CJAG: Cache-based Jamming Agreement

Establishing a covert channel between co-located VMs

Michael Schwarz and Manuel Weber

Graz University of Technology, Austria

Abstract. Cache-based covert channels are successfully able to circumvent the isolation between multiple tenants in the cloud. As co-located virtual machines run on the same hardware, and thus share the same hardware, it is possible to establish a communication channel through the cache. The communication itself has been thoroughly studied. However, the problem of detecting the other communication party and negotiating the communication channels is often neglected.
In this paper, we present *CJAG*, a technique to fully-automatically negotiate communication channels between co-located virtual machines. *CJAG* uses techniques used in wireless communication and applies them to cache-based communication channels. We implemented an open-source proof-of-concept tool that allows to test this technique both locally and on virtual machines. We show that our tool can reliable detect the other communication party on a co-located machine running on the Amazon cloud. Finally, we demonstrate that *CJAG* is able to establish 6 binary communication channel between two virtual machines in less than 0.87 s.

1 Introduction

Recent research has shown that caches can be used as a basis for covert channels in the cloud. A covert channel is a channel not intended to transfer information [9]. It allows to secretly transfer data between two parties, a sender and a receiver. As cache-based covert channels do not require network access, they are not detectable by traditional network-based intrusion detection or prevention systems.

The basic principle of cache-based covert channels is to exploit the timing difference between cached and uncached data. As the cache is shared among all virtual machines running on the same physical host, these timing differences can be observed by all tenants. Applications cannot directly influence the content of the cache, however, they can fill the cache, leading to an eviction of data for other tenants. Evicting data from the shared cache is the basic technique to transmit information from the sender to the receiver. To accomplish this, most covert channels use the so-called Prime+Probe cache attack [7, 10, 13]. As caches are fast, this technique allows to build high-throughput covert channels.

A covert channel between virtual machines effectively undermines the isolation properties guaranteed by the hypervisor. Thus, several countermeasures have been proposed to prevent such covert channels [2, 3, 14, 15]. The majority

of the countermeasures relies on injecting noise into the cache to disrupt the communication. However, Maurice et al. [12] showed that despite such countermeasures, it is still possible to build a reliable covert channel by applying error detection and error correction. Such a robust covert channel achieves error-free transmission rates of several hundred kilobits per second in the Amazon cloud.

Most of the research so far focused on the actual transmission and the achievable transfer rates. Actually establishing the communication is often considered out of scope, simply an engineering task, or a trivial prerequisite. Maurice et al. [12] proposed the idea of cache-based *jamming agreements* (JAG) to establish a communication channel between the sending and receiving party. This approach is inspired by an existing technique from the field of wireless communication, JAG [1]. The main idea is to generate a large amount of noise that stands out from the noise floor and can thus be detected reliably by the other communication party, even under heavy noise. Maurice et al. [12] demonstrate that it is possible to apply JAG to cache covert channels.

In this paper, we give a more thorough explanation of *CJAG*, the cache-based jamming agreement. *CJAG* is both a technique and an open-source tool[1] to establish a cache-based covert channel. With *CJAG*, sender and receiver can automatically locate each other, both natively as well as inside co-located virtual machines. We explain how unprivileged parties inside virtual machines can take advantage of large pages to efficiently build eviction sets for Prime+Probe. We show that locating cache sets and the channel negotiation process work on state-of-the art CPUs as well as on cloud providers such as Amazon. *CJAG* is able to establish 6 binary communication channels in less than 0.87 s.

Contributions. The contributions of this work are:

1. *CJAG* is a technique to automatically locate communication parties in the cloud. It is able to establish an arbitrary number of parallel binary communication channels.
2. We provide an open-source implementation which works on state-of-the art native hardware, within co-located virtual machines, and in the cloud.
3. We show that *CJAG* is very efficient and robust, even if the system is under heavy stress. We are able to establish 6 binary communication channels within 0.87 s.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on caches and cache-based covert channels. In Section 3, we describe the design of *CJAG*. In Section 4, we present the open-source implementation of *CJAG*. In Section 5, we evaluate the performance of *CJAG*. We conclude in Section 6.

[1] The source can be found on GitHub: `https://github.com/IAIK/CJAG`

2 Background

2.1 CPU Caches

As CPU speed increases at a significantly higher rate than main memory speed, there is need for a fast intermediate memory. Modern Intel CPUs contain three levels of such intermediate memory, called caches. The caches follow a strict hierarchy, from the fastest and smallest L1 cache close to the CPU, to a slower but large last-level cache (LLC), furthest away from the CPU. A cache is divided into *sets*, where the *set index* is determined by the physical address bits 6 to 16. Furthermore, every set contains multiple *ways*. The data is stored in one of the ways, where the way is determined by the cache replacement policy. Older CPUs used least-recently used (LRU) as a replacement policy, however for newer models the exact replacement policy is not known [4].

The last-level cache in Intel CPUs is inclusive, *i.e.*, all data that resides in L1 and L2 must also be in the last-level cache. Furthermore, the last-level cache is shared among all cores, thus a modification of the last-level cache can influences the core-local L1 and L2 caches. Since the Sandy Bridge microarchitecture, the last-level cache is sub-divided into slices. To increase the bandwidth, there is one cache slice per CPU core. A CPU core can also access a remote slice via a ring bus, however such a remote access has a higher latency.

To determine the cache slice of a physical address, the CPU uses an undocumented hash function. However, this hash function has already been reverse engineered for various Intel CPUs [5, 6, 8, 11].

2.2 Prime+Probe

Prime+Probe is an access-driven cache attack [7, 10, 13] that allows to monitor the cache activity of a victim process. A Prime+Probe attack consists of two steps. First, fill a cache set with data (*prime*) and schedule the victim process. Second, access the data in the set and measure the access time (*probe*). If the victim process accessed data that maps to the same cache set, the data of the attacker process is evicted. Thus, the attacker measures a higher access time as parts of the data have to be fetched from the main memory. In contrast, if the victim did not access any data mapping to the same cache set, the attacker measures a low access time, as everything is still cached.

Prime+Probe does not require any form of shared memory and is thus applicable across virtual machines. To successfully mount a Prime+Probe attack, the attacker must be able to build an *eviction set*, *i.e.*, a set of addresses that occupies a complete cache set. Liu et al. [10] and Maurice et al. [12] developed a method to generate eviction sets by relying on large pages. Large pages have a size of 2 MB and thus the least significant 21 bits of the virtual address are the same as for the physical address. These bits are sufficient to calculate the cache set, however the cache slice cannot be calculated as the hash function to determine the slice depends on all physical address bits. Liu et al. use a brute-force approach to find the correct eviction set by continuously trying if the eviction set

Table 1: Cache slice function from Maurice et al. [11].

		Address Bit																													
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 slices	o_0						⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
4 slices	o_0							⊕	⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1				⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
8 slices	o_0		⊕	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕	⊕		⊕		⊕		⊕				⊕
	o_1	⊕		⊕	⊕	⊕		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕		⊕				⊕	
	o_2	⊕	⊕	⊕	⊕			⊕	⊕			⊕	⊕				⊕	⊕		⊕			⊕			⊕	⊕				⊕		

successfully evicts a victim address. Maurice et al. improved the method by using knowledge about the hash function to generate eviction sets more efficiently.

2.3 Jamming Agreement

Jamming Agreement (JAG) was developed by Boano et al. [1] as a robust handshake method for wireless sensor networks. Sensor networks commonly communicate within the unregulated ISM-bands, mostly the 2.4GHz band, and thus have to co-exist with WiFi, Bluetooth and several other technologies. Sensor modules, called sensor nodes, generally run on battery and therefore need to communicate efficiently. This requires establishing general communication channels for frequency hopping protocols or finding timeslots for time-slotted protocols. The receiver confirms a successful communication channel establishment by responding with an acknowledgment (ACK). If the ACK is not received, the channel is not established. Although ACKs have a high probability to be successfully received, they can still be lost due to heavy interference e.g., by WiFi file transfer. Boano et al. introduced JAG to increase the probability of successful ACK reception. Instead of sending an ACK, the carrier is modulated, *i.e.*, the noise floor is increased for a time period t by generating interference. This has to be long enough to distinguish this kind of interference from other noise sources, such as WiFi, which is—although stronger—much shorter. If the communication partner can sense that the carrier does not return to the usual noise floor during period t, this is interpreted as an ACK using the JAG technique.

3 *CJAG* Technique

In this section, we describe the technique of *CJAG*. We describe how to build eviction sets without knowledge of the cache slice. Furthermore, we show how sender and receiver can agree on common cache sets without knowledge of physical addresses.

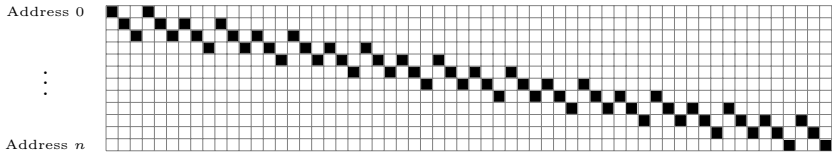

Fig. 1: The access pattern to the addresses in the eviction set.

3.1 Constructing Eviction Sets

A first prerequisite for a Prime+Probe-based covert channel is a method to generate eviction sets for cache sets. Maurice et al. presented the following method to construct eviction sets using large pages.

1. For the eviction set, we can only use one address per 4 KB page, otherwise the hardware prefetcher might prefetch addresses from the eviction set. Thus, we have to calculate the number of addresses on a 2 MB page that map to the same cache set, same cache slice, and are on different 4 KB pages. A 2 MB page contains $\frac{2\,\text{MB}}{4\,\text{KB}} = 512$ different 4 KB pages. The cache set index is determined by the (physical) address bits 6 to 16. We consider only addresses with some arbitrary but fixed bits 0 to 11, resulting in 32 cache sets containing only addresses with a distance of 4 KB to each other. The number of addresses in each of these cache sets is $\frac{512}{32} = 16$. Thus, the number of addresses per 2 MB page usable as an eviction set is therefore $nr_{addr} = \frac{16}{slices}$.
2. To fully evict a cache set, we require as many addresses as there are cache ways. Thus, the number of required 2 MB pages is $\left\lceil \frac{ways}{nr_{addr}} \right\rceil$.
3. We iterate in 4 KB steps over the allocated pages. For every address, we calculate a *virtual cache slice* by applying the cache slice function (cf. Table 1) to the least significant 21 bits of the address. All addresses within a 2 MB page that have the same virtual cache slice have the same (real) cache slice as well. However, virtual cache slices differ between different 2 MB pages by a constant offset depending on the physical address.
4. For all pages, we add all addresses mapping to the correct cache set to the eviction set, regardless of the virtual cache slice. This results in an eviction set containing all correct addresses, but also too many addresses, *i.e.*, addresses mapping to the correct cache set but to the wrong cache slice. Thus, for the addresses of every 2 MB page, we remove all addresses with the same virtual cache slice as long as the eviction set still works. Finally, we are left with a minimal eviction set for a specific cache set.

For older CPUs, it was sufficient to access every address in the eviction set once to fill the cache set with these addresses and evict everything else. However, modern CPUs use an undocumented eviction strategy and simply accessing all addresses from the eviction set is not sufficient anymore. We use the experimentally discovered access pattern by Gruss et al. [4] as shown in Figure 1.

Using this method combined with the access pattern allows us to efficiently create eviction sets without ever determining the real cache slice. However, this is also a disadvantage: As we do not know the correct slice, we cannot simply use the same cache set and slice for sender and receiver.

3.2 Negotiating Cache Sets

The second prerequisite for a Prime+Probe-based covert channel are common cache sets between sender and receiver. The receiver listens to the agreed cache sets using Prime+Probe. To transmit a binary '0', the sender does nothing and the receiver measures a low access time. For a binary '1', the sender evicts the cache set and thus the receiver measures a high access time.

Agreeing on the same cache sets on both sender and receiver side beforehand poses two problems. First, we cannot determine the real cache slice of the addresses. Second, some of the sets might not be usable as a stable communication channel as there is too much noise from other applications. Thus, sender and receiver require a dynamic method to negotiate cache sets to use for their communication.

Maurice et al. presented the following method for cache set negotiation.

1. Both sender and receiver start at the same (fixed) cache set to generate eviction sets for this and the following cache sets.
2. The sender starts at the first cache set and evicts it for a certain period of time t_s by accessing the eviction set (*jamming*). Then, the sender measures the access time to the set for a period of $2t_s$ to detect whether the receiver acknowledges the cache set by jamming back (*i.e.*, the receiver evicts the cache set).
3. The receiver measures the access time to one of its own cache sets for a period of t_r. If there are more cache misses than a specified threshold, the receiver acknowledges the set by evicting the set for $2t_r$ (jamming back). Otherwise, the receiver moves on to the next cache set.
4. If the sender receives the acknowledgment from the receiver, a common cache set for communication is found and the sender moves on to the next cache set. This procedure is repeated until sufficient common cache sets are found.

After this cache-based jamming agreement, both sender and receiver have the same number of eviction sets for the same cache sets. Thus, they can simply use Prime+Probe to transmit bits on these channels. Refer to Maurice et al. [12] on how to use the common cache sets as communication channel to build a robust covert channel.

4 *CJAG* Implementation

We provide an open-source implementation of *CJAG* which can be used as a basis for cache-based covert channels. The source and build instructions can be found on GitHub: `https://github.com/IAIK/CJAG`. In this section, we show the implementation details and how to use the implementation.

Function	Description
`int jag_init(cjag_config_t* config)`	Initializes a *CJAG* session based on the parameters in `config`.
`int jag_free(cjag_config_t* config)`	Cleans up a *CJAG* session.
`void jag_send(cjag_config_t* config, jag_callback_t cb)`	Function used by the sender. Every time a set is acknowledged, `cb` is called. After calling the function, the eviction set can be found in `config->addr`.
`void jag_receive(void **addrs, size_t* sets, cjag_config_t *config, jag_callback_t cb)`	Function used by the receiver. Every time a set is acknowledged, `cb` is called. `addrs` and `sets` have to be allocated by the caller. These variables contain the eviction set and set numbers respectively.

Table 2: The *CJAG* API.

4.1 API

CJAG provides a very simple to use API. Table 2 shows the API which consists of only 4 functions. The most important part is the configuration structure. This structure contains the parameters describing the last-level cache, timeouts, and the number of channels to establish. The hardest part is to provide the correct parameters to this structure. However, *CJAG* also provides an auto-detection mode for the last-level cache parameters (cf. Section 4.2).

If the configuration structure is correctly initialized, the cache set negotiation is straight forward. First, both sender and receiver have to call `jag_init` to allocate memory and generate cache sets and eviction sets.

The second step differs for sender and receiver. The sender calls `jag_send` to start the cache set negotiation. A callback function can be specified as optional parameter which—if given—is called after a set is acknowledged. The receiver calls `jag_receive` instead. The `addrs` and `sets` parameters have to be allocated before calling the function with a size of $ways \times channels \times$ sizeof(void$*$) and $channels \times$ sizeof(size_t) respectively. Again, a callback function can be specified as optional parameter which—if given—is called after a set is acknowledged. Both functions provide the eviction sets for the common cache sets after completion. They can be found in `config->addr` for the sender and in `addrs` for the receiver. *CJAG* provides a macro `GET_EVICTION_SET(addr, set, config)` for easy access to the eviction set addresses.

Finally, the *CJAG* sessions has to be cleaned up by calling `jag_free`. If the functions complete successfully, both programs share *channels* common cache sets with the corresponding eviction sets. These common sets can then be used to build the covert channel as described in Maurice et al. [12].

4.2 Parameter Detection

As setting the correct parameters for *CJAG* requires detailed knowledge of the processor, our implementation provides auto detection of the last-level cache

parameters. The `CPUID` instruction provides a function to get the last-level cache parameters. Specifically, CPUID-function 4 retrieves the "Deterministic Cache Parameters". The cache parameters include the number of cache ways, cache sets, and cache line size, allowing to furthermore calculate the size of the last-level cache.

CPUID does not provide the number of cache slices, however we can simply get that by parsing the number of physical CPU cores from `/proc/cpuinfo`.

One disadvantage of the automatic detection is that it might be wrong inside virtual machines, as the CPUID function might be emulated. However, the *CJAG* auto detection can be run on the host once to discover the parameters. When running *CJAG* inside virtual machines, the beforehand discovered parameters can simply be hardcoded into the application.

4.3 Application

We provide a demo application to test *CJAG* on native and virtual machines. The demo application implements both the *CJAG* technique as well as the parameter auto detection. When running natively, it can simply be started as `./cjag` for the sender and `./cjag -r` for the receiver. All parameters should either be auto detected, or at least be set to a sane default value.

Figure 2 shows the combined output of running *CJAG* in sender and receiver mode. At startup, *CJAG* displays the last-level cache parameters (①) which are either auto detected or given via the command-line arguments. A description of all available command-line arguments is shown using `./cjag --help`.

In the first phase (②), the sender jams the candidate cache sets, while the receiver listens to the candidate cache sets. After a cache set is acknowledged by the receiver, the sender continues with the next cache set until all cache sets are negotiated. The receiver displays the mapping (③) between the cache set names as seen by sender and receiver. At this point, the actual jamming agreement is already completed. However, to validate the cache sets, sender and receiver switch roles and test the common cache sets.

To verify the common cache sets (④), the receiver jams the common cache sets, and the sender switches to listening mode. If the sender measures activity on the same cache sets that were used to in the first phase, the common cache sets can indeed be used, and the verification succeeds. At this point, the sender has a proof that the common cache sets can be used as communication channels.

Common Errors. If *CJAG* does not work, this can have various reasons. As the Prime+Probe side channel requires the correct parameters to work reliably, *CJAG* provides multiple parameters that can be tweaked. The following checklist lists some common errors.

Cache parameters. It is of utmost importance that the last-level cache parameter are correct. If only a single parameter is wrong, the communication will most likely not work. Note that the auto detection will probably not work inside a virtual machine.

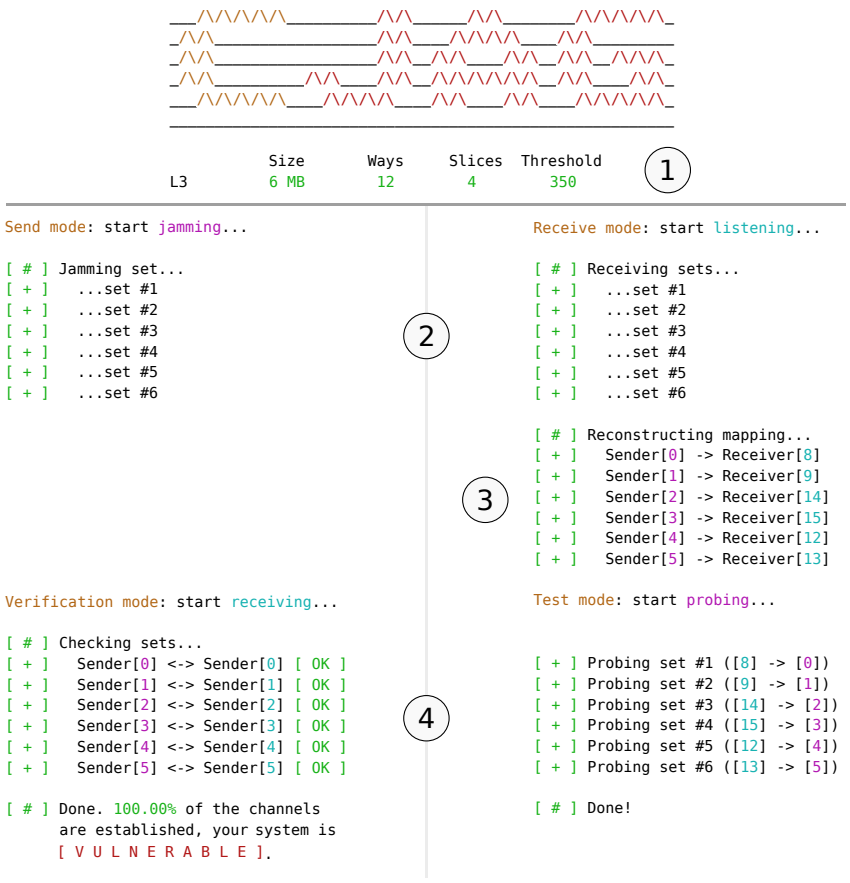

Fig. 2: Output of sender (left) and receiver (right) for a successful negotiation of 6 cache sets.

Environment	CPU	LLC Size	Ways	Slices	Threshold	Delay
HP ProBook 470 G0	i5-3230M	3 MB	12	2	290	≥ 0.1
Lenovo Thinkpad T460s	i7-6200U	3 MB	12	4	280	≥ 2
Lenovo Thinkpad W530	i7-3630QM	6 MB	12	4	350	≥ 0.05
Amazon EC2 (G2)	Xeon E5-2670	20 MB	20	8	280	≥ 0.1

Table 3: Test environments with the corresponding parameters.

Cache miss threshold. The cache miss threshold is the minimal number of cycles at which a data access is classified as a cache miss. If the receiver receives cache sets although the sender is not running, increase this value. In contrast, if the receiver never receives cache sets, decrease this value. Depending on the hardware, this value will usually be between 180 and 500.

Speed. If sender and receiver lose synchronization during the cache set agreement, the timeouts might be too low. *CJAG* provides a `--delay` parameter to increase the timeouts by the given factor.

Huge pages. Huge pages might not be enabled inside a virtual machine. For example, for VirtualBox, huge page support has to be enabled for a virtual machine by running `VBoxManage modifyvm <VM Name> --largepages on`. Most cloud providers, such as Amazon, have them enabled by default.

Co-location. *CJAG* only works across virtual machines if they run on the same physical host. Furthermore, each virtual machines requires at least one CPU core.

5 Evaluation

We show that *CJAG* works on native machines as well as across virtual machine borders. Table 3 shows a selection of the environments we used to test and the corresponding parameters. On native machines, *CJAG* takes on average only 0.11 s to establish and verify 6 binary communication channels. When sender and receiver are running in two co-located virtual machines, it takes on average 0.87 s to establish and verify the channels.

We successfully tested *CJAG* on all CPU microarchitectures, starting from Sandy Bridge up to Skylake. We evaluated the cross-VM functionality on VirtualBox on our native machines as well as on the Amazon cloud. *CJAG* is resistant against average system noise, e.g., a user surfing on the internet. *CJAG* can also cope with noise levels that are above average (cf. Maurice et al. [12]).

Limitations. *CJAG* has some known limitations. Some of them cannot be solved at the moment, some will be resolved in future work. First, the number of compatible CPUs is somewhat limited. Only Intel CPUs from Sandy Bridge to Skylake having 1, 2, 4, or 8 cores are supported. This limitations might not

be a problem on consumer PCs, however, the majority of physical machines in the cloud has more CPU cores. Reverse engineering the cache slice functions for different numbers of CPU cores is future work.

Second, the maximum number of channels is currently hardcoded to 32. This should not be a severe limitations, however this hard limit will be removed in the future.

Third, the cache-parameter auto detection does not work reliably inside virtual machines. The reason is the way virtual machines emulate the `cpuid` function. We suggest to only use the auto detection for testing on native machines and not when running inside a virtual machine.

Finally, *CJAG* is not yet able to automatically detect the cache miss threshold. Thus, we provide a program `cachespeed`[2] that is able to measure the cache miss threshold. A guiding value for *CJAG*'s threshold value can be obtained from column "+ mfence" of row "L3 Miss" after running `cachespeed`.

6 Conclusion

In this paper, we thoroughly described *CJAG*, a technique for cache-set negotiation across virtual machines. *CJAG* was initially proposed by Maurice et al. [12] and used as the basis for a Prime+Probe cross-VM covert channel. We present a fast and reliable, open source implementation of *CJAG* which works on the majority of modern consumer CPUs as well as on cloud providers. Our implementation includes automatic eviction set generation and a noise-resistant cache set negotiation, as well as automatic detection of the most important cache parameters. *CJAG* requires only 0.11 s to establish 6 binary communication channels on a native machine and 0.87 s on Amazon. The straightforward API makes it an ideal basis for any cache-based covert channel, whether native or in the cloud.

Acknowledgments

We would like to thank Lukas Giner for the first *CJAG* protocol implementation.

References

1. Boano, C.A., Zuniga, M.A., Römer, K., Voigt, T.: Jag: Reliable and predictable wireless agreement under external radio interference. In: Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd (2012)
2. Brumley, B.B.: Covert timing channels, caching, and cryptography. Ph.D. thesis (2011)
3. Fuchs, A., Lee, R.B.: Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In: Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15) (2015)

[2] It can be found in the *CJAG* repository at `https://github.com/IAIK/CJAG`.

4. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)
5. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13 (2013)
6. Inci, M.S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., Sunar, B.: Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Tech. rep., Cryptology ePrint Archive, Report 2015/898, 2015. (2015)
7. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and its Application to AES. In: S&P'15 (2015)
8. Irazoqui, G., Eisenbarth, T., Sunar, B.: Systematic reverse engineering of cache slice selection in intel processors. In: Proceedings of the 2015 Euromicro Conference on Digital System Design (2015)
9. Lampson, B.W.: A note on the confinement problem. Communications of the ACM (1973)
10. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: IEEE Symposium on Security and Privacy – SP. pp. 605–622. IEEE Computer Society (2015)
11. Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A.: Reverse Engineering Intel Complex Addressing Using Performance Counters. In: Research in Attacks, Intrusions, and Defenses – RAID. LNCS, vol. 9404, pp. 48–65. Springer (2015)
12. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17 (2017), to appear
13. Percival, C.: Cache missing for fun and profit. In: Proceedings of BSDCan (2005)
14. Schmidt, W., Hanspach, M., Keller, J.: A case study on covert channel establishment via software caches in high-assurance computing systems (2015)
15. Zhang, Y., Reiter, M.: Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: CCS'13 (2013)