Leveraging Intel DCI for Memory Forensics

Tobias Latzo, Matti Schulze and Felix Freiling*

Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany

ABSTRACT

The Intel Direct Connect Interface (DCI) provides a JTAG debugging interface which allows to debug Intel x86 CPUs by merely plugging in a slightly modified USB cable without opening the chassis. DCI offers the possibility to halt CPU operations and arbitrarily read and write main memory. We therefore explore the possibility to leverage DCI for the forensic acquisition of main memory. We introduce DCILeech, a tool which allows to acquire system memory with high quality: In contrast to software-based acquisition tools, it does not alter memory contents and therefore guarantees full integrity. Moreover, due to its power to halt the CPU, memory snapshots acquired by DCILeech exhibit no traces of concurrent system activity and therefore can be considered atomic. On the downside, DCI must be enabled on the target system and DCI-based memory acquisition is slow. We therefore also explore other applications of Intel DCI such as its use in practice for digital forensic triage.

Pre-Print

1. Introduction

The analysis of the main memory (RAM) of desktop and server systems is of increasing importance in digital forensic investigations today because it contains data that cannot be acquired using classic hard disk forensics, e.g., running processes, network attached storage, open network connections, and hard disc encryption keys. In contrast to persistent storage, main memory is volatile and so a proper memory snapshot needs to be acquired before analysis. Since this often happens on a running system, e.g., using DMA, such memory snapshots may exhibit traces of concurrent system activity. Such violations of *atomicity* of a snapshot can lead to misinterpretations during analysis [32]. Furthermore, software-based memory acquisition tools that are widely used in practice need to be installed or loaded first and therefore necessarily alter the contents of memory, reducing the *integrity* of the resulting snapshot and further putting forensic soundness at risk [4].

Intel Direct Connect Interface (DCI) is an interface that allows debugging Intel CPUs using a USB 3 port without opening the chassis. It was introduced with Intel Skylake in 2015 [13]. The interface provides JTAG debugging capabilities to an x86 system even with a small budget. On such a low level, it is even possible to debug software that is not debuggable with common tools, e.g., the firmware, the System Management Mode (SMM) or the Virtual Machine Monitor (VMM). Before accessing memory, the CPU needs to be stopped, which is beneficial for atomicity. DCI even allows to stop the CPU and read from arbitrary CPU registers (including debug register), a possibility that is hard to achieve otherwise. It can therefore be used to break CPU-bound encryption [30] which holds the encryption key in special registers, an attack for which the injection of software was needed before [2]. Furthermore, one does not need to install any software on the target to use DCI. However, Direct Connect Interface (DCI) must be enabled on the CPU before it can be used.

1.1. Related Work

Forensic memory acquisition is a lively field in which many advances have been made [10]. Latzo et al. [26] surveyed the many different techniques that have been developed, including those based on software, DMA, or hypervisor technology. Intel DCI appears to be less well-explored with only a moderate number of publications in that area. Most notably, Goryachy and Ermolov [13, 14, 15, 16] pioneered the field of DCI research and demonstrated how to use Intel DCI to debug the CPU or Platform Control Hub (PCH). We are aware of only one other public talk on the use of DCI to perform firmware debugging on Intel CPUs [24]. Leveraging Intel DCI for forensic memory acquisition appears not to have been yet explored.

Forensic imaging using JTAG is more usual in the area of smartphones or other embedded devices. In 2006, researchers showed how to use the JTAG *boundary scan* to create a bitwise image of the memory of an embedded device [3]. This is still a popular technique when it comes to debugging Internet of Things (IoT) devices. Manufacturers of those devices often want to prevent reverse engineering of their products. So they do not label the corresponding pins, or alternatively distribute them over the whole PCB. The *JTAGulator* [17] helps to find the pin assignment. Furthermore, JTAG turned out to be beneficial for Android rootkit detection [19]. For this, the authors extracted memory of a smartphone's kernel memory and reconstructed it for further analysis.

1.2. Contribution

In this paper, we study the capabilities of low-level memory acquisition using Intel DCI. Our main contributions are as follows:

- We introduce *DCILeech* — the first low-level memory acquisition method that utilizes Intel's DCI. This technique allows to dump system memory and produce a

*Corresponding author

tobias.latzo@fau.de (T. Latzo); matti.schulze@fau.de (M. Schulze); felix.freiling@fau.de (F. Freiling)

ORCID(s):

memory snapshot that satisfies full atomicity and full integrity. No software installation on the target is required. DCILeech benefits from its compatibility to PCILeech [12], which we demonstrate in the evaluation.

- We show how to access the decrypted memory of Intel SGX enclaves with the Debug profile. Using DFx Abstraction Layer (DAL), we were able to access specially protected enclave memory in the Enclave Page Cache (EPC).
- We show how to break CPU-bound encryption using Intel DCI.
- We sketch how Intel DCI can be used for the digital forensic triage.

1.3. Outline

In Section 2 we give some technical background information that is necessary to understand this paper. We then explain the fundamentals of Intel DCI, e.g., how to enable it, in Section 3. In Section 4 we provide some insights into the implementation of *DCILeech*. The evaluation of DCILeech can be found in Section 5. A possible workflow of the digital forensic triage with Intel DCI is sketched in Section 6. Finally, in Section 7 we conclude and give some suggestions for future work.

2. Background

In the following, we give some necessary information about JTAG debugging, Intel Software Guard Extensions (SGX), and memory forensics necessary to understand the rest of this paper.

2.1. JTAG Debugging

Joint Test Action Group (JTAG) is used synonymously for the IEEE-1149.1 standard [28]. It is used for testing and debugging of Integrated Circuits (ICs). JTAG allows to test and debug ICs when they are already installed. Today, JTAG is mainly known for flashing and debugging microcontrollers.

One main component is the Test Access Port (TAP) which is often called *JTAG interface*. It comes with five data lines where four a mandatory. If one finds those pins on an IC, the chances are good to get a JTAG connection. The pins are labeled as follows:

- TDI: Data Input
- TDO: Data Output
- TCK: Clock
- TMS: Test Mode Select
- TRST: Reset (optional)

When it comes to x86 systems, JTAG debugging gets slightly harder. Some special ports and devices are necessary for JTAG debugging [27] and the corresponding devices and software licenses are rather expensive. Furthermore, one needs a mainboard that comes with such a port. Since 2015, Intel DCI makes low-cost debugging easily possible for x86 systems, which we utilize in this paper.

2.2. Intel SGX

Intel SGX extends modern x86-64 processors to protect user-mode code and data from higher privileged layers, such as the firmware, hypervisor or Operating System (OS) using so-called *enclaves*. Enclaves can be seen as isolated containers of a user application with encrypted memory. The Central Processing Unit (CPU) decrypts it using the *Memory Encryption Engine* and stores the decrypted pages inside a special EPC that is integrated into the CPU. Enclaves can only execute user mode code as their operation is quite restricted. For example, code running inside an enclave cannot directly call into another application or execute system calls to request kernel functionality. The privilege level of enclaves is, therefore usually compared to a conventional user application. Code can be verified to run inside an enclave using Intel's remote attestation mechanism. First, an application launches an enclave, which then attests its integrity and confidentiality to a server. Afterward, a decryption key is sent to the enclave, which is used to decrypt the actual payload. To deploy software inside an enclave with the *Release* profile requires an *attestation key* from Intel. There is also the *Debug* profile that allows to develop and debug enclaves [25].

2.3. Criteria for Memory Acquisition

Vömel and Freiling defined three criteria for forensically sound memory acquisition: *correctness*, *atomicity* and *integrity* [37]. According to these definitions, a memory snapshot is *correct* if the memory acquisition tool acquires the actual content of the memory. Correctness is a necessary criterion of memory acquisition approaches.

Due to the common interleaving of memory acquisition with normal system operations, memory images might exhibit traces of concurrent system activity. An example is that the memory snapshot might hold evidence of the effect of a particular system action but not for its cause. A memory snapshot is *atomic* if such inconsistencies do not arise. An atomic memory snapshot is equivalent to a snapshot that could have existed if the system would have been partially “frozen”. Pagani et al. [32] showed that non-atomic snapshots actually occur and contain many such inconsistencies that obstruct proper analysis. While atomicity is hard to quantify precisely [38], Gruhn and Freiling [18] argued that a snapshot's atomicity can be approximated by the time it takes to take the entire snapshot. For a black-box evaluation, Gruhn and Freiling [18] therefore chose to quantify the atomicity of a snapshot by the time between the acquisition of the first memory region and the last memory region.

Formally, a memory snapshot satisfies *integrity* if the content of memory is not changed after the time an analyst

Pre-Print

Table 1
Flags to enable Intel DCI.

Flag	*Value*	*Description*
`Debug Interface`	1	Enables Silicon debug features
`Debug Interface Lock`	0	Allows changes of the MSR
`Direct Connect Interface`	1	Enables DCI
`DCI Enable (HDCIEN)`	1	Indicates that DCI is enabled

decides to take a snapshot. According to Vömel and Freiling [37], integrity aims at quantifying the level at which the process of taking the snapshot changes the content of memory. Gruhn and Freiling [18] quantified integrity by measuring the average time over all memory regions from the start of the acquisition until the time when the memory region is acquired. We, however, follow the original intention of the definition of integrity that is proportional to the amount of memory changed by the acquisition approach. An approach that does not change any memory content therefore satisfies full integrity.

3. Intel Direct Connect Interface

In this section, we introduce the basics of Intel DCI. DCI allows low-cost closed chassis debugging. There are two possibilities on how to connect host and target. First, there is Intel Silicon View Technology (SVT) — a device connected between the host and the target. Furthermore, there is the possibility to directly connect host and target via a USB 3 debug cable. The debug cable we used is a USB 3 A-to-A cable [21]. It is similar to crossover network cables and can also be hand-crafted from an ordinary USB A-to-A cable.

Debugging using Intel DCI is possible using *Intel System Studio* which is also offered as a free trial version. This software is needed because it integrates the software stack that is used for JTAG debugging via DCI. Also, Python-based command line interfaces for the software stacks come with Intel System Studio.

3.1. Enabling Intel DCI

Intel DCI is a powerful debug feature that can access random data of random processes. All security measures, such as the ring privileges, are circumvented. For this reason, manufacturers usually disable Intel DCI. In this section, we show how to enable Intel DCI.

Basically, some flags are set in the firmware settings to activate Intel DCI. These flags are usually hidden in the firmware settings and written during the boot process into the corresponding registers. However, some firmware implementations let the user enable DCI in the firmware settings, e.g., some Intel NUC systems [34]. The settings of the different flags in the firmware to enable Intel DCI [8] are shown in Table 1. Table 2 describes the subsequent values in the two relevant CPU registers that need to be set to enable DCI debugging.

For some systems it appears to be possible to change those flags using the *mm* command [36] of the Unified Extensible Firmware Interface (UEFI) Shell [13]. In our experiments, however, this approach did not work. For this reason, we enabled DCI in the following way on our analysis machine (described below in Section 5.2):

1. The firmware was dumped from the EEPROM by directly connecting to it via Serial Peripheral Interface (SPI). Then, the corresponding clip was attached to the appropriate pins of a Raspberry Pi. We used *flashrom* [9] to perform the dump.
2. To modify the firmware values, we used the *AMI BIOS Configuration Program* [1]. The values were set in accordance to Table 1. This tool allows to read and modify even hidden firmware settings in the corresponding snapshot.
3. Afterward, the firmware is saved and flashed to the EEPROM using *flashrom*. Then, one needs to *Reset to Default* in the firmware settings. This is needed because the current settings are stored on the CMOS chip. Restoring to default causes that the values from the flash chip are used.

3.2. OpenIPC and DAL

When using Intel System Studio, one can choose between two providers for DCI debugging. The most recent versions use *OpenIPC*. Older Intel System Studio versions also support DAL. Basically, both offer software stacks for private JTAG implementations: IEEE 1149.1 and IEEE 1149.7 [13]. These interfaces can also be used from a Python command line interface.

For DCILeech we make use of the OpenIPC interface. The following line of Python code, for example, shows how to read from a register:

```
ipc.threads[0].arch_register(REGISTER)
```

So it is possible to read from debug registers or other special-purpose registers. This affects the security of CPU-bound encryption where the encryption key is kept in registers [30]. Similarly one can read from memory. One only needs to specify the corresponding *physical* address. Note that it is also possible to specify a desired *virtual* address.

The differences between OpenIPC and DAL are not well documented. One difference we know is that there is a library called ITP that comes with the Python frontend ITPII using the DAL software stack. With this library it is possible to read memory from Intel SGX enclaves (see also Section 5.5).

Pre-Print

Table 2
Fields in registers, when DCI is enabled [5].

Register	*Field*	*Value*	*Description*
IA32_DEBUG_INTERFACE	Enable (R/W)	1	Enables debug features
	Lock (R/W)	0	Unlocks the MSR
	Debug Occurred (R/O)	1	Status of Enable bit
ECTRL	DCI Enable (HDCIEN)	1	DCI Debug is enabled

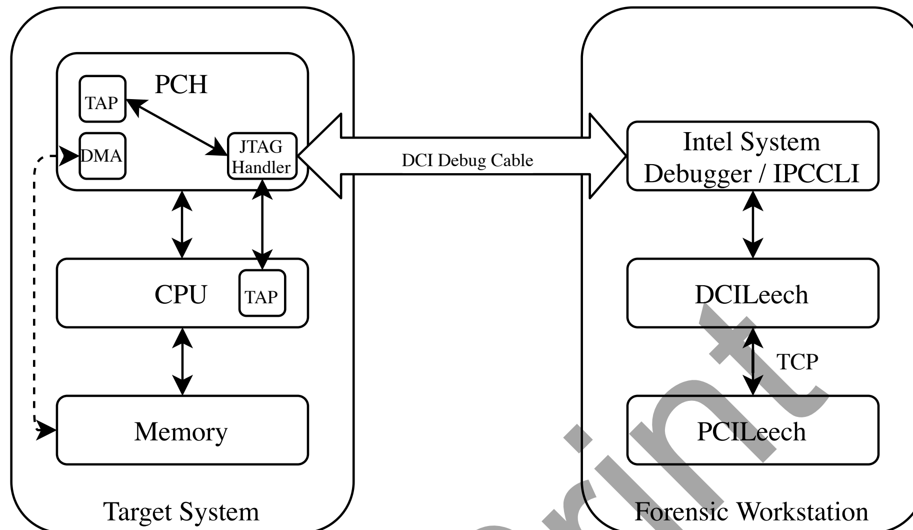

Figure 1: Schematic overview of the setup for DCILeech.

Pre-Print

4. DCILeech: Design and Implementation

In this section, we give some insights into the design and implementation of *DCILeech*. First, we provide a brief overview over the architecture. Then, we explain what steps were necessary to make DCILeech compatible to PCILeech [12].

4.1. Architecture

Figure 1 shows an overview of the setup we used. On the left side, one can see the *Target System* and on the right side the *Forensic Workstation*. *DCILeech* also benefits from *PCILeech*'s capabilities [11]. PCILeech is connected to the DCILeech server via a TCP connection, i.e., DCILeech is implemented as a PCILeech device. DCILeech itself uses the command line interface of Intel System Debugger which is able to perform debugging on the target system.

On the target system, the USB 3 port is connected to the PCH. The PCH comes with a JTAG handler which is connected to all TAPs (see also Section 2.1). All TAPs that are connected and enabled can be used for debugging the corresponding component.

4.2. DCILeech

DCILeech implements a PCILeech `rawtcp` device and can be started within the Intel System Studio GUI or the corresponding Python Debug Shell. It opens a TCP socket and waits for a connection. If a connection is established, DCILeech immediately halts the target's CPU. This is needed for OpenIPC to read and write memory. A pleasant side effect is that this allows atomic memory dumps. Afterward, PCILeech's requests are performed. The requests are sent as a `rawtcp_cmd` that is structured as shown in Listing 1.

```
enum rawtcp_cmd {
        STATUS,      // is device ready?
        MEM_READ,    // read from memory
        MEM_WRITE,   // write to memory
        DCI_GO,      // continue CPU
        DCI_HALT     // halt CPU
};

struct rawtcp_msg {
        enum rawtcp_cmd cmd;
        uint64_t addr;   // the address
        uint64_t cb;     // the length
};
```

Listing 1: The modified definitions of the `rawtcp_cmd` enum and `rawtcp_msg` struct.

If the status is requested, DCILeech always indicates that it is ready because everything is already initialized. If PCILeech sends a read request, DCILeech uses the `ipc.threads[0].memblock` function to read memory via DCI. If PCILeech wants to write to the host memory, DCILeech waits to receive the corresponding payload. Then, it is written to the physical memory of the target system. Therefore, also the `memblock`

function can be used.

4.3. PCILeech Patch

As one can see in Listing 1, we extended the `rawtcp_cmd` by two further commands:

1. `DCI_GO`: continues all CPU threads, and
2. `DCI_HALT`: halts all CPU threads.

DCILeech was designed to be compatible to PCILeech. However, PCILeech sometimes expects injected code to be executed before it can continue. This applies when it comes to kernel module injection. PCILeech waits for a specific physical address that is written by the injected code. Since the CPU is halted, this is never done. Thus, at this point the `DCI_GO` command is sent to let the CPU run and execute the injected code. After a second, the CPU is halted, again.

5. Evaluation

In this section, we evaluate how DCILeech performs regarding the three criteria of memory acquisition defined by Vömel and Freiling [37]: (1) Correctness, (2) atomicity and (3) integrity (see Section 2.3). Furthermore, we discuss the stealthiness (see Section 5.4) and demonstrate that we can read data of SGX enclaves (see Section 5.5).

Before reading memory, DCILeech halts the CPU, leading to *fully atomic dumps*. Note, this does not mean that the dump does not show any signs of "interruption". For example, if the CPU is halted during a critical write, this may lead to some inconsistencies stemming from the fact that some parts of the write have already been performed while others have not. However, while the chances are relatively low compared to a CPU that is running during the dump, halting the CPU will always avoid inconsistencies that violate causality (such as the effect of an activity is recorded but not its cause) [37].

DCILeech does not require any driver on the target system, and no code on the target system has to be executed. So we argue that snapshots acquired with DCILeech satisfy *full integrity*. Subsequent dumps with DCILeech therefore yield completely identical results.

In the following, we focus on the evaluation of the *correctness* of DCILeech.

5.1. Methodology

For the evaluation, we compare the physical memory acquired using DCILeech and LiME [35]. First, we dump the memory using LiME. Note, during the acquisition with LiME the CPU is running and writing memory. LiME operates from the kernel level and does not halt the system. Thus, the atomicity and integrity are limited and can probably be compared with the Windows kernel-level software acquisition tools that behave all similar in Gruhn's and Freiling's evaluation [18].

After the acquisition with LiME, we dump the memory using DCILeech. During the acquisition with DCILeech, the CPU is halted. Afterward, we calculate the diffs of the dumps. The dumps are compared byte-wise and page-wise (4 KiB).

Table 3
The memory ranges in our evaluation environment.

Start	*End*	*Size*
0x1000	0x9c3ff	621 KiB
0x100000	0x3fffffff	1032192 KiB

We also show that DCILeech works by demonstrating that PCILeech payloads do work properly.

5.2. Hardware Setup

For our experiments, we used a Fujitsu Esprimo Q957 with an i5-7500T (4 cores) with 8 GiB of Random-Access Memory (RAM) running Ubuntu with kernel version 5.4.0-42-generic on our target system. Additionally, for testing the compatibility with PCILeech payloads (see Section 5.3.2), we installed Windows 10 in dual boot.

However, for the evaluation, we had to limit the size of RAM using the `mem` parameter of GRUB to 1 GiB. This is because the speed of memory acquisition is low ($\approx$ 70 KiB/s) and we wanted to avoid having to wait several days for the outcome of an acquisition operation. It took more than four hours to acquire one gigabyte of RAM. Additionally, DCI debugging is not very stable. In our experiments, the acquisition regularly and annoyingly stopped because the system crashed.

Table 3 shows the memory ranges of our test setup in detail.

5.3. Correctness

A snapshot is considered to be *correct* if the acquired memory values are the values that are actually stored in memory [37]. For existing memory acquisition tools, correctness can be taken for granted [18]. However, DCILeech is a new acquisition tool, so we need to show that it is working correctly.

DCILeech uses hardware features that are not supported by any emulator we know. Thus we perform a black box evaluation. We do not know what the actual content of the physical memory is. So we use a LiME dump as a ground truth. The LiME dump is compared with the DCILeech dump. This allows a quantitative discussion of the correctness of DCILeech. Besides, we test different PCILeech features to demonstrate the compatibility of DCILeech.

5.3.1. Quantitative Analysis

In Figure 2 one can see the visualization of a page-wise (4 KiB) diff of a LiME dump and a DCILeech dump. Addresses are growing von the bottom left to the top right. Blue pixels indicate that there is no difference. The more reddish a pixel is, the more bytes are different in the corresponding page (4 KiB). On the right side, one can see the corresponding scale. Gray pixels indicate unmapped space, which is only hardly visible in the last row.

One can see that differing pages are spread over the memory space. More reddish areas can be found in upper mem-

Pre-Print

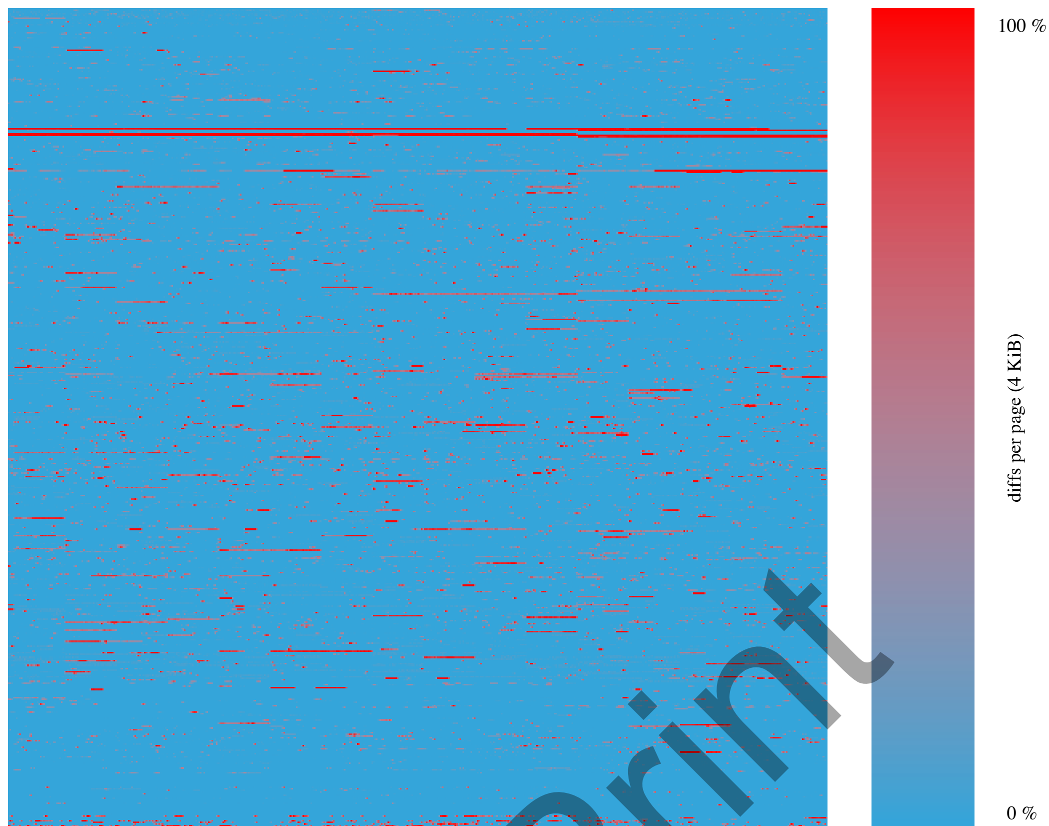

Figure 2: Visualization of the diffs of the memory snapshots taken by DCILeech and LiME. Blue pixels indicate no change of the corresponding page. The more reddish the pixel, the more bytes are different in the corresponding 4 KiB page.

Pre-Print

ory regions. A more detailed analysis revealed that about 50000 (18.76%) pages are different. However, a byte-wise comparison showed that in total, only 38 MiB (3.76%) are different. So, many pages are affected, but in total, not many bytes. So the quantitative analysis reveales that DCILeech appears to work correctly.

5.3.2. PCILeech Payloads

The DCILeech and LiME snapshots are relatively similar (96.24% byte-wise). Since DCILeech is implemented as a PCILeech device, it is also implementing *write* access to physical memory. To show that DCILeech is compatible with PCILeech, we tested some payloads. We are aware that writing memory is not forensically sound, but it can be useful during a live analysis. If memory is dumped beforehand, it might be beneficial to use some more advanced PCILeech features for the analysis.

For the evaluation of the compatibility to PCILeech, we successfully performed the following features:

- *Memory snapshot*

  The first feature is the basic memory acquisition feature `dump`. As parameter, `dump` expects the corresponding memory range. All our snapshots are made with this feature.

- *Kernel module injection*

  Since we can write to physical memory, we can exploit this to inject a kernel module into the host operating system via `kmdload`. PCILeech first searches for the Linux kernel base and is then able to inject the given kernel module. We injected a PCILeech kernel module that allows to more comfortably perform further analysis. Finally, the address of the kernel module is communicated to the analyst.

- *File retrieval and file pushing*

  This payload relies on the kernel module injection above. The injected kernel module is now used to pull files from the target system. For this, one needs the address where the PCILeech kernel module is loaded. Then, one can download the desired file using `lx64_filepull`. Pushing files was also performed using `lx64_filepush`. The file was then found on the target system.

- *Windows 10 unlock*

  This payload has been tested with Windows 10 as a target. The payload `wx64_unlock` searches in the memory for the code of the lock screen and is “shorting” the password query. So, one can log in with an empty password.

5.4. Stealthiness

For the user in front of the computer, debugging or memory acquisition using Intel DCI is not stealthy. First, the analyst needs to connect to the system via a USB cable. Furthermore, the CPU is halted, which looks like a freeze.

For the OS it is also possible to detect that it was debugged. After some experiments using Arch Linux, the following message appeared:

```
INFO: rcu_preempt detected stalls on CPUs/tasks:
[...]
NMI watchdog: Watchdog detected hard LOCKUP
on cpu [...]
```

Furthermore, it can be detected that DCI is enabled. The *CHIPSEC* framework comes with a DCI module that checks the registers listed in Table 2 [5]. If DCI is enabled, it is displayed in the corresponding report.

5.5. Intel SGX

In the following, we evaluate DCI's ability to read the memory of SGX enclaves. In order to do this, we wrote a small program using the *Intel SGX SDK* [23]. Note, the corresponding enclave is running in the Debug profile. In the enclave, we allocate memory and write the well-known Lena test image with a size of 88 KiB into the enclave's memory.

Now, we need to find the address of the EPC which contains the data. This can be done using *cpuid* [22]:

```
cpuid -l 0x12 -s 0x2
```

Now, it is possible to read the enclave's data via ITPII. The corresponding function is called `edbgread` [20] which dumps rather slow with about 4 KiB/s. After dumping some memory, we could find the image in the enclave's memory. Note that when reading from the EPC, OpenIPC returns `0xffffffffffffffff`. Reading from the corresponding address with LiME returns seemingly random values because this memory is encrypted.

This small experiment shows that we can read decrypted memory from SGX enclaves. Note, testing real-world SGX applications, i.e., with *Release* profile, was not in this work's scope. However, we think that the chances are good because there is the `set_debugoptin` function that can be called via ITPII [20]. This function sets the debug opt-in flag in the SGX Thread Control Structure that the enclave can be debugged. However, future work should consider reading the memory of SGX enclaves with Release profile.

6. Digital Forensic Triage with Intel DCI

Memory acquisition using Intel DCI is quite promising. However, it is hard to apply in practice. First, one needs a system that has Intel DCI enabled. Second, the acquisition speed is low. Our evaluation revealed that after hours of memory acquisition, system crashes become likely. In this section, we want to sketch possibilities of using Intel DCI for *digital forensic triage*.

Inspired by triage in medicine, digital forensic triage aims to prioritize the preservation of evidence [29]. In case of digital evidence, this means that the most volatile memory should first be saved. CPU registers can be regarded as the most volatile memory in a computer. Intel DCI allows reading registers without starting and loading special software which would overwrite register contents. System memory is also quite volatile and should also be acquired as fast as possible. In Figure 3 we propose a way of digital forensic triage with Intel DCI.

Probably the most challenging part of DCI-based memory acquisition is to enable DCI debugging. We found four ways to achieve this. First (1a), there are systems in the wild that have DCI enabled [31] which is actually a security vulnerability. Second (1b), some systems allow to enable DCI from the UEFI Shell [13]. For this, the computer needs to be restarted. The third possibility (1c) is to modify the system firmware as we did in Section 3.1. Another possibility (1d) is to exploit the firmware. Recently, malware security researchers observed TrickBot scanning for UEFI vulnerabilities that could allow malware to persist in UEFI in the future [6]. Using this technique, it should also be possible to enable DCI. Note, most of the techniques require restarting the target system, limiting the capabilities of Intel DCI for the digital forensic triage. However, researchers also showed that evil maid attacks should be considered [7].

If DCI is enabled, the analyst should first save all register contents (2), including the debug registers. This breaks CPU-bound encryption [30]. Next, if present, Intel SGX enclave memory should be saved (3). Since memory acquisition via DCI is slow, it is recommended to inject an acquisition software that exfiltrates system memory via network or a USB thumb drive. One needs first to save the pages that are later used for the acquisition software (4a). Then, the acquisition software can be injected (4b). The acquisition software should preserve atomicity. This means either the original threads must not be dispatched, e.g., by injecting a special OS that only dumps system memory. After injection, the context has to be set that all interrupts are received by the new OS. Another possibility is to virtualize the target OS on-the-fly [33].

Afterward, volatile memory is saved, and the investigator can start with the live analysis, including saving data from network-attached storage (5a). Eventually, the investigator can ascertain local storage (5b).

7. Conclusion and Future Work

In this paper, we introduced *DCILeech* that combines two powerful technologies: PCILeech and Intel DCI. No installation on the target is necessary. Furthermore, the CPU is halted. Hence, the OS cannot prevent a debug session, and the dump is performed atomically. We were also able to read from registers, e.g., `drX`, `xmmX` and `ymmX` which breaks CPU-bound encryption.

However, the evaluation in Section 5 also revealed some shortcomings. First, the acquisition speed is rather low, only

Pre-Print

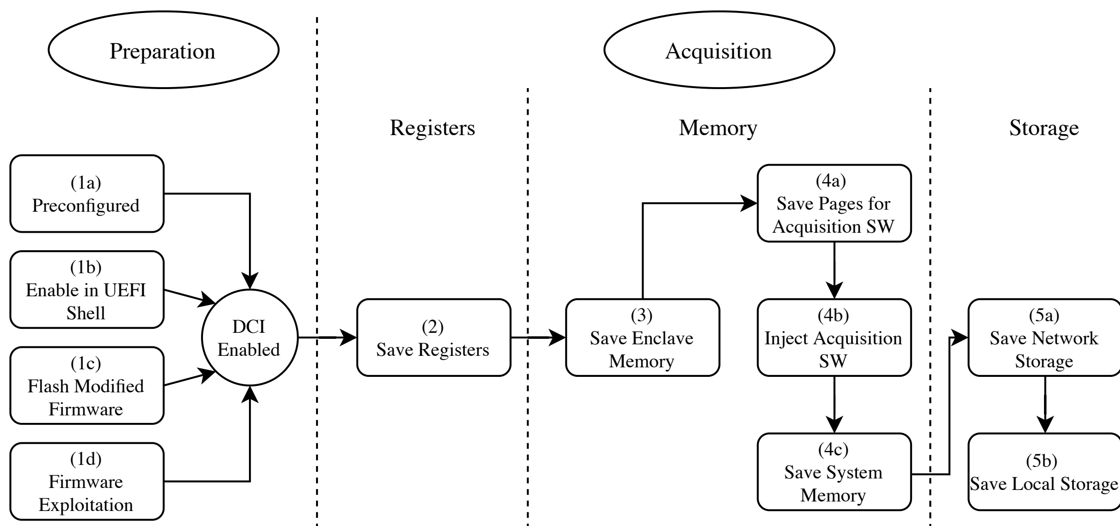

Figure 3: Digital forensic triage with Intel DCI.

about 70 KiB/s. We also had some crashes on the target side during long acquisition sessions. Future work should consider injecting an acquisition tool that can dump with more speed. Register contents and required memory pages can be dumped via DCI, beforehand. Hence, such a hybrid approach could also dump memory atomically. It only has to be guaranteed that other memory regions are not affected. This would probably be a terminating memory acquisition technique because the original OS must not be running when the acquisition is in progress.

Another problem is deployment. While no software needs to be installed on the target system, DCI debugging has to be enabled. For security reasons, manufacturers disable it. However, it also happens that it is enabled [31]. Our approach to activate DCI debugging might not be applicable for on-site forensic investigations. However, previous work showed that it is possible to enable DCI from the UEFI Shell [13]. Other researchers showed that an evil maid attack is also possible [7]. They modified the firmware of a computer in about four minutes.

Even though memory acquisition techniques using DCI are very beneficial in terms of integrity and atomicity, we would not recommend enabling DCI by default for forensic readiness. However, it might be beneficial to make it possible to enable DCI in a secured way. It is too powerful and can also be misused. However, it can also be used for offensive research because one can get "ground truths" and get insights into software components that are usually not accessible. Also, the ability to read volatile register values is unique.

Acknowledgments

This research is supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019).

CRediT authorship contribution statement

Tobias Latzo: Conceptualization, Investigation, Methodology, Supervision, Validation, Visualization, Writing - original draft, Writing – review & editing. **Matti Schulze:** Investigation, Software, Validation, Writing - review & editing. **Felix Freiling:** Conceptualization, Methodology, Supervision, Writing - original draft, Writing – review & editing.

Pre-Print

References

[1] American Megatrends Incorporation [2021], 'UEFI/BIOS Utilities'. Accessed: 2021-02-12.
URL: https://www.ami.com/products/firmware-tools-and-utilities/bios-uefi-utilities/

[2] Blass, E. and Robertson, W. [2012], TRESOR-HUNT: attacking cpu-bound encryption, *in* R. H. Zakon, ed., '28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012', ACM, pp. 71–78.
URL: https://doi.org/10.1145/2420950.2420961

[3] Breeuwsma, I. M. F. [2006], 'Forensic imaging of embedded systems using JTAG (boundary-scan)', *Digital Investigation* **3**(1), 32–42.
URL: https://doi.org/10.1016/j.diin.2006.01.003

[4] Casey, E. [2007], 'What does "forensically sound" really mean?', *Digital Investigation* **4**(2), 49–50.
URL: http://www.sciencedirect.com/science/article/pii/S1742287607000333

[5] Chipsec [2014], 'CHIPSEC: Platform Security Assessment Framework'. Accessed: 2021-02-04.
URL: https://github.com/chipsec/chipsec

[6] Constantin, L. [2020], 'TrickBot gets new UEFI attack capability that makes recovery incredibly hard'. Accessed: 2021-02-05.
URL: https://www.csoonline.com/article/3599908/trickbot-gets-new-uefi-attack-capability-that-makes-recovery-incredibly-hard.html

[7] Eclypsium Incorporation [2018], 'Eclypsium evil maid attack demo'. Accessed: 2021-02-05.
URL: https://www.youtube.com/watch?v=loBX%5FvEXxVA

[8] eiselekd [2018], 'Enable DCI debugging on Gigabyte-BKi5HA-7200'. Accessed: 2021-02-04.
URL: https://gist.github.com/eiselekd/d235b52a1615c79d3c6b3912731ab9b2

[9] flashrom team [2020], 'flashrom'. Accessed: 2021-02-04.
URL: https://www.flashrom.org/Flashrom

[10] Freiling, F., Groß, T., Latzo, T., Müller, T. and Palutke, R. [2018], 'Advances in forensic data acquisition', *IEEE Des. Test* **35**(5), 63–74.
URL: https://doi.org/10.1109/MDAT.2018.2862366

[11] Frisk, U. [2016], 'Rise of the machines: Direct memory attack the kernel'. Accessed: 2021-02-05.
URL: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEF%20CON%2024%20-%20Ulf-Frisk-Direct-Memory-Attack-the-Kernel.pdf

[12] Frisk, U. [n.d.], 'PCILeech'. Accessed: 2021-02-22.
URL: https://github.com/ufrisk/pcileech

[13] Goryachy, M. and Ermolov, M. [2016], 'Tapping into the Core', *33rd Chaos Communication Congress* .

[14] Goryachy, M. and Ermolov, M. [2017*a*], 'Inside Intel Management Engine', *34th Chaos Communication Congress* .

[15] Goryachy, M. and Ermolov, M. [2017*b*], 'Intel DCI Secrets', *The 8th Annual HITB Security Conference in The Netherlands* .
URL: https://conference.hitb.org/hitbsecconf2017ams/materials/D2T4%20-%20Maxim%20Goryachy%20and%20Mark%20Ermalov%20-%20Intel%20DCI%20Secrets.pdf

[16] Goryachy, M. and Ermolov, M. [2017*c*], 'Where there's a JTAG, there's a way: obtaining full system access via USB'. Accessed: 2021-02-05.
URL: https://www.ptsecurity.com/ww-en/analytics/where-theres-a-jtag-theres-a-way/

[17] Grand, J. [2020], 'JTAGulator'. Accessed: 2021-02-04.
URL: http://www.grandideastudio.com/jtagulator/

[18] Gruhn, M. and Freiling, F. C. [2016], 'Evaluating atomicity, and integrity of correct memory acquisition methods', *Digital Investigation* **16**, S1–S10.

[19] Guri, M., Poliak, Y., Shapira, B. and Elovici, Y. [2015], Joker: Trusted detection of kernel rootkits in android devices via JTAG interface, *in* '2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1', IEEE, pp. 65–73.
URL: https://doi.org/10.1109/Trustcom.2015.358

[20] Intel Corporation [2016], 'Intel DFx Abstraction Layer Python Command Line Interface'. Documentation is part of Intel System Studio.

[21] Intel Corporation [2020*a*], 'C01 - Intel SVT DCI DbC2/3 A-to-A Debug Cable 1 Meter'. Accessed: 2021-02-04.
URL: https://designintools.intel.com/SVT%5FDCI%5FDbC2%5F3%5FA%5Fto%5FA%5FDebug%5FCable%5F1%5FMeter%5Fp/itpdciamam1m.htm

[22] Intel Corporation [2020*b*], *Instruction Set Reference, A-L*, Vol. 2A, chapter 3.

[23] Intel Corporation [2020*c*], 'Intel Software Guard Extensions for Linux OS'. Accessed: 2021-02-06.
URL: https://github.com/intel/linux-sgx

[24] Jauregui, M. [2019], 'Intro to Closed Chassis Debugging', *2nd Open Source Firmware Conference* . Accessed: 2021-02-06.
URL: https://2019.osfc.io/uploads/talk/paper/18/Debugging%5FIntel%5FFirmware%5Fusing%5FDCI%5F%5F%5FUSB%5F3.0.pdf

[25] Johnson, S. P., Bombien, D. and Zimmerman, D. T. [2016], 'Intel SGX: Debug, Production, Pre-release – What's the Difference?'. Accessed: 2021-02-11.
URL: https://software.intel.com/content/www/us/en/develop/blogs/intel-sgx-debug-production-prelease-whats-the-difference.html

[26] Latzo, T., Palutke, R. and Freiling, F. [2019], 'A universal taxonomy and survey of forensic memory acquisition techniques', *Digital Investigation* **28**(Supplement), 56–69.
URL: https://doi.org/10.1016/j.diin.2019.01.001

[27] Lauterbach Gmbh [2020], 'Intel x86/x64 debugger'. Accessed: 2021-02-06.
URL: https://www2.lauterbach.com/pdf/debugger%5Fx86.pdf

[28] Mangel, M. and Bicchi, S. [2020], Jtag, *in* 'Praktische Einführung in Hardware Hacking', mitp Verlag, chapter 4.4.1, p. 106.

[29] Moser, A. and Cohen, M. I. [2013], 'Hunting in the enterprise: Forensic triage and incident response', *Digital Investigation* **10**(2), 89–98.
URL: https://doi.org/10.1016/j.diin.2013.03.003

[30] Müller, T., Freiling, F. and Dewald, A. [2011], TRESOR runs encryption securely outside RAM, *in* '20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings'.
URL: http://static.usenix.org/events/sec11/tech/full%5Fpapers/Muller.pdf

[31] NIST [2017], 'CVE-2017-5684,5,6'. Accessed: 2021-02-06.
URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5684,https://nvd.nist.gov/vuln/detail/CVE-2017-5685,https://nvd.nist.gov/vuln/detail/CVE-2017-5686

[32] Pagani, F., Fedorov, O. and Balzarotti, D. [2019], 'Introducing the temporal dimension to memory forensics', *ACM Trans. Priv. Secur.* **22**(2), 9:1–9:21.
URL: https://doi.org/10.1145/3310355

[33] Palutke, R., Ruderich, S., Wild, M. and Freiling, F. [2020], HyperLeech: Stealthy System Virtualization with Minimal Target Impact through DMA-Based Hypervisor Injection, *in* '23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)', USENIX Association, San Sebastian, pp. 165–179.
URL: https://www.usenix.org/conference/raid2020/presentation/palutke

[34] Soflen [2018], 'DCI Connection Problems'. Accessed: 2021-02-06.
URL: https://community.intel.com/t5/Intel-System-Studio/DCI-Connection-Problems/td-p/1160475

[35] Sylve, J. [2012], 'LiME'. Accessed: 2021-02-04.
URL: https://github.com/504ensicsLabs/LiME

[36] Unified EFI Forum [2008], 'UEFI Shell Specification'. Accessed: 2021-04-02.
URL: https://www.uefi.org/sites/default/files/resources/UEFI%5FShell%5FSpec%5F2%5F0.pdf

[37] Vömel, S. and Freiling, F. C. [2012], 'Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition', *Digital Investigation* **9**(2), 125–137.

[38] Vömel, S. and Stüttgen, J. [2013], 'An evaluation platform for forensic memory acquisition software', *Digital Investigation* **10**, S30 – S40. The Proceedings of the Thirteenth Annual DFRWS Conference.
URL: http://www.sciencedirect.com/science/article/pii/S1742287613000509

Pre-Print