# *Sarracenia*: Enhancing the Performance and Stealthiness of SSH Honeypots using Virtual Machine Introspection

Stewart Sentanoe✉, Benjamin Taubmann, and Hans P. Reiser

University of Passau, Germany
{se,bt,hr}@sec.uni-passau.de

**Abstract.** Secure Shell (SSH) is a preferred target for attacks, as it is frequently used with password-based authentication, and weak passwords can be easily exploited using brute-force attacks. To learn more about adversaries, we can use a honeypot that provides information about attack and exploitation methods. The problem of current honeypot implementations is that attackers can easily detect that they are interacting with a honeypot and stop their activities immediately. Moreover, there is no freely available high-interaction SSH honeypot that provides in-depth tracing of attacks.
In this paper, we introduce *Sarracenia*, a virtual high-interaction SSH honeypot which improves the stealthiness of monitoring by using virtual machine introspection (VMI) based tracing. We discuss the design of the system and how to extract valuable information such as user credential, executed commands, and file changes.

**Keywords:** Honeypot, Virtual Machine Introspection, Secure Shell

## 1 Introduction

A Honeypot is a system that aims at gathering knowledge about attacks by luring the adversaries to attack it [17, 13]. One challenge of honeypots is to ensure **stealthy and reliable extraction** of useful information that is not directly noticeable to an adversary in order to learn more about **honeypot-aware attacks**. This means that an adversary first checks if he is attacking a real system and only runs the full attack when he is sure not to be connected to a honeypot [25, 19, 38].

There are two kinds of honeypots that can gather in-depth traces of an attack: the high-interaction and the medium- interaction honeypots. A high-interaction honeypots monitor a real system either by installing a Man-in-The-Middle proxy that captures the SSH session [30, 35], or by tracing the execution using an in-guest agent, such as a kernel modules. A proxy-based approach provides a high level of stealthiness, however, it lacks the ability to reconstruct the full attack e.g., when additional binaries are downloaded and deleted directly after the execution, or when additional encrypted communication channels besides SSH

Author's version of the paper acceted for NordSec 2018, published by Springer in LNCS 11252.
(C) Springer Nature Switzerland AG 2018
The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-03638-6_16`.

are used which can not be decrypted by an SSH proxy. In-guest agents can provide in-depth tracing of the attack, but they can be detected, or disabled. To the best of our knowledge, there is no in-guest agent based SSH high-interaction honeypot.

Emulation is another commonly used approach for medium, or low-interaction honeypot. Emulation means that the honeypot provides a service that is similar to the expected one. Cowrie [23] is the most commonly used SSH honeypot and emulates the behavior of an SSH server and Debian operating system, which means that a user can log in to a system but all commands are emulated, i.e., only log the execution of the command but do not really have any other functionality. Because of this, it is pretty easy for an attacker to detect whether a system is an emulated honeypot, or a real system.

To sum up, the problem with current SSH honeypot solutions is that they are either easy to detect, or do not provide in-depth tracing to reconstruct full attacks.

Virtual Machine Introspection (VMI) is the process of examining and monitoring a virtual machine (VM) from the outside, i.e., the virtual machine monitor (VMM) point of view. VMI has proven to be effective in monitoring activities of a VM without the presence of an in-guest agent either by using system call tracing [24, 10, 7], or memory-based introspection [29, 8, 32, 11]. In the past, it has been shown that, compared to in-guest agents, VMI has several advantages in intrusion detection systems (IDS) [10] and monitoring of virtual honeypots [22, 15, 27].

This paper introduces *Sarracenia*, a **virtual high-interaction SSH honeypot** based on VMI which combine system call and user-space function tracing. This approach produces less overhead than tracing just system calls to extract information. The contributions of this paper are:

- The design and implementation of a VMI-based high-interaction SSH honeypot architecture that provides in-depth traces of attacks including executed commands, session replay, a list of manipulated and downloaded files including their content and the traffic of forwarded connections.
- Tracing mechanism that can be used to build another honeypot, malware tracing, or IDS system.
- To tackle the problem of honeypot-aware attacks, we employ well-known VMI-based tracing techniques of libvmi [24] and Drakvuf [21] in order to achieve a better level of stealthiness and to trace the execution of user-space function calls.
- The performance evaluation that measures the overhead added by VMI-based tracing.

Our performance evaluation shows that *Sarracenia* can match the performance of a normal SSH server with a small increase in execution time (approximately 0.01 $s$) when used to trace simple activities.

When we monitor the file system changes, the execution time of the honeypot increases by at least 0.08 $s$ based on how many files are generated and extracted.

| | Concept | Interaction | Shell commands | Stealthiness | File Change Detection | sftp and scp | Session Reconstruction | Playable Log | Port Forwarding | Extracted information |
|---|---|---|---|---|---|---|---|---|---|---|
| **Kojoney [6]** | Emulation | ○ | ◔ | ○ | ◔ | ○ | ◔ | ○ | ○ | ◔ |
| **Kippo [31]** | Emulation | ◑ | ◑ | ○ | ◑ | ○ | ● | ● | ○ | ◑ |
| **Cowrie [23]** | Emulation | ◑ | ◕ | ○ | ◕ | ● | ● | ● | ◑ | ◑ |
| **SSHHiPot [30]** | MiTM | ● | ● | ● | ○ | ● | ○ | ○ | ● | ◑ |
| **ssh-mitm [35]** | MiTM | ● | ● | ● | ○ | ● | ● | ○ | ● | ◑ |
| *Sarracenia* | VMI-Based | ● | ● | ● | ● | ● | ● | ● | ● | ● |

**Table 1.** Comparison of SSH honeypots, a full circle represents for is supported or high performance and an empty one for the opposite.

## 2 Related Work

This section reviews different approaches and related work concerning SSH honeypots and their analysis as well as tracing techniques.

### 2.1 SSH Honeypots

The subsequent sections discuss different approaches for SSH honeypots. Table 1 summarizes the differences between them and *Sarracenia* and also shows which features are supported by each approach.

**Low-Interaction** *Kojoney* [6] is a low-interaction honeypot (developed by Jose Antonio Coret) that emulates an SSH server. Emulation means that it imitates the behavior of a real SSH service which means that not all functionalities of the real SSH are available. *Kojoney* logs the username and password combination, executed commands and terminal window size. The advantage of emulation is that it can run mostly isolated (the honeypot process can run with restricted user permissions) and lessens the chance that the adversaries can take over the entire host. The disadvantage of this approach is that only a limited number of shell commands are available, which means that scripts of adversaries might fail or adversaries would leave the system without doing any further malicious activities.

**Medium-Interaction** *Cowrie* [23] (former: *Kippo* [31]) is a medium-interaction honeypot which also emulates an SSH server just like low-interaction but it adds fake Debian system which also emulated. The drawback is that not all the commands work the same way as in a real system because each command is re-implemented and does not provide full functionality. The file changes detection and extraction in *Cowrie* supports only selected commands such as *wget*, *curl* and *sftp/scp*. It also supports port forwarding, but instead of forwarding it to the real destination, it needs to be configured where to forward the data to e.g., forwarding SMTP connection to an SMTP honeypot.

**High-Interaction** *SSHHiPot* [30] and *sshmitm* [35] are a high-interaction SSH honeypot that implement the concept of an active Man-in-The-Middle (MiTM) proxy between the adversary and the SSH server.

The advantage of a MiTM approach is that all shell commands can be used by the adversaries, so more information about the adversaries' activities that happen inside the VM (high-interaction) can be extracted and since no agent inside is involved, it is difficult for the adversary to detect the monitoring.

The disadvantage of this approach is that it fails at detecting changes to the file system, i.e., it cannot restore files that have been transferred via an additional encrypted network communication channel such as *https*. Thus, it might not be possible to analyze the actual malware sample of an attack.

High-interaction honeypot suffers the problem of finding a compromise between restraining the network access to the other systems in order to protect and not contributing to further attacks and analyzing the full behavior of ongoing attacks. The goal of *Honeywall* [9, 28] is to control the network usage of the successful attacks [4] by acting as a network bridge gateway of all honeypots where the network activities are logged and *iptables* is used to apply network rules that limit the network access.

There are some researches that focus on how to detect the presence of a honeypot, or introspection system called anti honeypot [37] and introspection detection [36]. For high-interaction honeypots, there are two main methods which are: system level fingerprinting and operational analysis. One way to do system level fingerprinting is by timing benchmark [12] which calculates the execution time of commands. If the time is longer than on a sane system, it means that a monitoring, or introspection system might be present. Operational analysis can be done by executing several commands and compare the generated output of the remote server with a sane system [38].

### 2.2 VMI-based honeypots and tracing method

**VMI-Based Tracing** A VMI-based IDS [10] introduced by Garfinkel et al. They added hooks that analyzed and observed VM CPU, memory, and emulated devices in order to reconstruct the VM state. Using this same approach, Jiang et al. [16] developed *VMwatcher* that is able to implement hooks into several hypervisors to extract the memory and file system of a VM. The acquired data is exported to a separate VM where the memory is compared to a clean-state template and the file system is scanned by an anti-virus software.

Lengyel et al. [22] implemented a hybrid honeypot architecture that combines a low-interaction honeypot to collect malware and a high-interaction honeypot (sandbox) to analyze the captured malware. By monitoring the sandbox VM using VMI, they were able to record all activities of the malware. To detect anomalies, their system compared the result that obtained by the VMI against the clean original state of the VM. *Sarracenia* uses the similar approach, but we provide more API to do user-space function tracing that can be used to build another honeypot, or an IDS system.

We made a preliminary research on VMI based SSH honeypot [27]. We traced *write* and *read* system call to extract the username and the password. The method was effective, but not efficient since the overhead was pretty high and pattern matching also needs to be done.

**Kernel Module Monitoring** Block and Dewald [2] described how to monitor and extract information from the heap memory. They built plug-ins for Rekall [5] that are able to extract command history from *zsh* and password entry information from a password manager called *KeePassX*.

**Taint Analysis** Portokalidis et al. [26] built a honeypot system based on taint analysis using QEMU. It works by tagging the data that comes from an unsafe source and track the activities of the data. When a violation is detected, an alarm is raised and deeper inspection is made. Portokalidis et al. [25] introduced Eudaemon, a technique that analyzes a running process in an emulator which provides extensive instrumentation in the form of taint analysis. Bosman et al. [3] introduced Minemu, a fast *x86* taint tracker that address the problem of dynamic taint analysis which is high overhead.

## 3 System Architecture and Design

This section discusses the goals of *Sarracenia*, its architecture and components.

### 3.1 Goal

The goal of *Sarracenia* is to provide a virtual high-interaction honeypot that aims at attracting adversaries that would normally leave a honeypot when they detect that it is not a real system, in order to understand new attacks. Thus, stealthy and reliable monitoring is required in order to reconstruct an attack as accurately as possible. To achieve that, we capture all modifications and interactions of an adversary with the system under analysis. *Sarracenia* traces these actions of an adversary:

1. Entered and executed commands in order to **replay the SSH session**.
2. **Rebuild files** that have been transferred via *scp/sftp*.
3. **File system changes** to extract malware samples that have been loaded over encrypted channels such as *https*.
4. **Monitor port forwarding** of the SSH server where the destination address and the payload are extracted.

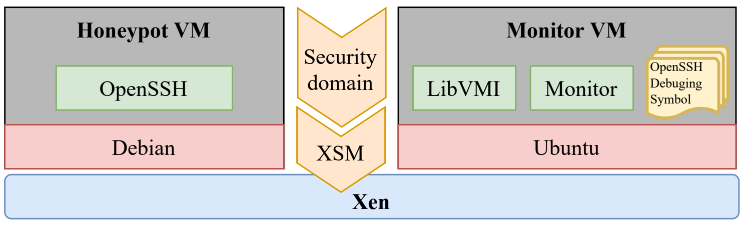

**Fig. 1.** *Sarracenia*'s component.

### 3.2 Components

The *Sarracenia* architecture uses two types of virtual machines. One virtual machine is the honeypot and the other one monitors the honeypot as shown in Figure 1.

We downloaded the OpenSSH's debugging symbol that match the honeypot's version (OpenSSH_7.2p2 Ubuntu-4ubuntu2.4, OpenSSL 1.0.2g 1 Mar 2016). We were able to extract the debugging information using libdwarfparser [18] and libelf in order to get the symbols and the layout of the required data structures. To avoid tracing problems coming from concurrency, the honeypot has only one CPU. The monitoring VM is a normal Debian installation with tracing tools installed. Both virtual machines are running on top of Xen.

The VMI access required from one VM to another VM is granted by using policies of the Xen security modules (XSM). It controls the access of Xen domains, hypervisor, and resources including memory and devices. We implemented policies so that a monitoring VM can access the memory of a honeypot but not vice versa. This concept is described by the CloudPhylactor [33] architecture.

### 3.3 Tracing Methods

In order to trace the control flow during an attack, we use VMI-based tracing. The main challenges of *Sarracenia* are: bridging the semantic gap e.g., to trace the user-space SSH daemon then extract the required information and low overhead e.g., the monitoring mechanism not impacting the performance of the honeypot which could be detected by an adversary. For low-level VMI functionality, we use LibVMI in conjunction with a self-written library [34] that simplifies the insertion and processing of software breakpoints.

The performance impact of tracing is mainly caused by context switches between the monitoring VM and the honeypot VM whenever information needs to be extracted, e.g., the user credentials during the authentication process. Thus, one goal of *Sarracenia* is to identify the best place in the control flow to extract the required data to minimize VM context switches.

The challenges associated with the semantic gap differ in these scenarios. In the case of monitoring only the read system call, we need to determine the call which, e.g., reads the password from the remote console and differentiate it from the rest of the read calls. However, by monitoring the validation function we have to know the symbol name of the function and where it is located in memory so that we can intercept it. Additionally, we need to know the parameters and also the layout of the data structures which can be extracted easily from the debugging information of a binary file.

To intercept the control flow of the honeypot we use software breakpoints there are two ways to do it:

**Pure INT3:** It replaces the original opcode at the beginning of a function with the *INT3* instruction, which causes an interrupt that is handled by the monitor. *INT3* approach is simpler but suffers an important problem which is the race condition when multiple vCPUs are used.

**Xen altp2m [20]:** It creates two additional memory after the guest's physical memory which contains shadow copy of the target page with the trap and empty page. Then, it sets an access permission for the shadow page. Whenever there is access violation e.g., execute attempt, it can simply change the pointer back to the original page, single step and change the pointer back to the shadow copy. When an adversary tries to read or write the shadow copy, it will change the pointer to the empty page. Thus, it conceals the breakpoint well. The advantage of this approach is it works nicely with multiple vCPUs. This approach also used by Drakvuf [21] which we used it for *Sarracenia* since it provides straightforward API to attach a breakpoint. But, it turns out that we able to detect the presence of Drakvuf by using *ioremap* function where we probe the memory beyond the physical range. Drakvuf's implementation problems and the fixes are:
  - The empty page consists of 00 (zero) where the real behavior according to Intel Documentation [14] is that attempt to read the invalid memory address (e.g., outside the physical range) will return all 1s (FF). We fixed this issue by replacing the 00 with FF during the empty page initialization process.
  - The empty page is not protected by access control and write attempt will be persistent. But, based on Intel documentation, write attempt of invalid memory address will be ignored. We fixed this issue by add access permission of the empty page and the shadow page. When write is executed, the system notifies Xen to emulate the writing process and return the emulation result. Thus, the value never get written to the memory.

*Sarracenia* implements two modes of operation for the tracing with different overhead:

**Process-bound:** Breakpoints on system calls are attached and detached dynamically based on the process that is running. To do this, we monitor write access to the *CR3* register that holds the addresses of the page directory base (PDB) — the data structure which is used by the memory management unit for address translation — which is different for each process with

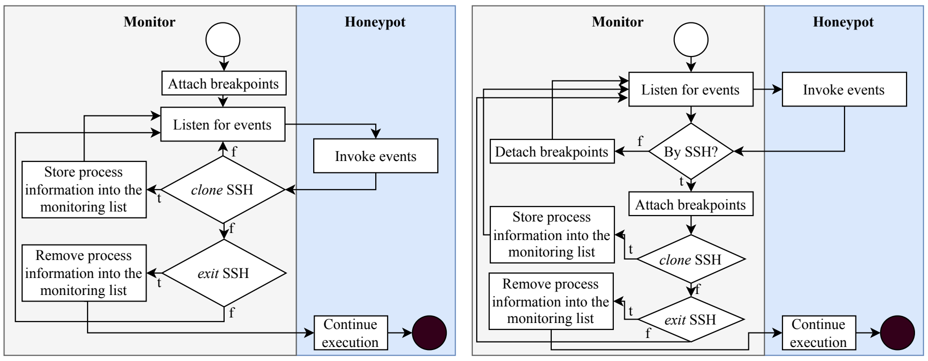

**Fig. 2.** Control flow for new SSH connections (left: system-wide right: process-bound).

LibVMI. Whenever a new process is dispatched, the content of this register is updated with the PDB of the next process. Thus, we can control that a specific process is monitored. This requires a VM context switch at every process change in order to check whether breakpoints should be set or not. Additionally, the breakpoints must be written to memory or removed with the original instruction if the new process should be monitored or not.

**System-wide:** All breakpoints are set from the beginning when the monitoring is started which means that all processes are traced. This does not require a context switch for every process change. However, it results in more context switches for system calls at run-time.

## 4 Data Acquisition

In order to analyze the activities of an adversary, we can use different levels of tracing with various amounts of information. In general, *Sarracenia* aims at capturing the same information as Cowrie, which is: (1) Detection of new SSH connections, (2) extraction of user credentials, source IP address and port, session keys of an authentication process, (3) reconstruction of SSH sessions, e.g., entered commands, (4) data of TCP port forwarding and (5) modification of file system changes. Table 2 shows the traced function and system call for each information extraction. In the subsequent sections, we describe how we extract this data in detail.

### 4.1 New SSH Connection

In order to detect new connections to the honeypot, we monitor the *clone* system call as shown in Figure 2. OpenSSH invokes *clone* to create a child process that handles each SSH session. When the session terminates, *sys_exit_group* is invoked.

| | Name | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| **System Call** | clone | ✓ | | | | |
| | sys_exit_group | ✓ | | | | |
| | exec | | | ✓ | | |
| | write | | | | | ✓ |
| | seek | | | | | ✓ |
| | close | | | | | ✓ |
| **Function** | kex_derive_keys | | ✓ | | | |
| | auth_password | | ✓ | | | |
| | sshbuf_get_u8 | | | ✓ | ✓ | |
| | ssh_packet_send2_wrapped | | | ✓ | ✓ | |
| | channel_connect_to_port | | | | ✓ | |

**Table 2.** Function and system calls that are traced for (1) Detection of new SSH connections, (2) extraction of user credentials, source IP address and port, session keys of an authentication process, (3) reconstruction of SSH session, i.e., entered commands, (4) data of TCP port forwarding, and (5) modification of file system

### 4.2 SSH Key Derivation, Source IP Address and Port Monitor

At the beginning of each SSH session, the key material is negotiated. To extract the SSH session keys, source IP address and port number, two OpenSSH functions are traced: *kex_derive_keys* and *do_authentication2*. From the input parameter of *kex_derive_keys*, the hash $h$ and the shared secret $K$ can be extracted. In this step, the memory address of the *ssh* struct which used to store the session keys is stored. When *do_authentication2* is called, the input parameter of this function holds information about the IP and port (remote and local) where the remote IP can be collected to get the overview of the adversary's location. At this state, the authentication method is about to begin which means that the session keys are already derived and can be extracted by accessing the memory address of the *ssh* struct that was stored before.

### 4.3 Authentication Phase Monitor

The username and password of an authentication attempt can be extracted from the input parameters of the function *auth_password*. To accept multiple passwords for a username, we modify the return value of the function by setting a breakpoint to the instruction where the function returns to. Instead of returning 1 for the correct password and 0 for the incorrect password, we inject 1 to make all passwords that are typed by the adversary to be accepted as long as the user exists in the VM.

### 4.4 SSH Packet Monitor

Each SSH packet is encrypted during transmission via the network. In order to extract the content of an SSH packet, we monitor two functions: *ssh_packet_send2_wrapped* and *sshbuf_get_u8* which responsible for the encryption and decryption process of the SSH network packet.

### 4.5 SSH Session Monitor (Keystrokes)

In order to reconstruct an SSH session, we monitor the function *ssh_packet_send2_wrapped* and extract the data section of the packet that contains the keystroke. The keystrokes are stored in a JSON formatted file, which can be replayed using asciinema.

### 4.6 Executed Command

To obtain the executed commands, the *exec* system call is traced. By tracing *exec*, we are able to get an overview of which commands are executed during an attack. Additionally, this is required to trace commands that are executed inline, i.e., that are not executed in an SSH bash and thus are not recorded by the SSH session monitor.

### 4.7 Port Forwarding

To extract the network packets which are forwarded by the SSH daemon, *Sarracenia* monitors the *channel_connect_to_port* function. The target IP and port can be extracted from the function parameters. The payload itself can be extracted from the SSH packet that explained in Section 4.4.

### 4.8 Changes on File System

In many cases, an adversary downloads additional malicious code from external sources. These files can be important when it comes to analyzing the attack. As there are several ways to download data (*sftp*, *wget*, *curl*, ...) *Sarracenia* uses the general approach of monitoring changes to the file system which works with different applications. Since adversaries might delete the file directly after executing it, it might not be possible to analyze the disk image after the attack. Thus, *Sarracenia* monitors changes to the file system and writes them into separate files.

To achieve that, *Sarracenia* monitors *write, seek* and *close* system calls. By keeping track of the file descriptor in the process namespace, whenever *write* is invoked by the same process, the data that is about to be written to that particular file can be extracted. When *close* is invoked by that process, we stop the tracking of a particular file descriptor.

Since monitoring this three systems calls is expensive as they are used by many processes, we evaluate in Section 5 different approaches to minimize the impact by using dynamic tracing, e.g., only monitoring these system calls for a small set of processes.

## 5 Evaluation and Discussion

The performance of the monitoring is an important aspect for a honeypot since it can lower the stealthiness. Thus, this section discusses the performance of *Sarracenia*.

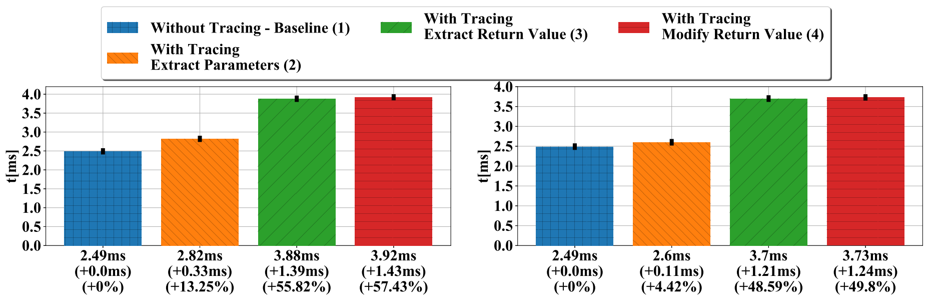

**Fig. 3.** Function tracing overhead (*auth_password*): (1) without tracing, (2) with tracing - only extracting function parameters, (3) with tracing - setting a second breakpoint to the end of the function to extract the return value and (4) with tracing - setting a second breakpoint and modified the return value. Left: Using INT3 and right: using altp2m.

## 5.1 Performance Analysis

**Function Tracing Overhead** To quantify the performance impact of VMI based control flow interception we measured the overhead of tracing one function call. Therefore, we called the function *auth_password* for 100 times and calculated the average runtime without (1) and with tracing (2-4). We distinguish between three different tracing variants: (2) extracting function parameters at the beginning of the function, (3) setting the second breakpoint to the end of the function to extract the return value, and (4) modifying the return value by writing to the *RAX* register. The results of the measurements are depicted in Figure 3.

Without any monitoring, it took 2.49 *ms*. When pure *INT3* is used, it took 2.82 *ms* for (2), which is an increase by 0.33 *ms*. When we add another breakpoint to intercept the return value (to accept all given password), it took 3.88 *ms* (increased by 1.39 *ms*) and 3.92 *ms* (increased by 1.43 *ms*) for (4). The measurements show that tracing increases the runtime of a process with small amount of overhead. But, it is still important to intercept the control flow as little as possible to minimize the overhead.

When altp2m is used, it took 2.60 *ms*, 3.70 *ms* and 3.73 *ms* which is an increase by 0.11 *ms*, 1.21 *ms*, and 1.24 *ms* for (2), (3), and (4) respectively.

**System Performance** To measure the performance of *Sarracenia*'s monitoring approach on the honeypot, we ran four **use cases**. For each use case, we used *time* command and used the *real* time:

A. **Simple command:** Execute `ls -alh` command.
B. **I/O intensive test:** Download a file with 2MB size using `wget` command.
C. **I/O and CPU intensive test:** Compile the Jansson library.

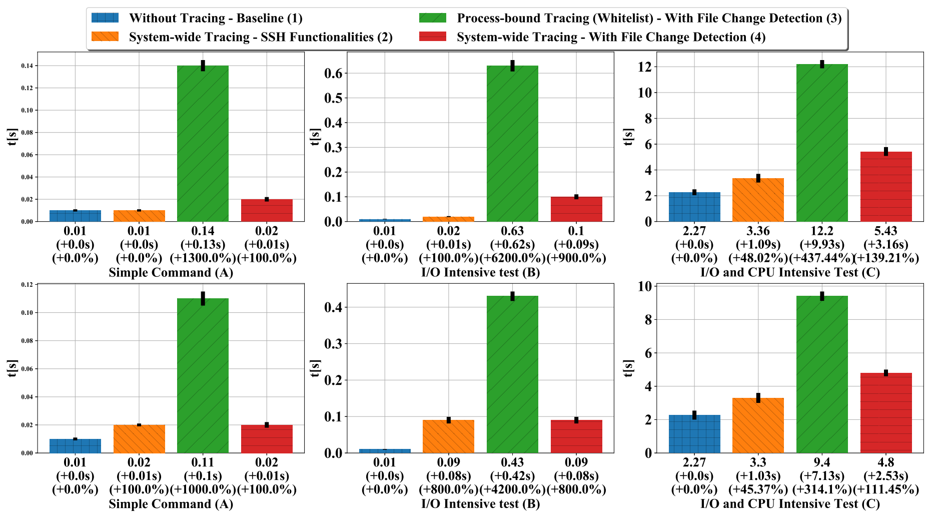

**Fig. 4.** Overhead of client's execution time based on different scenario and configuration where (1) is without monitoring as the baseline and (2) to (4) are monitored by *Sarracenia* using pure INT3 (first row) and altp2m (second row).

To measure the impact on the performance of the honeypot for each **tracing mechanism**, we run the four use cases with different configurations:

① **Without tracing:** The baseline to calculate the overhead of the tracing mechanism.
② **System-wide tracing - SSH functionalities:** *sys_clone*, *sys_exit_group*, and all OpenSSH functions are monitored.
③ **Process-bound tracing (whitelist) - with file change detection:** all system calls of OpenSSH functions are monitored. And, *wget, curl, sftp* and *scp* are monitored for file system changes.
④ **System-wide tracing - with file change detection:** all system calls and OpenSSH functions (see Table 2) are monitored.

We ran every combination of use-case and tracing mechanism 100 times. The summary of time measurement is depicted in Figure 4. The overhead is between 0.01 $s$ and 9.93 $s$. For activity (A), the overhead of all approaches was relatively small (min: 0 $s$ max: 0.13 $s$). For activity (B), the overhead start to be varied but still can be considered small since it started to do data extraction from the memory (min: 0.01 $s$ max: 0.62 $s$). For activity (C), the overhead started to be high since more kinds of stuff are happening in the honeypot and more data are extracted (min: 1.03 $s$ max: 9.93 $s$).

Process-bound suffered the highest overhead due to the attachment and detachment of a breakpoint process that increased the overhead which also depends on how many processes that causes interrupt. To attach a breakpoint using pure

*INT3* took 0.19 *ms* and 0.40 *ms* when altp2m is used. To detach a breakpoint it took 0.15 *ms* and 0.11 *ms* when pure *INT3* and altp2m is used respectively. But, during runtime, altp2m approach can simply switch the page table pointer instead of removing the *INT3* opcode.

Based on the overall result, we recommend using altp2m and system-wide tracing with file change detection approach for real-life deployment.

### 5.2 Stealthiness

As explained in Section 2.1, there are at least three approaches to detect whether a system is a honeypot or being monitored: operational analysis (execute arbitrary commands), system level fingerprinting (timing benchmark), detecting monitoring agent. We tested these approaches against *Sarracenia* and *Cowrie*. Since *Sarracenia* provides a fully fledged Linux system an attacker can execute and install any required tools. Thus, he can not see any difference to a normal system in contrast to Cowrie which only provides some limited amount of commands.

As discussed in Section 5.1 the overhead added to a single monitored function call can be between 0.11 *ms* and 1.43 *ms* which can be used to detect the presence of the introspection. This becomes noticeable when the same (monitored) system call is invoked multiple times and the timing of an untraced system is known. Reducing the impact of VMI-based tracing mechanisms is an ongoing research topic and reducing it to improve the stealthiness must be addressed in future work. Nevertheless, to the best of our knowledge when virtualization, e.g., in cloud computing, is used it is common that functions are delayed since several virtual machines share the same resources.

*Sarracenia* does not require any agent inside the honeypot. Thus, an adversary is not able to directly detect any monitoring component.

### 5.3 Portability

Since we have to bridge the semantic gap and interpret the contents of memory from the honeypot, it is important to discuss the portability of this approach, i.e., whether the approach will work in newer versions of a Linux system or SSH service. *Sarracenia* relies on the information we get from the *System.map* (function symbols of system calls) and the debugging information of the SSH daemon. Since the honeypot virtual machine is under our control, both information can be easily accessed or generated when the system is upgraded. *Sarracenia* is able to run on a standard Xen installation where Intel hardware virtualization is required.

### 5.4 Limitations

*Sarracenia* aims at extracting information from a virtual machine with VMI. Thus, it is vulnerable to adversaries that produce a great number of outputs, e.g., write a lot of data to files which are logged. This problem could be addressed for examples by a maximum log size for each adversary. We also do not cover attacks that target that SSH service, e.g., with buffer overflows. Then, we assume that adversaries do not do the timing based measurement. In the future, we need to discuss whether the timing behavior of intercepted functions can be used to detect VMI based monitoring especially in cloud environments where several virtual machines coexist on the same physical server. Lastly, we do not address the problem of attackers that revisit our honeypot and detect that it has been reset after a long period of time. This is a general problem of honeypots and is out of the scope of this paper. Finally, we do not consider attacks that actively put crafted data to main memory that subvert VMI based memory analysis [1].

## 6 Conclusion

In this paper, we presented *Sarracenia*, a VMI-based virtual high-interaction SSH honeypot. We explained the architectural design of it and compared it against several state-of-the-art approaches such as SSH emulation, Man-in-the-Middle, and custom SSH implementation. *Sarracenia*'s mechanism can be used to build another honeypot, malware tracing, and Intrusion Detection System.

Compared with other SSH honeypots, *Sarracenia* improves the stealthiness of the monitoring by applying VMI-based tracing and by providing a fully-fledged Linux system to an attacker. *Sarracenia* is able to extract useful information such as user's credentials, keystrokes, executed commands and changes on the file system including files that transferred over encrypted network channels and have been deleted after the execution.

*Sarracenia*'s performance varies depending on which tracing modules are enabled. Since one approach to detecting the presence of an analysis tools, is to check the timing behavior of a system, the stealthiness of VMI based tracing depends on the implementation of the interception mechanism, e.g., the breakpoints. Thus, minimizing the performance impact of each single breakpoints is an important objective of future VMI research.

To assess the effectiveness level of *Sarracenia*, long-term deployment and analysis of *Sarracenia* and other SSH honeypots are needed and it is the future work of this research.

## Acknowledgment

This work has been supported by the German Federal Ministry of Education and Research (BMBF) in the project DINGFEST-EFoVirt and German Research Foundation (DFG) in the project ARADIA.

## References

1. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., Xu, D.: Dksm: Subverting virtual machine introspection for fun and profit. In: 2010

29th IEEE Symposium on Reliable Distributed Systems. pp. 82–91 (Oct 2010). https://doi.org/10.1109/SRDS.2010.39
2. Block, F., Dewald, A.: Linux memory forensics: Dissecting the user space process heap. Digital Investigation **22**, S66–S75 (2017)
3. Bosman, E., Slowinska, A., Bos, H.: Minemu: The world's fastest taint tracker. In: International Workshop on Recent Advances in Intrusion Detection. pp. 1–20. Springer (2011)
4. Briffaut, J., Lalande, J.F., Toinard, C.: Security and results of a large-scale high-interaction honeypot. JCP **4**(5), 395–404 (2009)
5. Cohen, M.: Rekall memory forensics framework. DFIR Prague (2014), `https://digital-forensics.sans.org/summit-archives/dfirprague14/Rekall_Memory_Forensics_Michael_Cohen.pdf`
6. Coret, J.A.: Kojoney - A Honeypot For The SSH Service. `http://kojoney.sourceforge.net/` (2006), [accessed 2018-02-17]
7. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security. pp. 51–62. ACM (2008)
8. Dolan-Gavitt, B., Payne, B., Lee, W.: Leveraging forensic tools for virtual machine introspection. Tech. Rep. GT-CS-11-05, Georgia Institute of Technology (2011)
9. Enemy, K.Y.: Honeywall CDROM Roo 3rd Generation Technology. Honeynet Project & Research Alliance,[Online] Available: `https://projects.honeynet.org/honeywall/` **17** (2005)
10. Garfinkel, T., Rosenblum, M., et al.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed Systems Security Symposium (NDSS). vol. 3, pp. 191–206 (2003)
11. Graziano, M., Lanzi, A., Balzarotti, D.: Hypervisor memory forensics. In: International Workshop on Recent Advances in Intrusion Detection. pp. 21–40. Springer (2013)
12. Holz, T., Raynal, F.: Detecting honeypots and other suspicious environments. In: Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC. pp. 29–36. IEEE (2005)
13. Hoopes, J.: Virtualization for security: including sandboxing, disaster recovery, high availability, forensic analysis, and honeypotting. Syngress (2009)
14. Intel: Intel® 100 Series and Intel® C230 Series Chipset Family Platform Controller Hub (PCH) (5 2016)
15. Jiang, X., Wang, X.: "Out-of-the-box" Monitoring of VM-based High-Interaction Honeypots. In: International Workshop on Recent Advances in Intrusion Detection. pp. 198–218. Springer (2007)
16. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction. ACM Transactions on Information and System Security (TISSEC) **13**(2), 12 (2010)
17. Joshi, R., Sardana, A.: Honeypots: a new paradigm to information security. CRC Press (2011)
18. Kittel, T.: Library to parse dwarf information and access/use it in C/C++. `https://github.com/kittel/libdwarfparser` (2014), [accessed 2018-02-17]
19. Krawetz, N.: Anti-honeypot technology. IEEE Security & Privacy **2**(1), 76–79 (2004)
20. Lengyel, T.K.: Stealthy monitoring with xen altp2m, `https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/`, [accessed: 2018-02-13]

21. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference (2014)
22. Lengyel, T.K., Neumann, J., Maresca, S., Kiayias, A.: Towards hybrid honeynets via virtual machine introspection and cloning. In: International Conference on Network and System Security. pp. 164–177. Springer (2013)
23. Oosterhof, M.: Cowrie SSH/Telnet Honeypot. `https://github.com/micheloosterhof/cowrie` (2014), [accessed 2018-02-17]
24. Payne, B.D.: Simplifying virtual machine introspection using libvmi. Sandia report (2012)
25. Portokalidis, G., Bos, H.: Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008. pp. 287–299. Eurosys '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1352592.1352622
26. Portokalidis, G., Slowinska, A., Bos, H.: Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. SIGOPS Oper. Syst. Rev. **40**(4), 15–27 (Apr 2006). https://doi.org/10.1145/1218063.1217938
27. Sentanoe, S., Taubmann, B., Reiser, H.P.: Virtual machine introspection based ssh honeypot. In: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems. pp. 13–18. ACM (2017)
28. Spitzner, L.: Know your enemy: Genii honeynets. The Honeynet Alliance (2005)
29. Srivastava, A., Giffin, J.: Tamper-resistant, application-aware blocking of malicious network connections. In: Recent Advances in Intrusion Detection. pp. 39–58. Springer (2008)
30. Stuart: High-interaction MitM SSH honeypot. `https://github.com/magisterquis/sshhipot` (2016), [accessed 2018-02-17]
31. Tamminen, U.: Kippo - SSH Honeypot. `https://github.com/desaster/kippo` (2009), [accessed 2018-02-17]
32. Taubmann, B., Frädrich, C., Dusold, D., Reiser, H.P.: Tlskex: Harnessing virtual machine introspection for decrypting tls communication. Digital Investigation **16**, S114–S123 (2016)
33. Taubmann, B., Rakotondravony, N., Reiser, H.P.: Cloudphylactor: Harnessing mandatory access control for virtual machine introspection in cloud data centers. In: Trustcom/BigDataSE/ISPA, 2016 IEEE. pp. 957–964. IEEE (2016)
34. Taubmann, B., Rakotondravony, N., Reiser, H.P.: Libvmtrace: Tracing virtual machines (2016)
35. Testa, J.: SSH man-in-the-middle tool. `https://github.com/jtesta/ssh-mitm` (2017), [accessed 2018-02-17]
36. Tuzel, T., Bridgman, M., Zepf, J., Lengyel, T.K., Temkin, K.: Who watches the watcher? detecting hypervisor introspection from unprivileged guests. Digital Investigation (2018)
37. Uitto, J., Rauti, S., Laurén, S., Leppänen, V.: A survey on anti-honeypot and anti-introspection methods. In: Rocha, Á., Correia, A.M., Adeli, H., Reis, L.P., Costanzo, S. (eds.) Recent Advances in Information Systems and Technologies. pp. 125–134. Springer International Publishing, Cham (2017)
38. Wang, P., Wu, L., Cunningham, R., Zou, C.C.: Honeypot detection in advanced botnet attacks. International Journal of Information and Computer Security **4**(1), 30–51 (2010)