 discusses how to collect timings at the kernel level through eBPF. Section 4 describes the experimental testbed. Section 5 presents experimental results. Section 6 discusses concluding remarks and future work.

2. Base knowledge

We discuss base knowledge on the USB protocol in Section 2.1, extended Berkley Packet Filter (eBPF) in Section 2.2, and lattice attacks for key recovery in Section 2.3.

2.1. USB

The Universal Serial Bus (USB) standard allows external peripherals to be connected to a host computer to increase its capabilities. We focus on version 2.0 of the USB standard [11], whose architecture and data flow is outlined in Figure 1. The protocol operates between a USB host, such as a normal personal computer, and a USB device, such as a storage device or, as in our case, a smart card. The host runs user space software on an operating system, which provides the necessary drivers for accessing USB devices. In turn, those drivers use the host controller driver to access the host controller, which is hardware responsible for implementing USB communications with USB devices. Each device exposes

its functionalities through one or more *endpoints*, which represent the source or sink of any USB data transfer. Ultimately, the logical flow of information is between a client software and an endpoint of a USB device. Each endpoint is identified by an *endpoint address*, which is composed of an *endpoint number* and an *endpoint direction* [11, Chapter 2, 5.3].

The USB protocol is host-centric, which means that it is always the host that initiates every data transfer, even the ones from the device to the host [11, Section 4.4]. This may affect timing attacks because it introduces an upper bound on the rate at which the endpoint is polled for new data, hence it may also introduce dangerous quantization in time measurements. Another aspect which may influence the quantization is the type of particular data transfer that an endpoint uses.

We focus on bulk transfers because they are commonly used for USB smart cards, as they are designed to support the transfer of relatively large data bursts [12]. Bulk transfers offer no guarantees over bandwidth or latency of the communications [11, Sections 4.7, 5.4]. In order to perform a data transfer (called a *transaction* in USB) with bulk endpoints, the host specifies the direction of the data transfer through a *token packet*, which can be an *IN token* or an *OUT token*, respectively if the direction of the data transfer is from the device to the host, namely an *IN transaction*, or vice versa, namely an *OUT transaction* [11, Section 8.5.2].

- for IN transactions, the IN token includes the destination endpoint, such that the device may reply with a DATA packet back to the host. The device may answer with a NAK packet for declaring that there is no data to be sent.
- for OUT transactions, the host send a DATA packet after sending the OUT token. The device replies with an handshake packet (ACK, NAK or STALL) depending on its current state.

2.2. Extended Berkeley Packet Filter (eBPF)

The eBPF technology can be used for running limited sandboxed programs in the kernel space of a Linux operating system, without the need to patch and recompile the kernel. An eBPF program can be usually written in a relatively high-level language, such as C, which then gets compiled into eBPF bytecode, and loaded at runtime attached to a specific hook point, to inspect the execution of a particular kernel function when a system event occurs and to report results back to user space programs. eBPF programs have access to a limited set of helper functions offered by the kernel and to a very limited stack size [13, `Documentation/bpf/bpf_design_QA.rst`]. Thus, it is only possible to rely on them for very simple operations, such as tracing kernel functions, which fits our aim of measuring timings [14].

2.3. Lattice attacks for key recovery

A lattice attack is a mathematical heuristic based on the hidden number problem capable of recovering the (EC)DSA private key from a limited amount of signatures generated by using biased nonce values [1, 2]. This is the second part of the Brumley and Tuveri's attack [1] that we reproduce in this paper.

A *lattice* is an integer vector space with coefficients in $\mathbb{Z}$. Given an ordered base $(\mathbf{b}_1, \dots, \mathbf{b}_n) \in \mathbb{Z}^n$, the n-dimensional lattice that it generates is[1]

$$\mathscr{L}(\mathbf{b}_1, \dots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} c_i \mathbf{b}_i \ : \ c_i \in \mathbb{Z} \right\} \tag{1}$$

Let $\{h_1, \dots, h_d\}$ be a set of messages and let $\{(r_i, s_i) \ : \ i = 1, \dots, d\}$ be the set of respective ECDSA signatures performed with the key pair (Q, α), where α is the private key and $Q = [\alpha]G$ is the public key over a specific elliptic curve where G is the generator of the ECDSA subgroup of order n used for signing. Let $\{\tau_i \ : \ i = 1, \dots, d\}$ be a set of side-channel information associated to the signautres: τ_i is associated with the signature (r_i, s_i) such that $\forall i, j = 1, \dots, d, i \neq j \ : \ \tau_i < \tau_j \Rightarrow \left\lfloor \log_2 k_i \right\rfloor \leq \left\lfloor \log_2 k_j \right\rfloor$ where k_i is the nonce used for generating signature (r_i, s_i). Lastly, let l_i be the assumed number of null most significant bits of the nonce k_i used for producing the signature (r_i, s_i) (also called *leakage*).

[1] The more rigorous name of the considered lattice is *integral lattice*.

If we now consider the subset of z signatures with the minimum τ_i possible, then we can consider the lattice $\mathscr{L}$ spanned by

$$B = \begin{pmatrix} 2^{l_1+1}n & 0 & \dots & 0 & 0 & 0 \\ 0 & 2^{l_2+1}n & \dots & 0 & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 2^{l_z+1}n & 0 & 0 \\ 2^{l_1+1}t_1 & 2^{l_2+1}t_2 & \dots & 2^{l_z+1}t_z & 1 & 0 \\ 2^{l_1+1}u_1 + n & 2^{l_2+1}u_2 + n & \dots & 2^{l_z+1}u_z + n & 0 & n \end{pmatrix} \tag{2}$$

where $t_i = s_i^{-1}r_i \mod n$ and $u_i = s_i^{-1}h_i \mod n$.

It is possible to prove that there is a vector $\mathbf{e} = (e_1, \dots, e_z, \alpha, n)$ in the lattice spanned by the matrix in Equation 2 which is particularly short. What is interesting about the vector $\mathbf{e}$ is its next-to-last element, which is the private key used for producing the signatures. Note that it is also possible that such element is $n - \alpha$ [4, Section 2.2]. A more extended analysis for this claim is reported in [2, 4]. It is now possible to use the LLL and BKZ algorithms on the aforementioned basis of the lattice $\mathscr{L}$ in order to obtain an equivalent lattice which rows are possible candidates to be the vector $\mathbf{e}$. The resulting lattice is called *reduced lattice*. Therefore, by checking if each element in the next-to-last column of the reduced basis is α or $n - \alpha$ by multiplying each of them by G and check if the result is Q, it is possible to recover the private key [2].

The lattice spanned by the matrix in Equation 2 is the result of applying the embedding and recentering techniques as discussed in [2].

In order to perform a lattice attack, we need to assign values to the leakages l_i. It would be possible to assign different values to each l_i depending on what we expect from the particular setup (as it has been done in [2] with geometric bounds) or simply to assign the same value to all the l_i: $l_1 = l_2 = \dots = l_z = l$. The less the leakage we assume, the more signatures we require to add to the lattice in order to perform a successful attack [2].

3. Measuring timings in eBPF

We design an eBFP program to collect timings related to digital signatures produced by an OpenPGP smart card connected through USB. To measure timings in eBPF, we need to decide to which kernel function attach the eBPF program. We build upon publicly available source code developed for other purposes (e.g. [15]) that we modify for our scenario.

When a USB driver wants to initiate a data transfer, it constructs a structure named *USB Request Block* (URB), which contains the data to be sent and the endpoint address among other relevant information for the transfer, and uses the `usb_submit_urb` function present in the Linux kernel to do sanity checks and to pass the control of the URB to the USB host controller driver [16, 17] [13, `drivers/usb/core/urb.c`]. The host controller driver then adds the URB to a specific endpoint queue [13, `drivers/usb/core/hcd.c`], from which the host controller sends the URB to the USB device. When the host controller driver finished handling a URB, it passes the control of the URB back to the USB device driver by calling the `usb_hcd_giveback_urb` function [13, `drivers/usb/core/hcd.c`]. Hence this function gets called after data transfers in both directions. Therefore, in order to measure signature timings in kernel space, we attach the eBPF program to the kernel function `usb_hcd_giveback_urb`.

We design the eBPF program to collect relevant information contained in the URB:

- the PID:VID pair identifying a USB device [13, `include/uapi/linux/usb/ch9.h`], [11, Table 9.8], [16];
- a copy of the data transferred through USB, potentially truncated to the first few bytes. The truncation may be necessary due to the limited size of the stack allocated for eBPF programs. The copy is needed to tell apart requests for signatures and the relative response from the USB device: by analyzing the first bytes of the message, we can use the OpenPGP smart card standard

to understand the direction of the transfer (for more information see [10]). The full ECDSA signature is collected by the signing program in user space;
- a timestamp collected with the eBPF helper function `bpf_ktime_get_ns` [18].

This information is then passed to the user space program that loaded the eBPF program, which checks if the PID:VID pair is the same as the USB device, to ensure that the data comes from, or goes to, the USB device. If so, the user space program stores the collected timestamp along with the direction of the transaction. The set of the collected timestamps is later processed by a separate program to find out the exact time taken for a signature to be produced by the USB device.

4. Experimental testbed

We design an experimental testbed for the collection of digital signature timings based on the USB Armory MkII (or *Armory* for short) [8], which is a Single Board Computer (SBC) specialized for security and cryptographic purposes. It is possible to use the Armory as a OpenPGP smart card by flashing it with the GoKey firmware provided by the manufacturer [9], compiled with the TamaGo compiler [19]. The Armory exposes two bulk endpoints for communicating with the host computer and be able to sign messages, complying with the CCID standard for smart cards [12].

We modify the GoKey firmware [9, commit id 5aa37f492dc] and the TamaGo standard library (version 1.20.5) to reproduce Brumley and Tuveri's attack against a buggy implementation of the Montgomery ladder for computing point-scalar multiplication over elliptic curves in OpenSSL 0.9.8o [1]. Specifically, the implementation leaks the number of null most significant bits of the nonce for each signature because, when it iterates over all the bits of the nonce, it skips all the null bits at the beginning, thus making the point-scalar multiplication of a short nonce shorter in time than the same operation for a longer nonce. Moreover, in order to speed up signature collection we also removed the requirement of asking for a password every time a signature is made, which would take roughly two to three seconds of computation due to the password based key derivation function. Finally, in order to evaluate the performance of the attack, we patched the GoKey firmware in order to report also the nonces used for producing a signature. This is not a requirement to run a successful attack and it is not used for retrieving the private key (which would otherwise be trivial). Instead, nonces are used for further comparing the performances between measurements performed in kernel space and user space in order to compute the number of errors in the analysis we do in Section 5.

5. Experimental evaluation

We evaluate the effectiveness of the kernel space timings measurements described in Section 3 within the testbed described in the Section 4 against timings collected in user space. The user space program collects signatures using libusb version 1.0.26 [20]. The program issues signatures sequentially to the Armory and collects two timestamps: one immediately after sending data to the Armory and another one immediately after having received the relevant signature from the Armory by probing the `CLOCK_MONOTONIC` system clock.

Brumley and Tuveri's attack can be split into two different phases: signature timing collection and lattice attack [1]. The lattice attack provides us a simple benchmark for the timing collection process because its success or failure depends on the goodness of the timing collection.

The first thing that we assess is whether the timing leakage is measurable despite the time quantization introduced by the USB protocol. In order to do that, we use a constant nonce for signing operations in the GoKey firmware with a known number of null most significant bits from 0 to 12 and then we collected 5000 signatures for each value of the nonce, measuring the time each signature took both on the Armory and on the host in kernel space. Timings collected locally on the Armory are plotted in Figure 2 and corresponding timing measurements in kernel space on the host are plotted in Figure 3. It can be observed that in both cases there is an almost linear dependence between the running time and the length of the nonce.

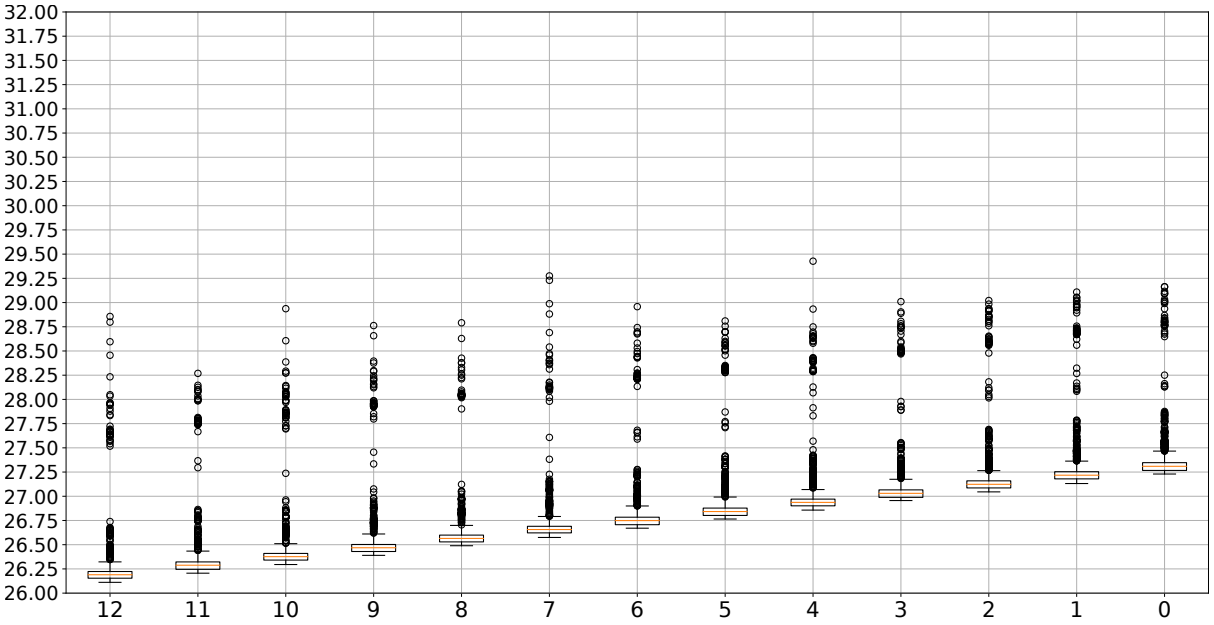

Figure 2: Timings [ms] measured locally on the Armory with regard to the number of null MSB in the nonce.

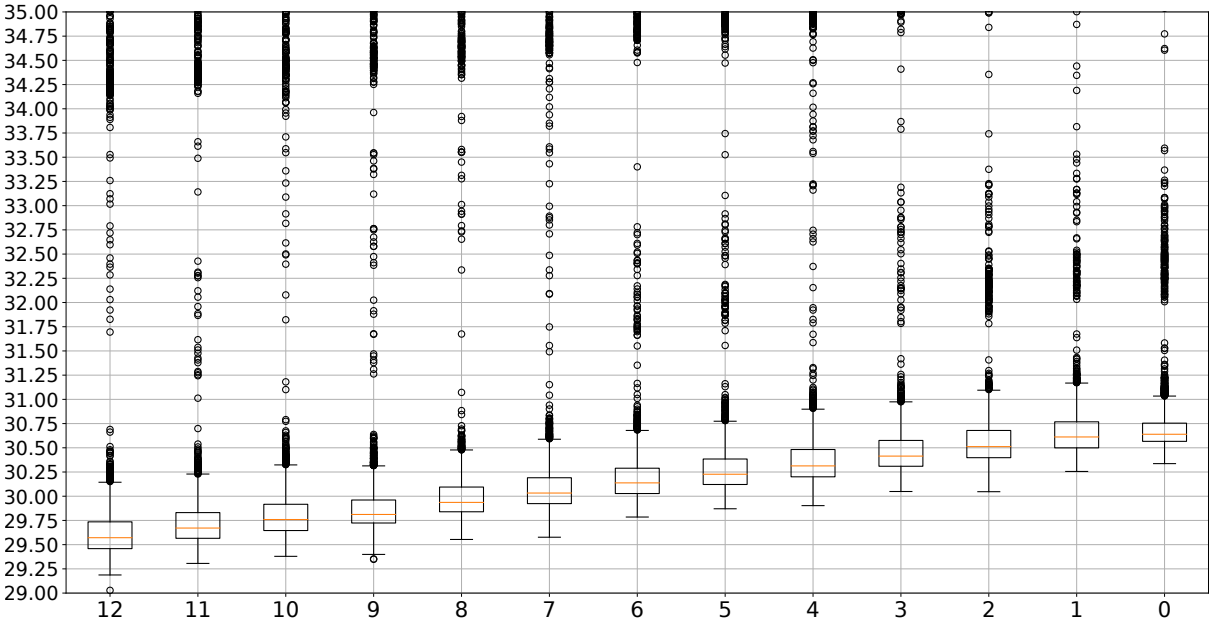

Figure 3: Timings [ms] measured in kernel space with regard to the number of null MSB in the nonce. Some outliers that took a lot of time were omitted from the plot.

In order to evaluate the performance of the lattice attack with respect to the measuring method, we issue 10000 signatures with 25 different private keys to the Armory while collecting both the user space and kernel space timings, and the nonce used for producing each signature.

We then compare the two timing collection methods in two ways: first, we compare the percentage of lattice attacks that succeed when sorting respectively by user and kernel space timings; second, we take a reference number of signatures for building the lattice and then compare the average number of errors made by sorting by user and kernel space timings. Errors are defined as signatures with an amount of leakage that is smaller than the assumed value.

In order to perform the lattice attack (see Section 2.3 for the notation), we consider a fixed value of the leakage $l = 5$ and a varying number z of signatures used for building the lattice. We build the lattice by ordering the entire set of 10000 signatures by user and kernel space timings and perform the attack with

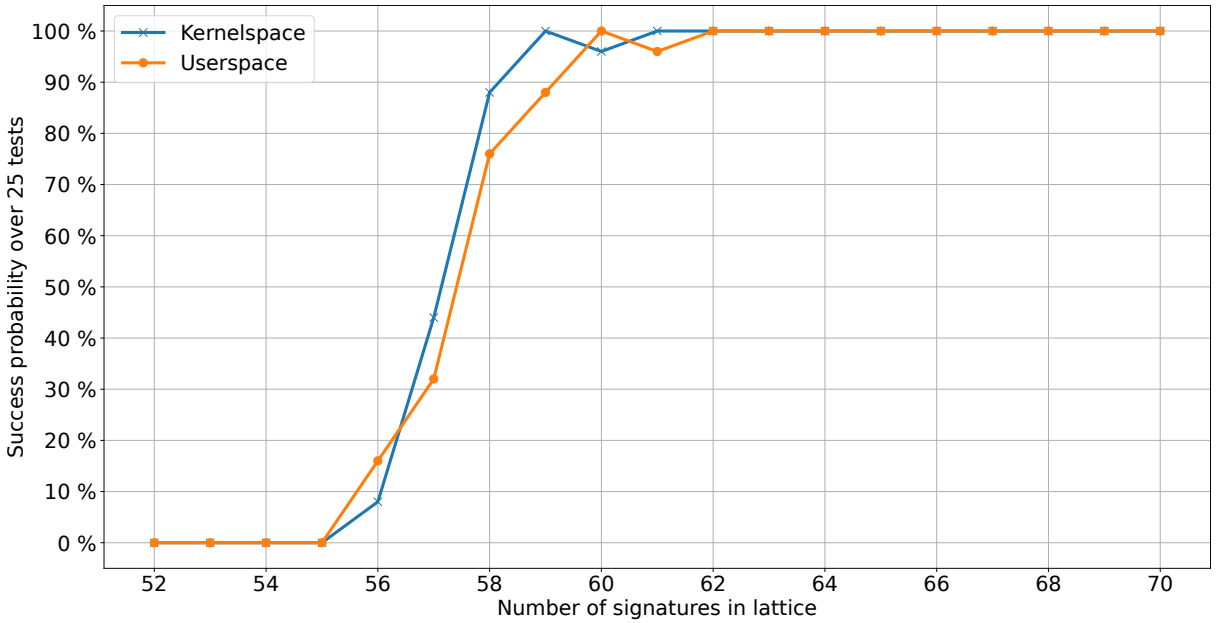

Figure 4: Comparison of the recentered lattice attack performance with 5 bits of assumed leakage for each signature and 10000 signatures.

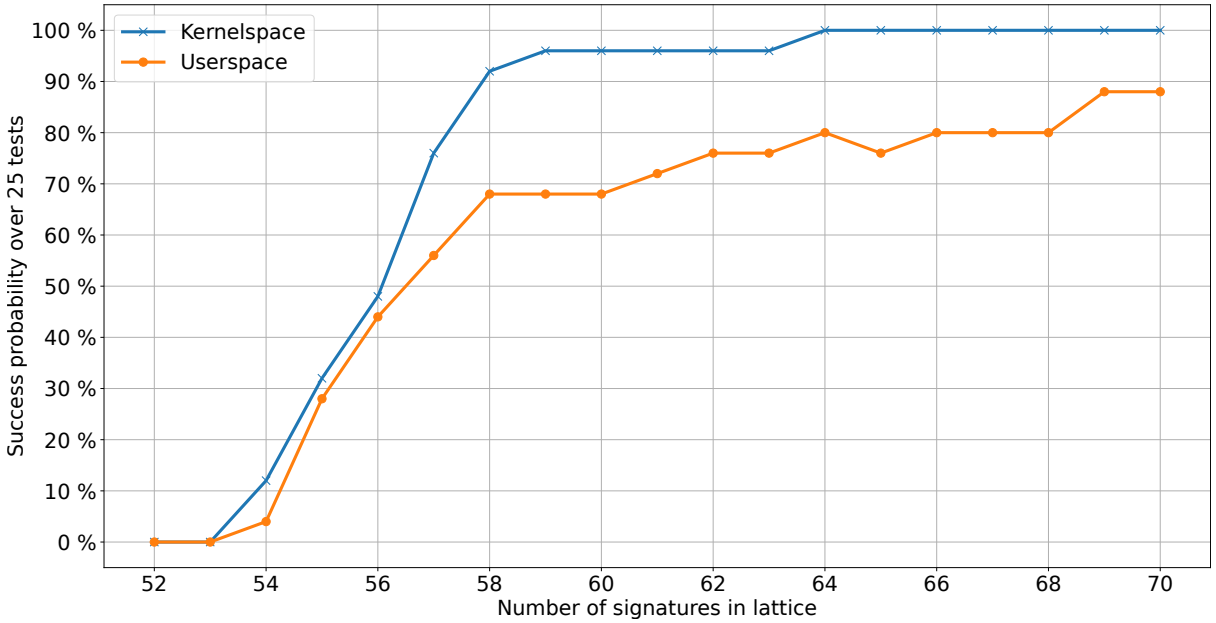

Figure 5: Comparison of the recentered lattice attack performance with 5 bits of assumed leakage for each signature and 5500 signatures.

both types of timings. The lattice reduction performed within the attack first uses the LLL algorithm and, in case of failure, it is retried with the BKZ algorithm with block sizes $\{15, 20, 30, 40, 45, 48, 51, 53, 55\}$ as in [2]. Then we repeat the same procedure only by using the last 5500 signatures of all the 10000 unsorted collected signatures, without reshuffling. We compare the performance of the lattice attack for 10000 and 5500 collected signatures respectively in Figures 4 and 5. Results in Figure 4 show that when leveraging a dataset with a very high number of signatures, the success probability improvement is very small because the increased noise is filtered out by the high amount of samples. Instead, results in Figure 5 show that with kernel space timings it is possible to collect less signatures and still perform

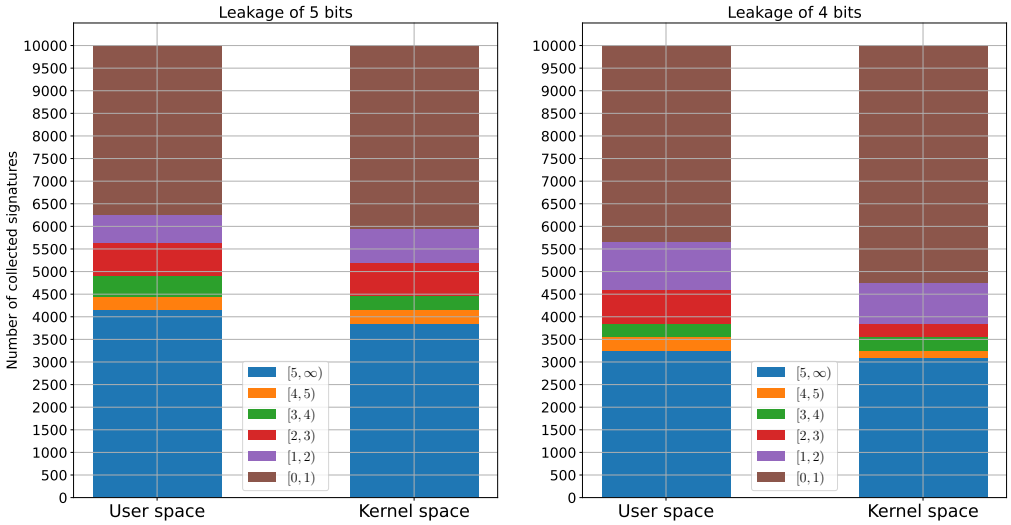

Figure 6: Comparison of average number of errors if we choose the first z_l signatures among the first n signatures sorted by user space and kernel space timing respectively.

a lattice attack with a high probability of success. By comparing kernel timings between 5500 and 10000 signatures, the probability of the attack succeeding with a number of signatures within the lattice ranging from 54 to 58 is higher by using the subsampled dataset with 5500 signatures than by using the whole dataset with 10000 signatures. This may be due to the fact that increasing the number of signatures also increases the number of signatures computed on nonces with a higher number of null most significant bits than assumed. We leave improved analyses as future work.

In order to perform the second analysis, we take $z_4 = 88$ and $z_5 = 60$ as a reference number of signatures assuming a leakage of respectively 4 and 5 bits. Then, we select the first n collected signatures to simulate the collection of less than 10000 signatures and we sort such subset by user and kernel space timings. After that we select the first z_l signatures in each subset, where z_l is defined as above. Finally, we average the amount of errors over the 25 tests we performed and plot the results in Figure 6. The plot assigns the average amount of errors in the first z_l signatures (computed over all the 25 tests) to each number n of signatures collected. From the plot we can see that, given a number of collected signatures n, the average amount of errors made by sorting by user space timings is always greater than or equal to the average amount of errors made by sorting by kernel space timings. Thus, measuring timings in kernel space increases their precision.

6. Conclusions

We showed that it is possible to perform timing attacks on USB smart cards through a normal personal computer. Moreover, the use of eBPF allows to increase the precision of timing measurements over USB, avoiding noise introduced by other operating system components or user space programs, without the need to write a kernel module or patch the kernel. The proposed approach allows to assess the security of USB smart cards and similar devices against timing side-channel attacks with greater confidence, even without specialized hardware. Future work will include further analyses of USB timing attacks in different scenarios, including other kinds of USB devices and cryptographic protocols, and probing other kernel functions that may allow more precise timing measurements.

Acknowledgments

This work has been supported by the project “C4SI” funded by the Emilia Romagna Region (Fondo Europeo di Sviluppo Regionale - FESR) - CUP E67G22000630003.

References

[1] B. B. Brumley, N. Tuveri, Remote Timing Attacks Are Still Practical, in: 16th European Symp. Research in Computer Security (ESORICS), 2011.
[2] J. Jancar, V. Sedlacek, P. Svenda, M. Sys, Minerva: The curse of ECDSA nonces (Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces), IACR Trans. Cryptographic Hardware and Embedded Systems (2020).
[3] D. Moghimi, B. Sunar, T. Eisenbarth, N. Heninger, TPM-FAIL: TPM meets Timing and Lattice Attacks, in: 29th Usenix Security Symp., 2020.
[4] C. Sun, T. Espitau, M. Tibouchi, M. Abe, Guessing Bits: Improved Lattice Attacks on (EC)DSA with Nonce Leakage, IACR Trans. Cryptographic Hardware and Embedded Systems Issue 1 (2022).
[5] D. Brumley, D. Boneh, Remote Timing Attacks are Practical, Computer Networks (2005).
[6] B. Nassi, E. Iluz, O. Cohen, O. Vayner, D. Nassi, B. Zadov, Y. Elovici, Video-Based Cryptanalysis: Extracting Cryptographic Keys from Video Footage of a Device’s Power LED Captured by Standard Video Cameras, in: 45th IEEE Symp. Security and Privacy (SP), 2024.
[7] G. Barthe, B. Grégoire, V. Laporte, Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”, in: 31st IEEE Computer Security Foundations Symp. (CSF), 2018.
[8] USB Armory, https://www.withsecure.com/en/solutions/innovative-security-hardware/usb-armory, accessed Oct. 2023.
[9] A. Barisani, GoKey Firmware, https://www.github.com/usbarmory/GoKey, 2023.
[10] A. Pietig, Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems (v. 3.4.1), https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.4.pdf, 2020.
[11] USB Implementers Forum, Universal Serial Bus Specification v2.0, 2000.
[12] USB Implementers Forum, Specification for Integrated Circuit(s) Cards Interface Devices (Rev. 1.1), https://www.usb.org/sites/default/files/DWG_Smart-Card_CCID_Rev110.pdf, Apr. 2005.
[13] Linux Kernel v6.7, https://www.kernel.org, accessed Apr. 2024.
[14] What is eBPF, https://ebpf.io/what-is-ebpf/, accessed Oct. 2023.
[15] eBPF USB inspector, https://github.com/gpioblink/ebpf-usb-inspector, accessed Oct. 2023.
[16] The kernel development community, The Linux-USB Host Side API, https://www.kernel.org/doc/html/v6.7/driver-api/usb/usb.html, accessed Oct. 2023.
[17] G. Kroah-Hartman, Writing USB Device Drivers, https://www.kernel.org/doc/html/v6.7/driver-api/usb/writing_usb_driver.html, accessed Feb. 2024.
[18] bpf-helpers(7), Linux man-pages, Apr. 2023. URL: https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html.
[19] A. Barisani, A. Rosano, TamaGo - bare metal Go for ARM/RISC-V SoCs, https://github.com/usbarmory/tamago, accessed Lug. 2023.
[20] libusb, https://libusb.info, accessed Feb. 2024.