Macchiato: Importing Cache Side Channels to SDNs

Amir Sabzi[1], Liron Schiff[2], Kashyap Thimmaraju[3], Andreas Blenk[4,6], Stefan Schmid[5,6,7]
[1] University of British Columbia, [2] Guardicore Labs [3] Humboldt Universität zu Berlin [4] TU Munich
[5] TU Berlin [6] Faculty of Computer Science, University of Vienna [7] Fraunhofer SIT

ABSTRACT

Since caches are shared and coherent, a memory access of one process may evict from the cache another process' memory block with an address mapped to the same cache line. This property is exploited by several attacks to form side channels. We show that MAC learning in Software Defined Networks (SDNs) has a similar property in the sense that a MAC address discovered by one network device may be revoked by the discovery of the same address at another switch. This allows us to implement Macchiato, a covert channel for SDNs between any two network devices (including hosts); prior SDN covert channels required at least one malicious switch. We evaluate a prototype implementation of Macchiato and discuss how methods to improve the performance of cache side channels (such as deep neural networks) can also be used in Macchiato.

CCS Concepts • **Security and privacy** → **Network security**; • **Networks** → **Programmable networks**;

1 INTRODUCTION

Modern data-driven and distributed network applications operate at an unprecedented scale [5]. To operate and manage such warehouse-scale distributed systems, academia and industry have gravitated towards a network architecture that outsources control over the network data plane to a logically centralized control plane [4, 5, 18]. The centralized control plane simplifies network management and also enables automation. For example, the basic L2 switching functionality found in (cloud, campus, or enterprise) networks can be implemented in SDNs by a central MAC learning application (also known as *mobility*) which, upon the discovery of a new MAC address, installs complete paths, also supporting VM migrations and reaction to network failures.

ANCS '21, December 13–16, 2021, Layfette, IN, USA

ACM ISBN 978-1-4503-9168-9/21/12…$15.00
https://doi.org/10.1145/3493425.3502758

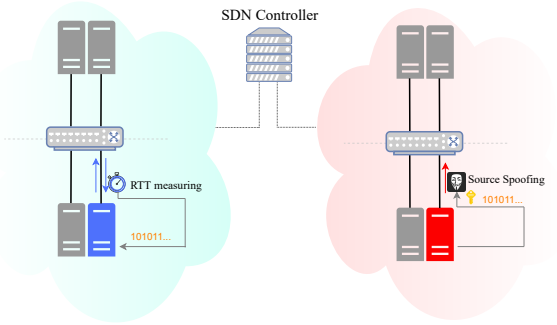

Figure 1: Macchiato covert channel between two isolated hosts. The (red) sender modulates confidential data (e.g., a private key) by triggering SDN reconfigurations of flows with source MAC address of the (blue) receiver host. The receiver identifies the reconfigurations by RTT measurements and extracts the sent data.

Thimmaraju et al. [30] described how this feature can be exploited by malicious switches as a rendezvous protocol. Malicious switches can fabricate (Packet-In) messages to the controller which triggers the mobility feature in the controller, resulting in deletion/addition events at the respective switches. If the switches have agreed upon specific identifiers (e.g., MAC or IP addresses) to trigger mobility events, the identifier of the deletion/addition operation can be used as a signaling mechanism to coordinate or synchronize an attack. In a similar vein, Krösche et al. [14] introduced a covert channel wherein malicious switches exploit the OpenFlow handshake for secret communication (the P4Runtime protocol prevents such a channel by design [7]).

However, these techniques make strong assumptions on the data plane. In particular, they require an exploitable vulnerability in the switch [29]. Inserting hardware trojans may also be expensive and unrealistic for attackers with low resources. Hence, we in this paper ask: *is it possible for hosts to exploit the control plane for covert communication without a malicious switch?*

This paper shows that in the same way as cache address collisions allow one process to extract information from another (a *side channel*), an SDN host can learn about another host network activity by using the same source MAC address. This host ability translates into a covert channel which we name MAC-macchiato ("MAC marked" in Italian) or simply

Macchiato as it is based on the mobility application that clears the old rules associated with the MAC address when it is marked in a new location.

Macchiato allows two hosts that are isolated (e.g., using VLANs) from each other, but connected to switches managed by the same controller, to secretly communicate with each other. As a result, hosts can violate fundamental network security policies and/or mechanisms, e.g., to leak secret keys, exfiltrate intellectual property, etc.

As illustrated in Fig. 1, the red host on the isolated network on the right sends a message (to arbitrary destination) while spoofing the blue host's MAC address. The switch connected to the red host will contact the controller to report a new MAC address discovery. The controller, running a mobility application, addresses this report as if the blue host migrated to a new location and perform flow reconfiguration - deleting flow rules used by the blue host to communicate with it local peers at the old location and installing required new flow rules at the new location. The flow reconfiguration introduces measurable forwarding delays to the blue host that can be used repeatedly by the red host to modulate and transmit confidential data.

Since our covert channel exploits measurable differences in packet responses (see Fig. 2) introduced by the mobility application in the controller, it is independent of the SDN protocol (OpenFlow, P4Runtime, etc.) between the switches and controller. This not only makes our attack novel but also increases the potential threat it poses in software defined networks. In an extreme scenario, the covert channel could be used by a malware residing in a well segmented zone in the network without Internet connectivity, to affect legitimate Internet facing services in a DMZ segment of the network and thereby leak critical information to the Internet [1, 27].

Contributions. In summary, this paper makes the following contributions.

- We describe Macchiato, a covert communication channel between end hosts that exploits MAC learning applications in SDNs, and which is inspired by cache side channels.
- We discuss how methods to improve the performance of cache side channels (such as deep neural networks) can be used in Macchiato.
- We evaluate the throughput of Macchiato theoretically and experimentally.
- We initiate the discussion of the countermeasures and recommend guidelines for network operators and developers to guard against the described attack.
- We make our code and experimental artefacts open-access together with this paper.

Related work. Our work is inspired by the side channels arising in caching systems. These side channels have been studied intensively in the literature [9, 16, 23, 25, 32], and also used for covert communication. However, there is limited research on covert channels in SDNs. Covert channels in SDN were first introduced by Thimmaraju et al. [30] followed by Krösche et al. [14]. Tahir et al. [27], designed and developed Sneak-Peek, a high speed covert channel in data center networks that also utilizes a delay mechanism. The sender's flow introduces a delay into the receivers flow over the same network link thereby covertly communicating information based on the delay measured by the receiver. Although similar to Macchiato, Macchiato requires less resources for the attacker compared to Sneak-Peek.

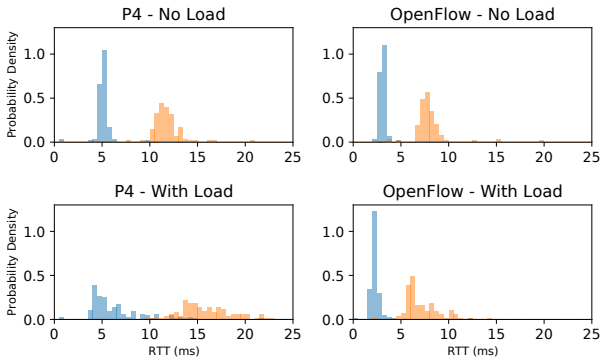

Figure 2: Distribution of RTTs with (orange) and without (blue) flow reconfiguration, and for P4 and OpenFlow with and without external load on the controller.

Le et al. [15] designed a covert channel in SDNs that exploits rules conflicts in the control plane to establish host to host and switch to switch covert channels. However, in their threat model the adversary can install applications in the controller that causes rule conflicts, or third party controller applications create conflicting rules unintentionally. However, we assume the controller and all its applications as a trusted entity which works properly. Cao et al. [2] designed an SDN-based covert channel that involves a malicious controller application to covertly exfiltrate data from the controller to the hosts in the network. However, in this paper, we assume the controller applications are benign and the attacker's goal is to exfiltrate data from one host to another.

Organization. In Section 2 we introduce the high level concepts of Macchiato, followed by technical details in Section 3. Next, we evaluate Macchiato in Section 4, discuss countermeasures in Section 5 and close with a discussion in Section 6.

2 ATTACK OVERVIEW

Here, we define the threat model and explain the core idea behind our covert channel.

2.1 Threat Model

We consider the SDN setting where two network hosts are compromised by an adversary with an objective to transmit information, e.g., private keys or confidential data covertly

between them. We assume that the adversary code running at each side of the covert channel has privileged access over the host network stack. For example, the adversary can send packets with arbitrary (spoofed) source Ethernet and IP addresses, use an incorrect gateway, and send ARP requests and responses maliciously. We note that some variants of our covert channel can operate even when running privileged code only at one side of the channel given that it obtains knowledge of the other side's MAC address.

The position of compromised hosts in the network cannot be determined by the adversary. For instance, compromised hosts can be physically disconnected or separated by a firewall in the data plane. However, the compromised hosts should be connected to switches that are managed by the same logically centralized controller. These switches can be managed by the controller using OpenFlow[20], P4runtime[28], or any similar protocol. The controller should support the path update functionality, which is implemented in several forms by different control applications including Mobility, Learning switch, and MAC learning. We assume the network is agile, allowing host creation and migration so control applications are not restricted by static switch configurations.

Other than malicious hosts, all other hosts are considered trustworthy, and cannot be compromised by the adversary but may respond to network requests such as pings and ARP packets. All network infrastructure components are trusted as well, including switches, routers, firewalls, intrusion detection system, controller and its applications.

To maintain data plane consistency, control applications, such as intent routing, MAC learning and mobility, update paths when events are reported by the switches, indicating that the network has changed. Such network changes may include host migration, flow rule timeouts and link failures. The update procedure, to which we refer as *flow reconfiguration*, can vary depending on the types of switches or controllers, but essentially always consists of configuring new flow rules and deleting old ones in all affected switches. A first approach for exploiting flow reconfigurations for establishing a covert channel was suggested by Thimmaraju et al. [30], by making one switch trigger the reconfiguration of rules in another switch; however their approach required at least one compromised switch.

2.2 Analogy to Caching

We observe that flow reconfiguration due to reported events in SDNs are similar to cache miss and eviction processes in operating systems and computer architecture. Several works [33],[10] have demonstrated how this cache miss and eviction when measured by one side and inflicted by the other can be used for covert and side channels across processes. For instance, in the Prime+Probe [21],[22],[23] to transmit a '1', the sender accesses cache lines mapped to a particular last-level cache set that contains receiver's cache lines. This will result in eviction of the receiver's cache lines from the last-level cache. Due to the inclusive property, these cache lines will be also evicted from L1 cache. Therefore, when the receiver tries to access its lines, it will observe a long probing time because the evicted lines have moved to memory. To transmit a '0', in the Prime+Probe attack, the sender does not have to do anything. As a result, the receiver can probe its cache lines in a short time because they are present in the L1 cache sets of the receiver.

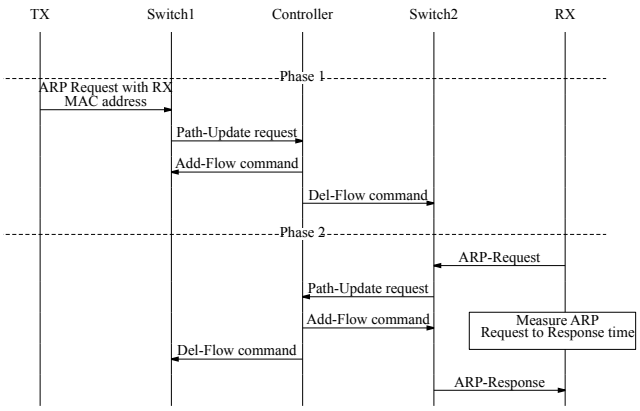

Figure 3: The messages sequence pattern of the 2-phase Macchiato covert channel.

2.3 Macchiato Core Idea

Macchiato is based on the observation that while hosts cannot send events and receive flow reconfiguration messages to/from the controller like switches, they still can trigger and detect flow reconfiguration indirectly by time measurements similarly to the processes performing Prime+Probe. Figure 3 illustrates the message sequence. The host on the sender side of the covert channel can trigger flow reconfiguration by faking host migration and the host on the receiver side can use RTT measurements for detection. We elaborate on these procedures in the next section and show that by performing the RTT measurements repeatedly and synchronously the hosts can covertly transmit long bit streams.

3 CHANNEL MODULATION

In this section, we elaborate on the algorithms used by the hosts to modulate their traffic covertly. Moreover, we discuss possible sources of error/noise and suggest ways to improve the effective bandwidth of the channel.

3.1 Sender and Receiver Algorithms

We focus on MAC learning and mobility applications which install MAC-based path rules between any two communicating hosts, based on their discovered (learned) locations.

When a host is rediscovered on another location, the immediately connected switch does not have an appropriate flow rule and the packet from the relocated host is forwarded to the controller. As a response the controller issues Flow-mod messages (OpenFlow parlance) to delete the old rules associated with that host as well as Flow-mod messages to configure new flow rules along the path from the new host location to the destination of the packet. Finally, the controller instructs the immediate switch to forward the reported packet according to the new path. In contrast when a host sends a packet from its last known location to a destination it already communicated, the relevant path is configured and the packet is immediately forwarded along it.

The difference between the packet forwarding time having a rule installed and having no rule installed is unavoidable, but may vary based on the processing power of the controller, the control protocol, and control plane RTT. As can be seen in Figure 2 this time difference is high enough for both P4runtime and OpenFlow and allows two compromised hosts to covertly communicate with each other using Algorithms 1 and 2 for sender and receiver respectively.

The sender and receiver algorithms use one round per sent bit. Each round consists of two phases. In the first phase, configured to last δ_1 seconds, the sender tests the to be sent bit; if the bit is 1 then it sends an ARP request to any arbitrary host (configured by parameter *s_dst*), even itself, using the Ethernet source address used by the receiver (configured by parameter *src*). During this phase the receiver waits to allow a possible reconfiguration, triggered by sender ARP request, to complete. In the second phase, configured to last δ_2 seconds, the receiver sends an ARP request to any reachable host (configured by parameter *r_dst*), including itself, using its own Ethernet address as source address or any other source address used also by the sender (configured by parameter *src*). Note that at least one of the endpoints need to use a source address different than its own and therefore needs to run in privileged mode.

After sending the ARP request, the receiver waits for a response and measures the RTT. The RTT is then compared to a threshold (parameter *thresh*) to identify if a reconfiguration was triggered by the receiver's ARP request indicating that another reconfiguration was previously triggered by the sender. Therefore, if the RTT is higher than the threshold, it is considered as receiving 1, otherwise as receiving 0. Both endpoints wait till the (second) phase is over to be synchronized for the next round.

Next we elaborate on how the modulation parameters δ_1, δ_2 and *thresh* are configured and calibrated.

Algorithm 1: Sender

```
Input: send_buf, len, δ1, δ2, src, s_dst
for i ∈ [len] do
    if send_buf[i] == "1" then
        Send ARP request src → s_dst;
    end
    Wait δ1 + δ2;
end
```

Algorithm 2: Receiver

```
Input: rcv_buf, len, δ1, δ2, thresh, src, r_dst
for i ∈ [len] do
    Wait δ1;
    Send ARP request src → r_dst;
    RTT ← Wait ARP response;
    if RTT ≥ thresh then
        rcv_buf[i] = "1";
    else
        rcv_buf[i] = "0";
    end
    Wait δ2 − RTT;
end
```

3.2 The Decision Threshold

From the receiver point of view, the observed RTT of an ARP request, $r \in \mathbb{R}$, can be resulted from two different conditional RTT distributions, given the sender sent 0 or given it sent 1, i.e., $P(R = r|S = s)$ where s is 0 or 1 respectively. Such distributions are depicted in Fig. 2, where the orange samples come from the distribution $P(r|S = 1)$, and the blue ones come from the $P(r|S = 0)$.

The receiver decision is based on a threshold (T), i.e. if r is greater than T, it is considered as 1, otherwise as 0. Therefore, the error probability can be defined as a function of the threshold:

$$P_e(T) = P(r \geq T|S = 0)P(S = 0) + P(r \leq T|S = 1)P(S = 1)$$

In order to minimize the error the receiver first learns the distributions of $P(r|S = 1)$ and $P(r|S = 0)$ using a calibration phase. In this phase, the sender will send a predefined sequence of ones and zeros, which the receiver is aware of, and the receiver will record the round trip time of the request that corresponds to each bit. Based on the aggregated data, the receiver can either fit a normal distribution to the data or use kernel density estimation to estimate the probability density function of the data.

Assuming the RTT distributions are Gaussian, the receiver can approximate them using the means μ_0 and μ_1 and variances σ_0^2 and σ_1^2 of the RTT samples for the 0 and 1 bits respectively. Considering an equal probability for sending ones and zeros, we get that that the receiver can minimize the error by using the following threshold:

$$T^* = \arg\min_T Q(\frac{T - \mu_0}{\sigma_0}) + Q(-\frac{T - \mu_1}{\sigma_1}) \quad (1)$$

where Q is the tail distribution function of the standard normal distribution, i.e., $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} dt$. We will evaluate this method in Section 4.

3.3 Optimizing the Channel Using Deep Neural Networks

While using a constant threshold to discriminate network behavior is a straightforward method to implement a covert channel, it can be an oversimplification. Due to the dynamic behavior of the computer networks, the distribution of $P(r|S = 1)$ and $P(r|S = 0)$ can change during the communication for a variety of factors (e.g. controller load, users' traffic pattern, available resources of malicious hosts, etc.) and the calculated threshold will not be valid for the whole communication procedure. This phenomenon can introduce a significant error to the covert channel.

However, while the RTTs can change, an observable difference remains between ARP requests that include flow reconfiguration to those without. This difference provides an opportunity to reduce error by encoding each bit by two transmission rounds, following the Manchester coding[6]. We defined a transition from a lower RTT in the first round to a higher RTT in the next round as a 0, and changing from a higher RTT in the first round to a lower RTT in the next round as a 1. This method decreases the error ratio significantly but it reduces the throughput by a factor of 2.

As a more effective and generalized approach, that was successfully used in cache-based side channels [8, 31], we consider to use machine learning to decode sent bits based on recent communication network conditions. Each training sample is defined by $< y_i, X_i >$, where $y_i \in \{0, 1\}$ is the transmitted bit (the label), and $X_i = \{x_{i-k}, x_{i-k+1}, \ldots, x_i, x_{i+1}, \ldots, x_{i+k}\}$ are the RTT measurements (the features). Note that x_i is the RTT for the round when bit i was sent, and x_{i-j} or x_{i+j} are the RTT of the jth round before or after the ith round respectively.

To classify the RTT features as one or zero, we used a deep neural network (DNN) consisting of 4 layers: three fully connected hidden layers and one soft-max layer at the output. We used dropout regularization after each layer to reduce over-fitting. The receiver trains its network at the calibration phase with a predefined sequence, and then uses this model to classify the features of each sample. The results in Section 4 show that this method can enhance the accuracy of the covert channel remarkably.

3.4 Parallelization

A relatively simple way to improve the bandwidth is to use multiple Macchiato instances in parallel, each is based on a different source MAC address. However, using of multiple MAC addresses by sender and receiver requires them both to run in privileged mode. Moreover, generating high rate of path updates increases the load of the switches and controller and may affect the response time and in turn also the error rate and the actual bandwidth.

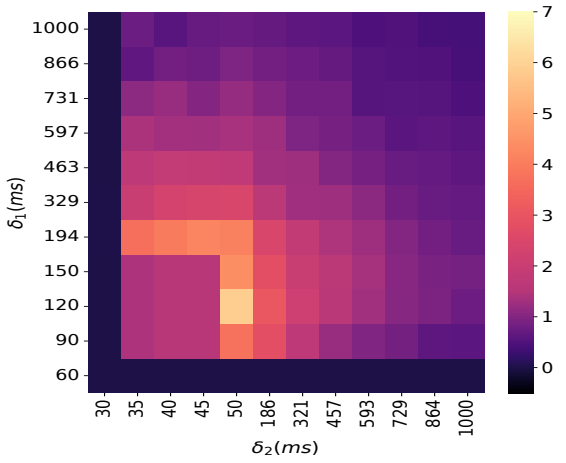

Figure 4: Heat map for *Effective Throughput* (bps) w.r.t timing intervals (δ_1,δ_2).

4 EVALUATION

We evaluate Macchiato performance considering different time parameters for the rounds, modulation schemes and controller load conditions. The evaluation is based on one server running mininet for simulating a data plane with 20 switches and hosts, and second server running the controller managing the (virtual) switches. We implemented the sender and receiver algorithms using Python scripts that execute the ping and nping programs according to pre-configured phase time intervals and source MAC address.

4.1 Timing Intervals Analysis

As described in Section 3, each modulation round consists of a sending phase and a receiving phase, which last δ_1 and δ_2 seconds respectively. Clearly, reducing these phase time intervals will increase the rate of rounds, but it also inflicts errors when the intervals are shorter than flow reconfiguration and forwarding times. Therefore, we define the effective throughput of the channel by $T := \frac{C_{\epsilon_0,\epsilon_1}}{\delta_1+\delta_2}$, where $C_{\epsilon_0,\epsilon_1}$ is the capacity of binary asymmetric channel (BAC) with bit flipping errors ϵ_0 and ϵ_1 [19].

We evaluated the effect of the timing intervals by testing multiple δ_1,δ_2 pairs. For each pair we transmitted predefined 128 bits, measured the error rate and calculated the effective throughput. The results are illustrated in Fig 4. We can see that the maximal effective throughput (5.9 bits per second) is achieved by the pair $\delta_1 = 120ms, \delta_2 = 50ms$.

4.2 Control Plane Considerations

We explored the impact of controller load on Macchiato error. For the experiment we used ofcprobe [12], which is a platform-independent controller analysis tool, to inflict

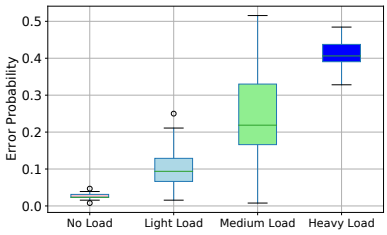

Figure 5: Estimated error per bit in different controller load conditions.

Table 1: Comparison of the error rate for different encoding and detection methods.

Encoding Method	Detection Mechanism	Bit Rate (bps)	Probability of Error		Effective Output (bps)	
			No load	Load	No load	Load
PAM	Threshold	5.9	0.004	0.135	5.7	2.5
	DNN	5.9	0.0	0.049	5.9	4.2
Manchester	Edge-Sensitive	2.9	0.0	0.02	2.9	2.5

load on the controller. We defined four scenarios based on the average packet-in message rate sent by each switch to the controller (on top of Macchiato related traffic): No Load, Light Load - 2 packet-ins/sec, Medium Load - 5 packet-ins/sec and Heavy Load - 20 packet-ins/sec.

In Fig. 5 we can see that the higher load on the controller the higher the error. Moreover, we compared RTT histograms between P4 and OpenFlow and considered the no load and the medium load scenarios. We can see in Fig. 2 that as expected, the load increases the RTT average and variance.

4.3 Robustness

As we described in Section 3.3, network changes after the covert channel calibration may impact the error (and throuput). In order to evaluate the robustness of the different methods we performed an experiment where we calibrate (train) the channels in no load conditions and test their performance (effective throughput) in load conditions (equivalent to the medium load scenario described before). Testing consisted of sending 1024 bits and measuring the errors. Note that the Manchester methods has no learning and the test results are independent of train results.

As we can see in Table 1, DNN is the most robust modulation, keeping high throughput in load conditions even when trained in no load conditions. The Manchester modulation, while having lower error rate, has half the nominal rate as it uses two rounds per bit.

5 COUNTERMEASURES

Since our covert channel is based on control plane activity of the mobility application, it can be impacted by control plane security techniques such as Topoguard [11] or Sphinx [3] that can block or delay handling such activity. However blocking mobility messages will prevent the core benefit of mobility which is network ability to adapt to moving hosts. On the other hand, adding delay and prioritization to specific messages will just slow down our covert channel but will not mitigate it. Other security techniques provided by SDN hypervisors such as CoVisor [13], Flowvisor [26], FortNOX [24] can be used to bound the impact of mobility within network segmentation boundaries. However they require to maintain a dedicated policy in the hypervisor and do not apply to all controllers.

We suggest a mitigation technique that breaks the strong coherency imposed by the mobility application: the MAC forwarding at all switches are consistent with the last known location of that MAC address. As explained in Section 2, our channel sender exploits this coherency to revoke rules at another switch which the receiver can identify. We can break this coherency by replacing the rules revocation by a predefined rules idle-timeout, making irrelevant rules to be self-deleted by the switches without controller intervention.

6 DISCUSSION

The analogy between memory hierarchy in computer systems and network layers in the software-defined networks provides us with the opportunity to benefit from the efforts of researchers in that area. For instance, Maurice et al. [17] explored two major sources of error in cache-based covert channels. The first one is eviction of the receiver's cache sets by other programs thereby increasing the receiver prob time and inflicting bit substitution errors. The second source of error are hardware and scheduling interrupts that can reschedule either one of the sender or receiver processes inflicting burst errors.

Similar error sources applies to Macchiato; Other hosts may temporarily increase data plane and control plane load and even evict receiver flow rules in case of switch table depletion thereby affecting the RTT measurement and inflicting bit substitution errors. Moreover, data plane and control plane failures may prevent or introduce new flow reconfigurations thereby inflicting burst errors.

As a future research direction, the study of access-driven cache covert channels, and subsequently, applying them to SDN-based covert channels, can help improve the security of software-defined networks, by either discovering potential attack vectors or finding new countermeasures.

Acknowledgements. Supported by the Austrian Science Fund (FWF), project DELTA (I 5025-N), a joint project with the Hungarian National Research, Development and Innovation Office (NKFIH), as well as by the Deutsche Forschungsgemeinschaft (DFG), project ADVISE (438892507). The authors alone are responsible for the content of the paper.

REFERENCES

[1] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. 2014. On detecting co-resident cloud instances using network flow watermarking techniques. *International Journal of Information Security* 13, 2 (2014), 171–189.

[2] Jiahao Cao, Kun Sun, Qi Li, Mingwei Xu, Zijie Yang, Kyung Joon Kwak, and Jason Li. 2019. Covert Channels in SDN: Leaking Out Information from Controllers to End Hosts. In *International Conference on Security and Privacy in Communication Systems*. Springer, 429–449.

[3] Mohan Dhawan, Rishabh Poddar, Kshiteej Mahajan, and Vijay Mann. 2015. SPHINX: Detecting Security Attacks in Software-Defined Networks.. In *Proc. NDSS*.

[4] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. 2021. Orion: Google's Software-Defined Networking Control Plane. In *Proc. NSDI*. 83–98.

[5] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 315–328.

[6] R. Forster. 2001. Manchester encoding: Opposing definitions resolved. *Engineering Science and Education Journal* 9 (01 2001), 278 – 280. https://doi.org/10.1049/esej:20000609

[7] Yuri Gbur. 2018. *A Feasibility Study of SDN Teleportation in P4Runtime*. Bachelor's Thesis. Technische Universität Berlin. https://github.com/yurigbur/publications/blob/master/Bachelorthesis.pdf.

[8] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[9] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[10] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 897–912. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss

[11] Sungmin Hong, Lei Xu, Haopei Wang, and Guofei Gu. 2015. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proc. NDSS*.

[12] Michael Jarschel et al. 2014. OFCProbe: A platform-independent tool for OpenFlow controller analysis. In *Proc. IEEE International Conference on Communications and Electronics*. IEEE, 182–187.

[13] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. 2015. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *Proc. NSDI*.

[14] Robert Krösche, Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. 2018. I DPID It My Way! A Covert Timing Channel in Software-Defined Networks. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. 217–225. https://doi.org/10.23919/IFIPNetworking.2018.8696597

[15] Qi Li, Yanyu Chen, Patrick PC Lee, Mingwei Xu, and Kui Ren. 2018. Security policy violations in SDN data plane. *IEEE/ACM Trans. Networking* 26, 4 (2018), 1715–1727.

[16] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 46–64.

[17] Clementine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Proc. NDSS*. https://doi.org/10.14722/ndss.2017.23294

[18] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (2008), 69–74.

[19] Stefan M. Moser, Po-Ning Chen, and Hsuan-Yin Lin. 2012. *Error Probability Analysis of Binary Asymmetric Channels*. Technical Report. National Chiao Tung University, Department of Electrical Engineering. https://moser-isi.ethz.ch/docs/papers/smos-2012-4.pdf

[20] OpenFlow Spec. 2013. *openflow.org* (2013). http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[21] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications *(CCS '15)*. 1406–1418.

[22] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (San Jose, CA) *(CT-RSA'06)*. Springer-Verlag, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/11605805_1

[23] Colin Percival. 2005. Cache Missing for Fun and Profit. In *In Proc. of BSDCan 2005*.

[24] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. 2012. A Security Enforcement Kernel for OpenFlow Networks. In *Proc. ACM Workshop on Hot Topics in Software Defined Networking (HotSDN)*. 121–126.

[25] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. ACM Conference on Computer and Communications Security (CCS)*. 199–212.

[26] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. 2009. *Flowvisor: A network virtualization layer*. Technical Report. OpenFlow.

[27] Rashid Tahir et al. 2016. Sneak-Peek: High speed covert channels in data center networks. In *Proc. IEEE INFOCOM*. 1–9.

[28] The P4.org API Working Group. 20201. P4runtime. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. Accessed: 01-07-2021.

[29] Kashyap Thimmaraju et al. 2018. Taking Control of SDN-based Cloud Systems via the Data Plane. In *Proc. ACM Symposium on Software Defined Networking Research (SOSR)*.

[30] Kashyap Thimmaraju, Liron Schiff, and Stefan Schmid. 2017. Outsmarting Network Security with SDN Teleportation. In *Proc. IEEE European Security & Privacy (S&P)*.

[31] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. 2019. Pythia: Remote Oracles for the Masses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 693–710. https://www.usenix.org/conference/usenixsecurity19/presentation/tsai

[32] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 29–40.

[33] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom