On Reverse Engineering Hyper-V
- Praktikum Report -

Ștefan-Octavian Radu

Advisor: Lukas Beierlieb
Julius-Maximilians-Universität Würzburg
Institute for Computer Science Chair for Computer Science II
stefan.radu@stud-mail.uni-wuerzburg.de
https://se.informatik.uni-wuerzburg.de/

Keywords: Hyper-V · Reverse Engineering · Ghidra · QEMU/KVM · Windbg

1 Introduction

Hyper-V is the hypervisor solution shipped as part of the Windows operating system. Guests running under Hyper-V can communicate with the hypervisor through a calling mechanism similar to system calls (syscalls) in an operating system. One would refer to such calls as *hypercalls*. The Hypervisor Top Level Functional Specification (TLFS) documents, among other things, the functionality of the hypercall system and documents a number of hypercalls [8]. However, being a closed-source system, not all hypercalls are documented.

With the rising popularity of cloud solutions, securing the technologies that the cloud is built upon is more relevant than ever. Our motivation for this project is to contribute to the effort of security researchers who don't have access to the source code of Hyper-V. By analysing the undocumented hypercalls from Hyper-V, we can gain a better understanding of its inner workings, and potentially find bugs which would compromise its security. We will present a comprehensive description of a reverse engineering setup for Hyper-V running on the GNU/Linux operating system. We will also exemplify the usage of this setup by presenting our results from reverse engineering the `HvCallQueryNumaDistance` hypercall (number `0x78`).

1.1 Lab Environment Specifications

The analysis environment features an Intel Core i5-8250U CPU and 8GB of memory. The operating system running is based on the `6.7.6-arch1-1` version of the Linux kernel. Linux is a particularly unusual choice for working on the internals of the Windows operating system. We hope that presenting this setup will prove useful to other researchers who prefer using the Linux operating system. We will present our findings including shortcomings and roadblocks. The target to be analysed is the `hvix64.exe` executable, typically found in `C:\Windows\System32\hvix64.exe` on a machine running Windows. The version of our binary file is `10.0.19041.2006`.

1.2 Outline

The remainder of this work is structured as follows. In Sections 2 and 3 we discuss the static and dynamic analysis setup respectively. Section 4 is comprised of our findings regarding the `0x78` hypercall. We end with conclusions in section 5.

2 Static Analysis Setup

In this section we will cover the setup we used for statically analysing the Hyper-V hypercalls. In particular we will cover Ghidra [1], a powerful open-source disassembler and decompiler. We will mainly discuss project setup and automation via scripting. We will briefly mention other popular tools in the field of reverse engineering that could be used as alternatives to Ghidra.

2.1 Ghidra

The main tool used for static analysis is Ghidra. Ghidra is a Software Reverse Engineering (SRE) framework developed by the National Security Agency (NSA) of the United States. Ghidra includes a fully featured set of tools for analysing code compiled for a large variety of platforms, including Microsoft Windows. From the tool set, the features which were most useful for analysing Hyper-V include the powerful decompiler, the disassembler and the scripting interface available as a Java or Python API [1]. Although its interface is dated and not very intuitive to use, its feature set is very powerful.

Ghidra is distributed as Free and Open Source Software (FOSS) under the Apache 2.0 licence [2].

2.2 Setup

There are a number of useful steps which should be performed in order to more easily work on reverse engineering Hyper-V's components. We took inspiration from Microsoft's Introductory blog post on Hyper-V research, which has some very insightful content, but is unfortunately geared towards IDA [14].

Since we're running on a 64-bit Intel system, we will inspect the `hvix64.exe` *Hypervisor 2.0* binary. Since we're running a GNU/Linux system, we ran a Windows guest through the QEMU/KVM and pulled out the binary from the virtualised environment.

For the start, we create a new Ghidra project, import the binary and open it with the decompiler tool. *Here be dragons!*

Useful Imports. Before starting to inspect the hypercall handlers' code, we can introduce some helpful symbols into the Ghidra project. These include a number of important structures, enums, or constants. To do this, we acquired

the vmx.h [1] and hvgdk.h [2] header files. To import the definitions into the Ghidra project we go to File → Parse C Source, create a new Parse configuration and add the two headers in the list of Source files to parse. We encountered a number of errors on import, which could be fixed by adding missing definitions and rewriting some macros as follows:

- add #define C_ASSERT(e) typedef char __C_ASSERT__[(e)?1:-1] in vmx.h
- add typedef unsigned short UINT16; in hvgdk.h
- add typedef __int64 UINT64; in hvgdk.h
- change each DECLSPEC(X) to __declspec(align(X)) in hvgdk.h

We expect the previously mentioned errors to not be raised on a Windows system.

Defining Types. We further defined a number of useful types directly related to the hypercalls, such as the hypercall's structure and the hypercall enum group, among others. This can be done through Ghidra's *Data Type Manager*, which is typically found in the lower left corner, or by going to Window → Data Type Manager. Some IDA PRO scripts, such as this one[3], served as reference for the data types. Figure 1 can be inspected for a view of the Structure Editor.

With the types defined, we can start retyping global variables. The first big target is the hypercalls table from the CONST section. We can manually set the start address of the CONST section to be of type HV_HYPERCALL_TABLE_ENTRY[SIZE]. At the type of writing, 260 hypercall entries can be distinguished in the table. Each of the hypercalls returns HV_STATUS. Setting this return type for each available hypercall is very useful during analysis. However, doing the type setting manually is time consuming and impractical.

Scripting. Ghidra offers a scripting API in Java or Python for automating a wide array of reverse engineering tasks. We can automate all the types creation and type assignment by writing and running some Python scripts. Such scripts simplify the setup process considerably. Listings 1.1 and 1.2 exemplify data type creation and function signature editing. The full scripts are available on github[4].

```
struct = StructureDataType("HV_HYPERCALL_TABLE_ENTRY", 0)
struct.add(QWordDataType(), 8, "handler_routine_fp", "")
# similarly added field ...
struct.add(WordDataType(), 2, "rep_output_param_size", "")
dt_mgr = currentProgram.dataTypeManager
dt = dt_mgr.findDataType("/" + "HV_HYPERCALL_STATS_GROUP")
```

[1] https://github.com/ionescu007/SimpleVisor/blob/master/vmx.h
[2] https://github.com/ionescu007/hdk/blob/master/hvgdk.h
[3] https://github.com/gerhart01/Hyper-V-scripts/blob/master/CreatemVmcallHandlersTable2016.py
[4] https://github.com/Stefan-Radu/master/tree/master/an2_DE_erasmus/praktikum_hyperv_RE/hyper-v-ghidra-scripts

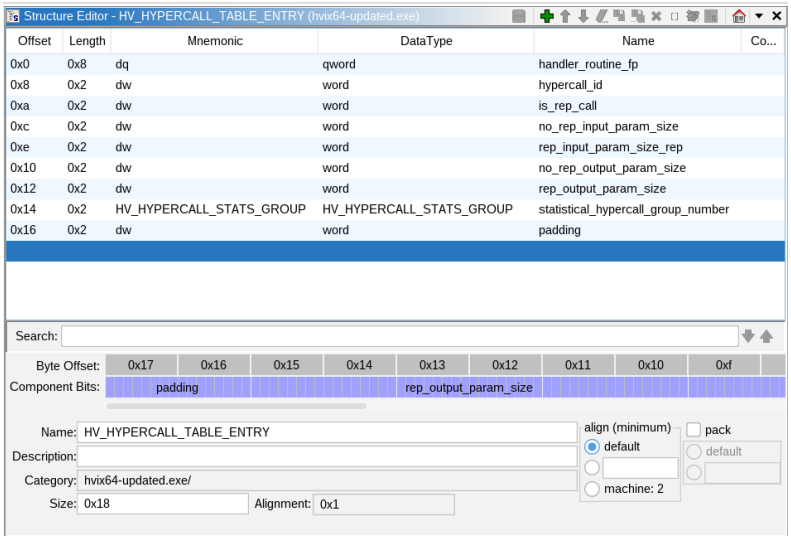

Fig. 1. A view of the `HV_HYPERCALL_TABLE_ENTRY` struct, which we defined through the Ghidra structure Editor. We used it to retype each entry in the hypercall table from the `CONST` section of the binary.

```
struct.add(dt, 2, "statistical_hypercall_group_number", "")
struct.add(WordDataType(), 2, "padding", "")
currentProgram.dataTypeManager.addDataType(struct, None)
```

Listing 1.1. Python script which creates the `HV_HYPERCALL_TABLE_ENTRY` structure.

```
hvcalls_entry_type = dt_manager
    .findDataType("/HV_HYPERCALL_TABLE_ENTRY")
hvcall_entry_size = hvcalls_entry_type.getLength()
create_array_cmd = CreateArrayCmd(start_address,
    260, hvcalls_entry_type, hvcall_entry_size)
create_array_cmd.applyTo(currentProgram)

hvcall_ret_type = dt_manager.findDataType("/HV_STATUS")
hvcalls_entry_table = getDataAt(start_address)
entry = hvcalls_entry_table.getComponentAt(
    IDX * hvcall_entry_size)
hvcall_address = toAddr(entry.getComponent(0)
    .getValue().toString())
hvcall = getFunctionAt(hvcall_address)
hvcall.setName("HV_HYPERCALL_NAME", SourceType.ANALYSIS)
hvcall.setReturnType(hvcall_ret_type, SourceType.ANALYSIS)
```

Listing 1.2. Example of array creation and changing the name and return type of a hypercall function reference.

A relevant note is that Ghidra is built in Java. The Python scripts run through `Jython`, which bridges the script code to the Java API. This adds an obvious layer of overhead. Moreover, the python interface is based on `Python 2`, which is outdated. This, in turn, limits the usage of external libraries. There are a number of plugins, such as [12], which address these issue, by introducing `Python 3` scripting capabilities.

Result. After running the scripts, the `CONST` section will be displayed in a more intuitive and easy to work manner. As seen in Figures 2 and 3, we get a clearly defined table. Inspecting a hypercall handler's implementation can be done by clicking on the corresponding field of an entry.

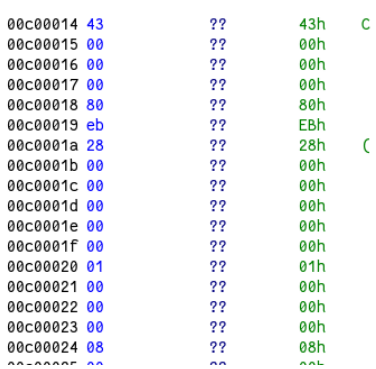

Fig. 2. The `CONST` section before setup is displayed as a random array of bytes.

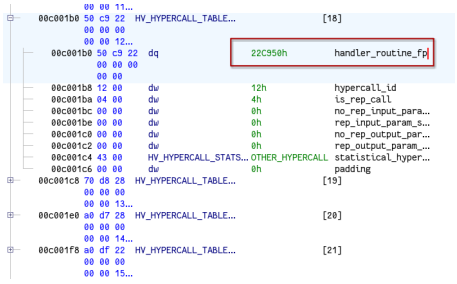

Fig. 3. After running the script, the `CONST` section is displayed as an array. We can jump to any of the hypercall handlers' code by clicking on the `handler_routine_fp` field of the corresponding entry.

Other Useful Features. Some other features that are by default hidden in Ghidra, but could prove useful include the: `Function Graph`, `Function Call Tree`, `Entropy Overlay`. In our case the python liner `pyright` did not provide autocompletion by default for the Ghidra API interface. We easily solved this issue by installing the `python-ghidra-stubs-bin` package from the Arch User Repository (AUR).

Other Tools. The main advantages of Ghidra over other tools are that it is fully featured and free at the same time. IDA is a professional tool and is probably the direct competitor of Ghidra. However, the high price tags of its variants prohibits its usage by a large number of researchers. There exists a free version of IDA, but it offers a severely limited set of features compared to the paid tiers. Another professional tool, Binary Ninja, is gaining a lot of traction in recent times, especially with its recent 4.0 release which looks very promising. Similarly to IDA however, it is a paid tool, but the price brackets are more accessible and they offer discounts for students. Radare2 is also a very powerful and open-source tool worth mentioning. It is designed as a command

line application, but graphical interfaces such as Cutter exist. What it lack is a powerful decompiler, but the interfaces well with Ghidra decompiler.

3 Dynamic Analysis Setup

In this section we will cover the dynamic analysis setup we used for analysing the Hyper-V hypercalls. We will discuss the setup needed to debug the windows kernel and the Hyper-V Hypervisor on a Linux system, through QEMU/KVM.

3.1 Tools

As previously mentioned, the host operating system that is running on our machine is Linux. For our analysis we are targeting a component of the Windows operating system. For this reason, we will run Windows inside a virtual machine. In fact, we need two virtual machines: one for running the debugger and one for running the debugged system, which we will further refer to as the *debuggee*.

We used KVM (Kernel-based Virtual Machine) [3] as our virtualization solution. KVM is a virtualization module for Linux systems running on x86 hardware. It enables the kernel to function as a hypervisor with minimal need for hardware emulation. KVM interoperates with QEMU (Quick Emulator) for emulating devices. As such, virtual machines running under QEMU/KVM have in most scenarios, near-native performance. We also used virt-manager (Virtual Machine Manager) [4] as a graphical interface for managing the virtual machines. All three software components mentioned are free and open-source.

For debugging we used WinDbg, a debugger developed by Microsoft, which offers kernel-level debugging capabilities. This is a requirement for debugging Hyper-V.

3.2 Setup

We need two virtual machines running Windows: one for running the debugger and one which serve as the debuggee. The former does not need any special setup and can easily run with the default CPU settings. However, we will need to make sure that both VMs are on the same network. As such, we set both of the VMs to use the default `NAT` network, and the `e1000e` device model. With both machines running we disabled the Firewall on both of them. This just makes the whole testing process more convenient, as we can now ping one machine from another to test their connectivity. Once pinging was successful, we recorded the IP address of each machine and continued to the next step.

We will split the rest of the setup in two parts. We will first cover the setup of the machine running the debugger, and then the setup of the machine running debuggee.

3.3 Debugger setup

The setup of the debugger consists of installing the `windbg` debugger. This can be achieved by installing the *Debugging Tools for Windows* as part of the *Windows SDK* [9]. The required size for the install is more than 3GB, but the total installed size of the component we need is only around 500MB. For ease of use we created a shortcut to the `windbg.exe` executable in a convenient location which was easy to reference later.

3.4 Debuggee Setup

This is by far the more difficult setup out of the two. First of all we need to make the virtualization capabilities of the CPU available to the VM. Failing to setup the virtual CPU correctly will lead to a system which does not boot after enabling virtualization. Moreover, having snapshots of a working system makes reverting breaking changes very easy.

First, we ensured that nested virtualization is enabled. Running `cat /etc/modprobe.d/kvm_intel.conf` should print `options kvm-intel nested=Y` on an Intel system. A similar result should be expected on AMD [5].

CPU Configuration. Next, we need to alter the VM configuration. The default CPU configuration did not work in our case and lead to Windows not booting after enabling Hyper-V. Listing 1.3 shows the changes which produced a working machine in our case. These changes include suggestions from a number of different sources. Line 2 enforces the emulation of the *Skylake* architecture, without TSX enabled. This change will significantly impact performance, compared to passing through the native CPU, likely because of disabling TSX, as a source suggested [11], but also as a result of emulating a different architecture. Line 3 makes the CPU inside the VM appear as if it were a bare-metal hardware component. Line 4 enables the virtualization features inside the VM [15] [11] [13].

```
1  <cpu mode='custom' match='exact' check='partial'>
2    <model fallback='allow'>Skylake-Client-noTSX-IBRS</model>
3    <feature policy='disable' name='hypervisor'/>
4    <feature policy='require' name='vmx'/>
5  </cpu>
6  <features>
7  ...
8  <hyperv mode="custom">
9    <vpindex state="on"/>
10   <synic state="on"/>
11   <vendor_id state="on" value="KVMKVMKVM"/>
12 </hyperv>
13 </features>
```

Listing 1.3. Settings necessary for running Hyper-V inside a QEMU/KVM-based virtual machine.

Further CPU Configuration. After a number of experiments we performed on the CPU configuration, we achieved a smaller working configuration which was not featured in any of the online resources that mentioned setting-up Hyper-V under KVM [15] [11] [13]. The final configuration which we used can be inspected in Listing 1.4.

```
<features>
  <hyperv mode="custom">
    <vendor_id state="on" value="KVMKVMKVM"/>
  </hyperv>
</features>
<cpu mode="custom" match="exact" check="partial">
  <model fallback="allow">Skylake-Client-noTSX-IBRS</model>
  <feature policy="require" name="vmx"/>
</cpu>
```

Listing 1.4. Reduced CPU configuration.

With the CPU set up correctly and the machine running we enable Hyper-V by running `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All` in a privileged console and then follow the instructions and restart the machine. It should start-up without any issues.

Further, to continue the setup, we have two options. We can either do the setup automatically using a tool called `kdnet`, or do the setup manually. We will present both options.

Manual Setup. Manually setting up the debuggee involves enabling both kernel and hypervisor debugging and instructing the machine to connect to the debugger on startup. All relevant commands can be seen in Listing 1.5 [14].

```
bcdedit /hypervisorsettings net hostip:<debugger_ip> `
    port:50001 key:1.1.1.1
bcdedit /set hypervisordebug on
bcdedit /dbgsettings net hostip:<debugger_ip> `
    port:50002 key:1.1.1.2
bcdedit /debug on
bcdedit /set hypervisorlaunchtype auto
bcdedit /bootdebug on
```

Listing 1.5. Commands for the manual setup of the debuggee.

Kdnet Setup. Alternatively, having already installed the *Debugging Tools for Windows* on the debugger machine, we can follow the instructions on Microsoft's website [10]. Even though the tutorial only addresses kernel debugging, we can easily enable hypervisor debugging with the same tool, by specifying the `-h` flag: `kdnet.exe <debugger_ip> 50002 -h`.

We can inspect that everything is set up as expected by running one or more of the commands found in Listing 1.6

```
bcdedit
bcdedit /dbgsettings
bcdedit /hypervisorsettings
```

Listing 1.6. Optional commands which can be run to inspect the config.

3.5 Debugging

With the setup complete, we can start debugging Hyper-V. To do this, we need both the debugger and the debuggee running. On the debugger, we start two instances of `windbg`, one for the kernel and one for the hypervisor using the commands in Listing 1.7. After that, we restart the debuggee by running `shutdown -r -t 0`.

```
.\windbg.lnk -k net:port=50001,key=1.1.1.1
.\windbg.lnk -k net:port=50002,key=1.1.1.2
```

Listing 1.7. Commands to run for starting windbg and accepting connections from the debuggee to the kernel and hypervisor debuggers.

We further follow instructions from the intro blog to Hyper-V research [14], get the base address where the `hyperv` module was loaded and rebase the binary analysed in Ghidra at that address (`Display Memory Map` → `Set Image Base`), in order to bypass ASLR. We can then use a (rather hacky but very easy) trick in order to call and start debugging any hypercall. In the kernel window we set a breakpoint at `nt!HvcallInitiateHypercall`. When this breakpoint is hit, the kernel is about to issue a new hypercall. We can hijack this call, by modifying `rcx` and setting our desired hypercall id, as well as the necessary flags for setting our desired calling convention. We can then also override the input parameters, by following the calling convention specified in `rcx`. More details on call conventions and input parameters can be found in the TLFS [8].

A Useful Extension. `ret-sync` [6] is a very useful tool which aims to bridge the gap between dynamic analysis and static analysis. It achieves this by synchronising a debugging session (windbg in our case) with a disassembler (Ghidra in our case). We were unfortunately not able to use it, because of some missing windows dependencies for synchronising with `windbg`. Naturally, we assume this is the case because we were running Ghidra on Linux. In retrospect, it would be more advantageous to run Ghidra in the debugger VM, alongside `windbg`. Unfortunately, due to limited resources, our setup did not support this path of action.

4 Findings

We will discuss our findings about the undocumented hypercall number `0x78`, named `HvCallQueryNumaDistance`. In this context, it is important to mention that our system has only 1 `NUMA` node available [7].

Inspecting the corresponding entry in the hypercall table, we see that it is not a repeat call, and that it has 8 bytes of input and 8 bytes of output. We worked through the Ghidra decompilation to "beautify" it by retyping and renaming variables and function signatures. The result of the beautifying process can be seen in Listings 1.8, 1.9 and 1.10.

From the decompilation it becomes more apparent that the single input parameter is a pointer to two `uint` values, and the output parameter is a pointer to a single `INT_64` value. A plausible assumption would be that the two input integers represent IDs of two NUMA nodes and the output will hold the computed *NUMA distance* between these two nodes [7].

We notice that the execution path splits based on what appears to be the `AccessVpRunTimeReg` privilege flag [8]. When the caller does not have the previously mentioned privilege, execution will follow the first branch of the if statement as seen in Listing 1.8. Dynamic analysis shows that by default, when executing the `0x78` hypercall handler, the previously mentioned privilege flag is set and execution follows the second branch of the if statement. However, we can force the execution on the first branch by changing the `rip` register (instruction pointer). Further analysis showed that:

- If the input values are equal, a default value is returned, which in our case was equal to -1. This could suggest that querying for the distance of two equal nodes, does not make sense (Line 3).
- If the input values differ, the result is computed based on the default value and a threshold. We could not derrive any meaningful conclusions from these computations (Lines 6 - 10).

```
1  if ((*(byte *)(lRam000000ff000103a8 + 0x120) &
       AccessVpRunTimeReg) == 0) {
2    if (*input == input[1]) {
3      *output = DEFULT_VALUE;  // in our analysis -1
4    }
5    else {
6      uVar2 = (ulonglong)(DEFULT_VALUE * 29) >> 4;
7      if (uVar2 < THRESHOLD) {
8        uVar2 = THRESHOLD;
9      }
10     *output = uVar2;
11   }
12 }
13 else {
14     IVar1 = get_numa_distance(input[0],input[1]);
15   *output = IVar1;
16 }
17 return HV_STATUS_SUCCESS;
```

Listing 1.8. Decompilartion of the `HvCallQueryNumaDistance` hypercall handler, after being "beautified" in Gidra.

As previously mentioned, the `AccessVpRunTimeReg` appears to be set, so the default execution follows the second branch of the if statement, where the `get_numa_distance` function is called on the two input values. The Ghidra decompilation can be seen in Listing 1.9. The two calls to the function `numa_find` determine if the second parameter is found in a global array, which we renamed `NUMA_ARRAY` in Listing 1.10. In that case, the index is returned through the first parameter. Dynamic analysis suggested in our case that the array size is 1, which would correspond to our system having only 1 NUMA node. Looking at the function call graph, we noticed a code section where this array is extended with new values. As such, we intuitively assume that the referenced array holds the IDs of registered NUMA nodes since system start-up. Since the order they are registered can supposedly be arbitrary, the `NUMA_DISTANCE_TABLE` which holds the NUMA distance between the two nodes is indexed by the index and not the ID of the respective nodes.

```
find_r1 = numa_find(&index_1,param_1);
if ((find_r1 == HV_STATUS_SUCCESS) &&
    (find_r2 = numa_find(&index_2, param_2,
     find_r2 == HV_STATUS_SUCCESS))) {
  return NUMA_DISTANCE_TABLE[index_1][index_2];
}
return -1;
```

Listing 1.9. Main part of the `get_numa_distance` function decompilation from Ghidra.

Since there was only one NUMA node available on our system, the only valid query was the id pair $(0, 0)$. The result of the query is -1, further reinforcing the theory which suggests that the NUMA distance between a node and itself is not defined. If we query other values, the result is the same, as a result of invalid parameters.

```
idx = 0;
if (NUMA_ARRAY_SIZE != 0) {
  iter = &NUMA_ARRAY;
  do {
    if (*iter == target) {
      *idx_found = idx;
      return HV_STATUS_SUCCESS;
    }
    idx = idx + 1;
    iter = iter + 1;
  } while (idx < NUMA_ARRAY_SIZE);
}
return HV_STATUS_INVALID_PARAMETER;
```

Listing 1.10. Main part of the `numa_find` function decompilation from Ghidra.

5 Conclusions

In this report, we covered approaches to Hyper-V research, focusing on reverse engineering hypercalls. We considered the interesting scenario of doing both static analysis and dynamic analysis on Linux. Firstly, we discussed our static analysis setup, focusing on some aspects of Ghidra, a powerful reverse engineering framework. We showed how the whole setup for reverse engineering Hyper-V, in particular the `hvix64.exe` windows binary, can and probably should be automated through the powerful scripting API of Ghidra. Next, we covered our dynamic analysis setup. We presented our approach to debugging the windows kernel and the windows hypervisor on a Linux host, by running two VMs on top of QEMU/KVM. We discussed in detail the setup of the debuggee machine, which needed support for nested virtualization, because of which, also a special CPU setup. Lastly, we showed the setup in action by reverse engineering the hypercall handler routine number `0x78`.

Our work will hopefully serve as a basis for future attempts at reverse engineering Hyper-V, potentially on Linux. Since Hyper-V is a closed-source platform, efforts such as this one should benefit future researchers and contribute to securing this popular virtualization platform, by uncovering bugs or by providing further insights into its undocumented compenents.

References

1. Ghidra (Mar 2024), https://github.com/NationalSecurityAgency/ghidra, [Online; accessed 19. Mar. 2024]
2. Ghidra License (Mar 2024), https://github.com/NationalSecurityAgency/ghidra?tab=Apache-2.0-1-ov-file#readme, [Online; accessed 19. Mar. 2024]
3. KVM (Mar 2024), https://linux-kvm.org/page/Main_Page, [Online; accessed 20. Mar. 2024]
4. Virtual Machine Manager (Jan 2024), https://virt-manager.org, [Online; accessed 20. Mar. 2024]
5. Nested Guests - KVM (Oct 2023), https://www.linux-kvm.org/page/Nested_Guests, [Online; accessed 20. Mar. 2024]
6. ret-sync (Mar 2024), https://github.com/bootleg/ret-sync?tab=readme-ov-file#ghidra-usage, [Online; accessed 21. Mar. 2024]
7. Hyper-V Virtual NUMA Overview (Oct 2016), https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/dn282282(v=ws.11), [Online; accessed 23. Mar. 2024]
8. alexgrest: Hypervisor Specification (May 2022), https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/tlfs, [Online; accessed 21. Mar. 2024]
9. Domars: Debugging Tools for Windows - Windows drivers (Dec 2023), https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools, [Online; accessed 21. Mar. 2024]
10. Domars: Set Up KDNET Network Kernel Debugging Automatically - Windows drivers (Aug 2023), https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection-automatically, [Online; accessed 21. Mar. 2024]
11. Linpro, R.: Nested Virtualization - Hyper-V 2019 in qemu-kvm (Apr 2021), https://www.redpill-linpro.com/techblog/2021/04/07/nested-virtualization-hyper-v-in-qemu-kvm.html, [Online; accessed 20. Mar. 2024]
12. Mediant: Ghidrathon (Mar 2024), https://github.com/mandiant/Ghidrathon, [Online; accessed 20. Mar. 2024]
13. PeterGV: Using WinDbg Over KDNet on QEMU-KVM (Oct 2021), https://www.osr.com/blog/2021/10/05/using-windbg-over-kdnet-on-qemu-kvm, [Online; accessed 20. Mar. 2024]
14. siwat: First Steps in Hyper-V Research | MSRC Blog | Microsoft Security Response Center (Mar 2024), https://msrc.microsoft.com/blog/2018/12/first-steps-in-hyper-v-research, [Online; accessed 19. Mar. 2024]
15. Strube, R.: Kvm nested Virtualbox windows guest (Sep 2020), https://superuser.com/a/1589286, [Online; accessed 20. Mar. 2024]