The All Seeing Eye and Apate: Bridging the Gap between IDS and Honeypots

Christoph Pohl and Hans-Joachim Hof

MuSe - Munich IT Security Research Group
Munich University of Applied Sciences
Munich, Germany
Email: {christoph.pohl0, hof }@hm.edu

Abstract—**Timing attacks are a challenge for current intrusion detection solutions. Timing attacks are dangerous for web applications because they may leak information about side channel vulnerabilities. This paper presents a methodology that is especially good at detecting timing attacks. Unlike current solutions, the proposed Intrusion Detection System uses a huge number of sensors for vulnerability detection. Honeypots are used in IT Security to detect and gather information about ongoing intrusions by presenting an interactive system as attractive target to an attacker. The longer an attacker interacts with a honeypot, the more valuable information about the attack can be collected. Honeypots should appear like a valuable target to motivate an attacker. This paper presents, in addition to the possibilities of timing attack vulnerabilities, a novel way to inject honeypot and analysis capabilities in any software based on x64 or i386 architecture. It fulfills two basic requirements: it can be injected into machine code without the need of recompilation and it can be configured during runtime. This means the honeypot is able to change the behavior of any function during runtime. The concept uses sophisticated stealth technologies to provide stealthiness. In conclusion, the research presents a novel way to detect side channel vulnerabilities and an inbuilt hypervisor to provide configurable honeypot capabilities to explore these vulnerabilities to an attacker. The proposed solution in this paper offers a highly configurable injection technology, which can change the behavior of any function without the need of recompilation or even reinstallation. It is able to provide these capabilities in the kernel or userland of actual *Nix systems.**

***Keywords*—intrusion detection; honeypot; virtualisation; sensor; brute force; timing**

I. Introduction

This paper is an extended version of [1]. It also extends another research, published by the authors in [2], [3]. Hence, this research paper is based on those publications and extends them with a novel way of honeypot creation.

Intrusion Detection Systems (IDS) in combination with firewalls are the last defense line in security when protecting web applications. The purpose of an IDS is to alert a human operator or an Intrusion Prevention System that an attack is in preparation or currently taking place.

One common challenge for web applications is the detection of timing attacks. A timing attack is an attack, which uses time differences between different actions to gain information. Intrusion Detection Systems typically use sensors to collect data. In this work, a sensor describes a data source that provides data useful for attack detection. Useful in this context means that the data must be linked to actions of a web application. Data of sensors is analyzed by All-Seeing Eye to detect attacks.

Usually, honeypots can be classified into Low- and High Interaction Honeypots. A Low Interaction Honeypot is able to simulate services or system environments. A High Interaction Honeypot provides a real exploitable system.

A common challenge in honeypot creation, is to inject exploits into a High Interaction Honeypot. The provider of such a honeypot needs to install exploitable software or to inject vulnerabilities into a software. This means a high consumption of resources.

The major reasearch question in this research is twofold:

- Is it possible to detect timing attack vulnerabilities and to identify the correct function, reponsible for this leak?
- How to change the behavior of functions to deploy a honeypot (for example to provide timing attack vulnerabilities), but without the need of reinstallation, recompilation or resource expensive development?

The proposed solution in this paper offers a highly configurable injection technology, which can change the behavior of any function without the need of recompilation or even reinstallation. It is able to provide these capabilities in the kernel or userland of actual *Nix systems. This manipulation technology allows the provider to present different environments or behavior depending on current system status. For example: the attacker knows that a system is based on ext4-File system and uses a standard hard drive (SATA based) without any virtualisation. He expects that the system will have a throughput of about 65 MB/s. The real honeypot system, based on ESXI virtualisation with extensive caching has a throughput of 550MB/s (ESXI will cache all IO in RAM). To scale down the system, the provider needs to install the honeypot on the expected system, or to rewrite the syscall for writing and reading. This means a lot of overhead in recompilation the kernel or installation on specific hardware.

The proposed solution is able to hook functions and provide a hypervisor-like technology, which makes it possible to

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

change the behavior without the need of any compilation, nor installation.

Another possibility is to inject a rule engine for function parameter, system parameters and the result generation of any function. The honeypot provider is able to formulate rules which can change the behavior based on those parameters. For example, the provider is able to present a file structure for PID 42 and a completely different file structure for PID 84. This manipulation is able to decoy an attacker or even to suppress harmful actions. A rule just needs to prevent a system call by returning an error code.

For productive usage, the honeypot should not be detectable by an attacker (or just with sophisticated analysis tools). It must also provide a low overhead in time consumption (performance).

The proposed solution fulfills all requirements. Hence, it is a novel way to build easy to configure honeypot systems.

The rest of this paper is structured as follows: Section II presents related work. Section III describes the concept and implementation of the sensors used by All-Seeing Eye. The use of multiple sensors to detect intrusions is described in Section IV. Section V evaluates All-Seeing Eye under different attacks, especially timing attacks. Section VI describes a novel way to inject honeypot technologies in a running system. Section VII evaluates this technology with different settings. Section VIII concludes the paper and gives an outlook on future work.

II. Related Work

Anomaly detection is based on the hypothesis that there are deviations between normal behavior and behavior under intrusion [4], [5], [6], [7]. Many techniques have been researched for the detection like network traffic analysis [8], [9], [10], statistical analysis in records [11] or sequence analysis with system calls [12], [13], [14], [15]. A combination of this research with anomaly detection methods based on multiple sensors allows to find yet unknown attacks. Configuring intrusion detecting systems for one distinct system or one distinct vulnerability needs configuration with current solutions. The solution presented in this paper does not need any configuration.

In [13], [14], it is proved that call chains of system calls show different behavior under normal conditions and under intrusion, hence intrusion detection is possible. However, a normal model must be trained using learning data to detect attacks. In [14], it is shown that normal behavior produces fingerprintable signatures in system call data. A deviation from these signatures is defined as intrusion. This method is restricted to the usage of system calls and does not use more fine granular sensor data. In [16], a way to detect anomalies with information flow analysis is shown. Profiling techniques are used, injecting small sensors in a running application. They propose a model with clusters of allowed information flows and compare this normal model against actual information flow. Similar models are proposed in [17], [18], [19]. This approach is similar to our approach, but [16] focuses on offline audits for penetration testing. The approach presented in this paper is intended to be used online, hence it does not analyze the whole information flow but focuses on the method call chain, and is therefore more efficient.

In [20], it is shown that vulnerability probing can be detected using multiple sensors, especially sensor that calculate the possibility a resource is called by a user. These sensors are called access frequency based sensors. However, the system presented in [20] needs a lot of information about the system to protect (e.g., patterns describing legitimate resource calls), hence is difficult to deploy in the field. The solution presented in this paper does not need any configuration.

A well known honeypot tool, based on LKM for 2.6 Linux Kernel, is Sebek [21][22]. Sebek is primarily used for logging purposes in High Interaction Honeypot. Thus, it provides several possibilities focused on logging (like logging via network or GUI). In [23][24], ways to detect Sebek are described. Sebek does not provide the possibility to manipulate system calls as Apate does.

Another approach for monitoring systems is to use virtual machine introspection and system view reconstruction. This approach is used, e.g., in [25][26][27]. This approach is stealthier then Apate, because the introspection is done by the hardware layer of the virtual machine. However, Apate also provides means to manipulate the behavior of system calls, which is not supported by [25][26][27].

SELinux [28] is a well known tool, which inserts hooks at different locations inside the kernel. This provides the possibility for access control on critical kernel routines. SELinux can be controlled on a very fine granular level with an embedded configuration language. While SELinux is very useful in hardening a kernel, it is not designed for honeypot purposes. Especially, it lacks in the possibility to decoy the attacker with “wrong” information.

Grsecurity [29] with PAX [30] is similar to Apate. However, it greatly differs in ease of deployment and ease of configuration [31]. It also lacks in the possibility to decoy the attacker with “wrong” information.

In conclusion, non of the mentioned related work fulfill all requirements. Apate fulfills all requirements, hence is a useful building block for upcoming High Interaction Honeypots.

III. Sensors for a Massive Multi-Sensor Zero-Configuration Intrusion Detection System

A sensor describes a data source that provides useful data for attack detection. Useful in this context means that the data must be linked to actions of a web application. Data of sensors is analyzed by the proposed Intrusion Detection System All-Seeing Eye to detect attacks. Sensors are:

- Already available data sources like memory consumption of an application.
- Software sensors inserted into a web application or a web application framework.
- Sensors in the underlying operating system p.ex. sensors in the glibc or in system call routines

All-Seeing Eye depends on the availability of a large number of sensors that can be used for attack detection.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A. Sensor Implementation

Software sensors are implemented by injecting hooks at the beginning and end of functions. Hence, hooks are called before and after code execution of a method. With this approach, e.g., it is possible to measure the method execution time for each method used. It is also possible to identify the order of method execution. Hooks are injected directly into bytecode. It is not necessary to recompile any application protected by All-Seeing Eye. It is not necessary to perform any configuration for the web application that should be protected, hence All-Seeing Eye is called "zero configuration". As injection the technology proposed in [1], [2] is used. For the testbed used for the evaluation presented in this paper the sensors are placed in OpenCMS [32]. OpenCMS is a well known and widely used framework for Content Management. All-Seeing Eye takes care that methods used by the protected web application do not clash with method names used by All-Seeing Eye. Sensor data is written to a log file for further analyses.

One way to minimize the output of sensors (and the number of data to write to the log file) is to produce no output for methods that have an execution time lower than the resolution of the timestamps (1 ms). It is suspected that these methods would not generate any interesting output as these methods are usually helper methods or wrappers.

IV. Massive Multi-Sensor Zero-Configuration Intrusion Detection System

This section describes the design of the proposed massive multi-sensor zero-configuration intrusion detection system All-Seeing Eye. All-Seeing Eye uses the software sensors, described in more detail in the last section, to calculate intrusion metrics. The metrics described in the following are focused on detection of outliers in timeline data values to detect brute force attacks. However, the approach presented in this paper is not limited to this attack class, it can be easily adapted to detect various other attacks. Even attacks on the business logic can be detected as the presented approach uses software sensors embedded in the code of an application. This is out of scope of this paper.

An advantage of All-Seeing Eye is that it allows to detect side channel attacks without knowledge of the web application which is to be protected. In the absence of an attack, there is a high correlation between method calls defined in a method chain. As shown in Section V a single call results in correlated calls (method chain) of other methods. The system under load shows the same correlations. These correlations are further called as fingerprint s. Under attack, however, the system shows a different behavior, hence allows to identify attacks, see Section V for details. All-Seeing Eye does not need a preconfigured or constructed normal model. For this approach the normal model is created from history. At time $t = 0$ it is always assumed that there is no attack, hence status c is always $c! = attack$. If there is no attack, the same fingerprint s should show up in each distinct time period T with the same probability. A deviation from the number of fingerprints (written as $|s|$) in a time period T is defined as possible intrusion. This behavior is well known, as stated in Section II. The new approach here is the lack of need to define what a similar request is. The normal model is built using a quantile function, where the result is called α. α uses a floating history time period, which is defined as $n \times T$ and $t \in T$ are in state $c! = attack$.The multiplier n defines how much of the history is used. To control the sensitivity of the system, a configuration parameter p is used. In normal model α, a deviation is detected by:

$$c = \begin{cases} attack & \text{if } |s_{currentT}| \geq \alpha \times p \\ !attack & \text{if } |s_{currentT}| < \alpha \times p \end{cases} \tag{1}$$

This calculation is robust against statistical outliers and can be evaluated fast enough for real time calculation, in combination with structures related to sort optimization. In further researches these calculations will be done (together with other sensor calculations) with a graphical processing unit.

V. Evaluation

For the evaluation of the massive multi-sensor zero-configuration intrusion detection system, two typical attacks on web applications are used: timing attacks and vulnerability probing. Especially timing attacks are hard to detect for common intrusion detection systems. For our test environment OpenCMS version 8.5.1 [32] is used as web application to protect. OpenCMS is a well known and widely used framework for Content Management.

A. Evaluation Environment

For the evaluation of All-Seeing Eye, a paravirtualized, openvz solution [33] is used. This approach has the advantage that is is very realistic compared to simulations. The presented hardware settings are the settings of the corresponding virtual machine. Table I lists hardware and software used for the evaluation.

TABLE I: Experimental setup

Hardware(Server)	
CPU	4 Cores (2.1GHZ on hostsystem)
Memory	6 GB Ram
Ethernet	Bridged at 1 GBit Nic
Software(Server)	
Server Version	Apache Tomcat 7.0.28 [34]
JVM	Sun 1.6.0.27-b27

B. Fingerprints of Normal Behavior

To validate the hypothesis, that requests to the same target have the same fingerprint, the following experiment has been conducted.

First, a baseline is established for all other experiments. To do so, several requests are sent to the server and the server is restarted after each request. No interfering processes are running on this server.

After establishing the baseline, the whole website is crawled in a second step, ensuring that requests are sequential. The crawler is configured to request a single page and all depending images and scripts. To test the software under load in the third step, another crawler requests the server with 20 concurrent users, with a delay of one second between each request/user. Overall, there were in average 20 requests per second for different websites. The second and third step have been repeated 100 times. In each run of the experiment, the

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

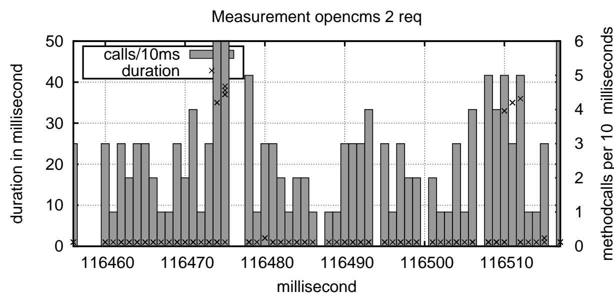

Figure 1: Two fingerprints of different requests

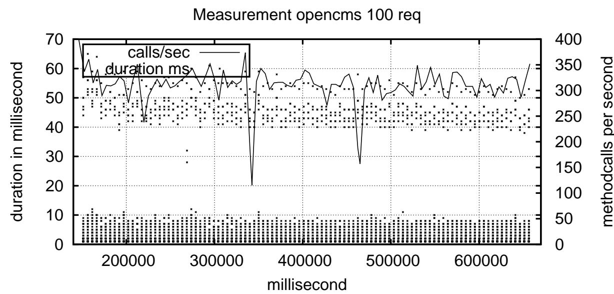

Figure 2: 100 requests on the same page

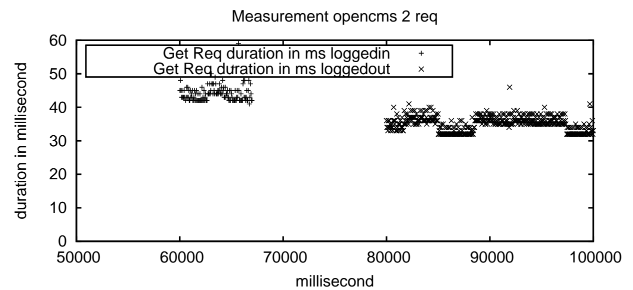

Figure 3: Time difference between logged in users and users not logged in on the start page

metrics described above produced unique fingerprints for every requested target. This can be seen in Figure 1. In this figure the fingerprint of the start page and the request to the login page are extracted from the logged data. Under load the signature looks like the picture presented in Figure 2.

The result in Figure 1 shows that there are stable correlations between method calls. For better readability, points that differ less then 3 milliseconds are averaged. The experiments show that it is possible with All-Seeing Eye to identify similar requests using their fingerprint.

C. Fingerprints of Information Leakage and Probing for Vulnerabilities

OpenCMS version 8.5.1 has a known information leakage vulnerability, as described in [35]: using the default setting there is no limit for failed logins per time period. Also, a large amount of information is given in error messages, especially the error message "this username is unknown", if the given user name does not exist and "password is wrong", if the given password is wrong for an existing user allow an attacker to find valid user names, by trying possible user names from a dictionary and using error messages to find out if an user name is valid. This attack can be detected with a statistical analysis to detect the brute force analysis II. To do so, a detection technique needs to identify if a single resource is called many times but with different parameter in the request header. A normal model is needed for allowing patterns to test for deviations of the normal model. This needs deep knowledge of the system to protect and the vulnerability itself. All-Seeing Eye is able to detect this attack (and also other probings using brute force attacks), without this knowledge about system and vulnerability.

To evaluate if All-Seeing Eye can recognize brute-force attacks, an experiment has been conducted where an attacker probes the login page and tries to identify valid user names. The following pattern was used to generate the login requests:

```
http://192.168.2.89:8080/opencms/.../index.html?
    action=login&username=username_1&password=
    passwordnotindb...snippedEnd
//username_1..username_n in dictionary
http://192.168.2.89:8080/opencms/.../index.html?
    action=login&username=username_n&password=
    passwordnotindb...snippedEnd
```

The attacker used a dictionary with 1000 names for the brute-force attack. To make detection harder, the attacker uses 50 different user agents as well as 20 different IPs. Only one valid username exists in the database.

Figure 2 shows a subset of 100 requests. From the figure, it is obvious that the probing attempts produce many similar fingerprints. It shows a high correlation between different requests, the sensor values and the order different sensors are called in one requests. This order and the values are stable over all requests. Hence, All-Seeing Eyes can easily detect a probing attack even if someone uses different header data. No a-priory knowledge of the system which is to be protected or the vulnerability itself is necessary.

D. Fingerprints of Timing attacks

OpenCMS version 8.5.1 is vulnerable to timing attacks as can be seen in Figure 3. The figure shows the times for loading of the start page for users that are already logged in as well as for users that are not logged in. A significant difference (849 ms to 798 ms) exists.

Using this timing difference an attacker can brute force user names by a dictionary attack. All logged in users can be detected. As with the information leakage and probing attack in Subsection V-C, current intrusion detection solutions need information about the system which is to be protected and the vulnerability to detect this attack.

To test if All-Seeing Eye is able to detect timing attacks without knowledge (zero-Configuration), the following experiment has been conducted: An attacker uses a dictionary of 1000 user names to execute the timing attack. Each request has different header data in the request only in the login name and the password as shown in listing 1.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Listing 1: Header Data

```
//successful login
http://192.168.2.89:8080/opencms/.../index.html?
    action=login&username=admin98&password=admin
    1...snippedEnd
//username not present, pwd not present
http://192.168.2.89:8080/opencms/.../index.html?
    action=login&username=usernamenotpresent&
    password=wrongpwd...snippedEnd
//username present with wrong password
http://192.168.2.89:8080/opencms/.../index.html?
    action=login&username=admin832&password=
    wrongpwd...snippedEnd
```

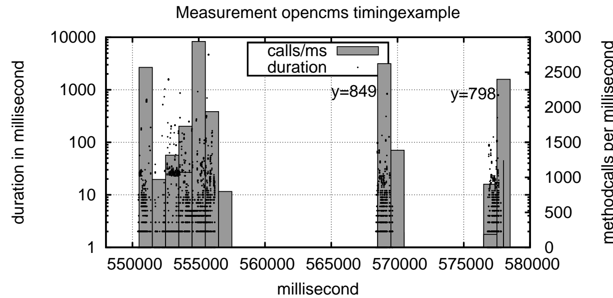

Figure 4: 100 requests on the same page

Figure 4 shows an example of fingerprints of the experiment.

The first fingerprint shows a successful login, the second fingerprint shows a login with no present username and third fingerprint shows a login with present username but wrong password. The results clearly show a correlation of the differences in order, time, and amount of method calls for each request.

VI. Bridging the Gap to Honeypot Deployment

As stated in Section III, it is possible to analyze an application for timing attack vulnerabilities without the need of sophisticated penetrations tests. However, whenever a provider wants to offer a vulnerability, he needs to install this exploitable piece of software. In the proposed example of timing attacks, the provider is able to identify the place where such a vulnerability has to be installed.

More generally, this place can be any function or a set of functions available on the target system. The provider needs to change this function or set of functions to change the behavior of this functionality.

This solution, further called APATE, intercepts functions and allows to execute custom routines in those functions. Figure 5 shows this interception strategy.

Any function call to the hooked function will be intercepted by a preprocessor hook. This hook leads to the hypervisor. Inside the hypervisor some custom code gets processed. The hypervisor and its language is explained in Section VI-E. The result of this routine decides on the action to invoke. Within this hypervisor process, it is possible to manipulate,

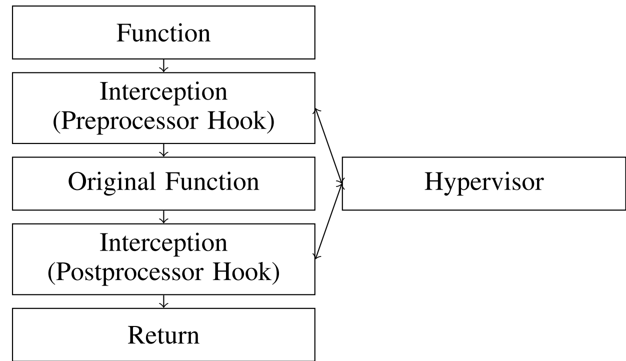

Figure 5: interception strategy of Apate

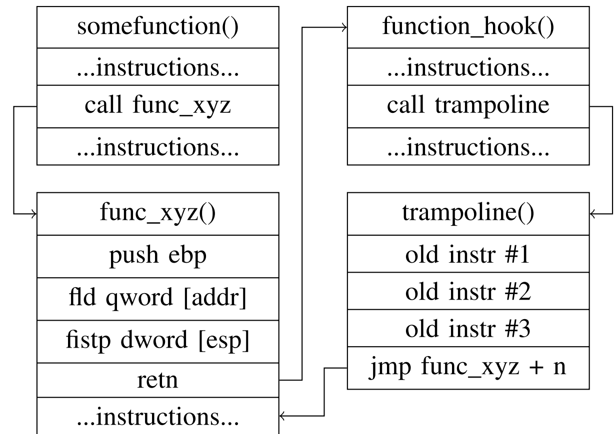

Figure 6: Hooking using a so-called trampoline

block and/or log the original function parameters and the further execution of the original function. The manipulation possibilities are explained in detail in Section VI-A. The postprocessing hook has the same capabilities than the preprocessor. In addition it is able to manipulate the return code of the original function.

To prevent detection, Apate is injected into the original function with a trampoline technology. Figure 6 describes this trampoline.

This technology, well known in rootkits for Windows or Linux, is explained in detail in [2]. The hook injector overwrites original code (located in `func_xyz` in Figure 6) with a jump to the hooking code. The hook will process the hypervisor. At the end of the hook, the trampoline gets called. The trampoline holds the overwritten code from the original code, and returns then back to the original code. This technology makes it hard for rootkit detection tools and is live patchable.

A. *Manipulating functions*

Apate and, therefore, the manipulation of functions can be configured with the help of a high level language. In detail the configuration high level language and the rule possibilities are explained in [2]. This high level language formulates rules (or any other functionality), which gets process by the hook. A brief overview over this functionality is described in Figure 7.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

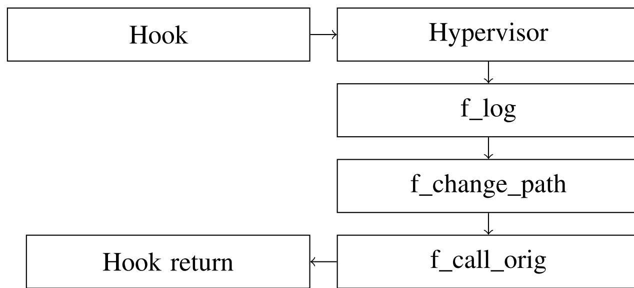

Figure 7: Conceptual Manipulation Strategy

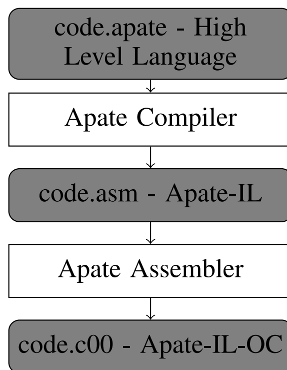

Figure 8: Configuration workflow of Apate

In this case the hypervisor works with three “rules”. The first rule logs the function call and its parameter, the second rule manipulates some parameter and the third rule proceeds the original function with the manipulated parameters. Of course, it is possible to write rules which delays the original function or prevent the execution of the original function.

In demarcation to [2] the high level language results in binary machine code like bytecode, which gets processed by the hypervisor. Figure 8 shows the compilation and assembling of the rules. A rule is formulated in a high level language (*code.apate*). This gets compiled over bison and flex to an intermediate Assembler like language (*code.asm*). This language is further called the Apate Intermediate Language or *Apate-IL*. This Apate-IL must be architecture dependent, as it provides addresses. In the assembling step the Assembler like code gets assembled to a machine code like language (*code.c00*, see also Section VI-E). This binary code is processable by the hypervisor and further called the Apate Intermediate Language Operational code or *Apate-IL-OC*.

B. The Apate High Level Language

The language provides a flexible language to build hooks for functions in any x86 or i386 architecture. It is able to define functions, reuse patterns, store variables and basic mathematical computation. With those abilities it is possible to build transparent rulesets for the hooking of functionalities. This section gives a brief overview over some core components of this language. The language is inspired by Haskell[36] and pf [37]. A more detailed description is given in [2].

Listing 9 shows some example source code for the Apate language. In this case, 3 different conditions will be generated.

```
define c1,c2,c3 as condition
define r1,r2 as rule
define a1,a2 as action
define cb1 as conditionblock
define rc1 as rulechain
define sy1 as syscall

let c1 be testforpname
let c2 be testforparam
let c3 be testforuid
let a1 be manipulateparam
let a2 be log
let sy1 be sys_open

let cb1 be {(c1("mysql") && c2(0;"/var/\
   lib/mysql/*"))}

let r1 be {cb1->a1(0;"/var/lib/mysql/*" \
  ;"/honey/mysql/")}
let r2 be {{c3(">",0)}->a2()}
let rc1 be {r2,:r1} // :defines exit

bind rc1 to sy1
```

Figure 9: Example Sourcecode Apate language

`c1` tests the actual process name against a given name. `c2` tests if a parameter of the hooked function is equal to another value. `c3` tests if the uid is equal to a given value. The actions `a1,a2` manipulates a parameter and write some log. The `bind` statement binds the rules to the syscall `open`. This will build a hook in the `open` function. In conclusion, the rulechain rewrites the parameter to any call for `open` and when the param inherits */var/lib/mysql* to the path */honey/mysql*. This redirection of the path gets logged for further analysis.

C. The Apate Intermediate Language

The intermediate language is based on the concept of the Intel i386 assembly language. A command consists of the command and at maximum two parameters. As minimum a command has no parameter. A parameter can be a constant, a register, a value of an register, a pointer or the value of a pointer. The hypervisor has an own stack, registers and memory management. This leads to the basic concept in Listing 2:

Listing 2: Apate-IL Concept

```
labelname:
command <dest> <source>
command <dest>
command
```

Technically, a label gets assembled to a [nop] and all references to the label are transformed to the appropriate address. Apate-IL consists of the basic commands like [nop],[jmp|jz|jnz],[add|sub|mul|div],[cmp|test],[call],[ret],[push|pull] and a few other commands for convenience.

In addition to a standard assembly language, there are commands especially designed for honeypot purposes:

- [sleep] - delay for ticks
- [inwind] - does some sophisticated jumps for anti disassembling

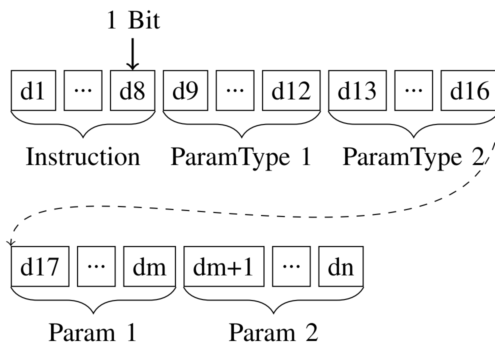

Figure 10: A command in Apate IL-OC

- [adebug] - does some anti debugging techniques
- [asm] - executes real Machine code
- [fcall] - calls “real” functions
- [oops] - provides kernel oops
- [fpush| fpull] - makes it possible to interact with the underlying process (the original function)

The [sleep] command consumes real CPU-Time and provides a sleep functionality. The [inwind] command is just a wrapper for inwind calls. This technology makes it possible to jump into a real command. This means, that whenever a “long” command, like the [move <dest> integer] command uses the constant integer to formulate a new “short” command like [jz 5], which is only four bytes over all and has the size of an integer, the inwind command calculates the jump address to this “obfuscated” command. This is just for convenience to avoid annoying calculations. The command [adebug] provides some, out of scope in this paper, technology against debugging. The command [asm] maps real machine code(provided as shellcode) into RAM and runs it. This makes it possible to optimize some calculations. The [fcall] command is able to call real functions on any given address and is used as a wrapper for the real [call] command. The [oops] command provides in combination with a special kernel system call a real system kernel oops. The [fpush| fpull] commands can interact with a special stack, provided by the hypervisor, to communicate with the hooked function. Both commands are used to read parameters and store them back.

D. The Apate Intermediate Language Operational Code and the Assembler

The Apate-IL-OC is assembled from the Apate Intermediate Language. It is a binary, optimized and preprocessed version of the Apate-IL. A single command is shown in Figure 10. An instruction is a 8 Bit operational code, followed by 2 nibbles, representing the parameter types. This type decides if the param is an address (64 or 32 Bit), an Integer (64 or 32Bit) constant, or a register ($r0 \ldots 9$). The decision for 64 or 32 Bit is architecture dependent and uses the length of size_t. The following parameters can have different sizes, depending on their type. Some commands, like [nop], does not have any parameters and, therefore, there is no need for type decision. Such a command needs only 8 Bit. Whenever a command has just one parameter, the second nibble has the value 0x0, which means no type.

A command can have any operational code between 0x00 and FE. The FE opcode is used as a debugging trap.

During the assembling part, the assembler lexes the Apate Intermediate language. With the help of Yacc an AST gets built from the sourcecode. This AST gets preprocessed by the assembler. Any label and references to labels gets transformed to addresses in a first step. In a second step, the preprocessor checks for wrong (or invalid) addresses, invalid commands or irregular command chains. In a third step, any data that should be stored in a data section gets collected.

Following this step, the assembler will transform the AST to the Apate Intermediate Operating Language, which is in fact some binary code.

In a last step, the assembler generates a binary representation, consisting of a header (inspired by a ELF header), the Apate IL-OC section and a data section.

Out of scope in this research is the ability to store some information about encoding, different instructions sets and other information, which are needed for sophisticated obfuscation and anti disassembling technologies.

E. Hypervisor engine

The hypervisor needs to fulfill different requirements:

- Provide a turing complete language for flexible rules
- Provide a hypervisor to process this language
- Provide a hypervisor that has low resource consumption
- Provide the ability to call system functions, change process memory content
- Embed real x64 or i386 machine code

Another requirements like different instruction sets, Huffman encoding, inbuilt obfuscation and anti-disassembling strategies are not in scope of this paper but part of the real hypervisor prototype. Those (not listed) requirements make it substantially harder to detect and analyze the hypervisor system and their rules.

The Apate Hypervisor has a classical design, based on register, stack and an instruction array. Figure 11 describes the basic architecture for the Apate hypervisor.

The Code Section holds the pure Instruction Code, as provided by the assembler. This Instruction Code gets processed by the Decoder and Execution Unit. The Data Section stores all constants from the source file. Most of the following data storage units are architecture dependent. The underlying Host architecture decides if a value is 32 Bit or 64 Bit.

The Register is able to store $32|64$ Bit Values and can be compare with the General Purpose Register from other architectures like x64. The Flag register holds Flags like Zero Flag or Traps for the Debugger. The stack can keep 1024 $32|64$ Bit values. The Pointer Register is used to store pointer like the instruction pointer, the return pointer and the stack pointer.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

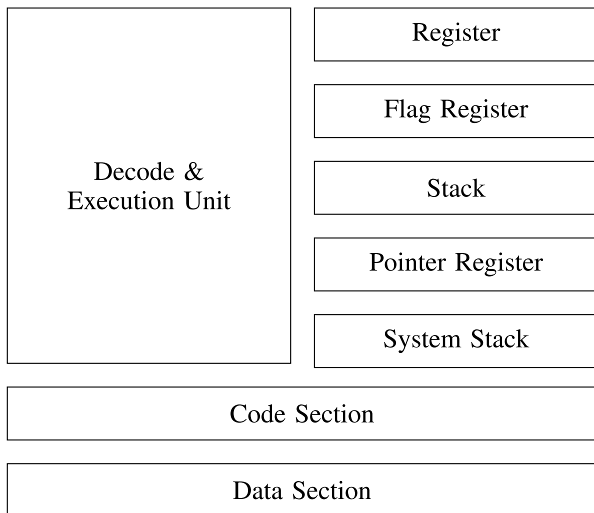

Figure 11: Basic concept of Hypervisor

The System Stack is a special stack to interact with the host system. The hypervisor is able to write values to and read values from this stack, and the internal language is able to do the same. With this communication stack, parameters and other values can be injected into the hypervisor based software.

Table II shows some of the components in Apate Hypervisor.

TABLE II: Register and Stack in Apate

Type	Name	Number	Size (Bit)
Instruction Section	-	n	8
Data Section	-	n	8
General Purpose Register	r0...r9	10	64|32
Stack	-	1024	64|32
Instruction Pointer	eip	1	32
Stack Pointer	stp	1	32
Return Pointer	rtp	1	32
Flag Register	-	1	8
Flag	Zero-Flag	1	1
Flag	Sign-Flag	1	1
Flag	Error-Flag	1	1
Debugger Flag	Next-Flag	1	1
Debugger Flag	Trap-Flag	1	1

The CPU starts with the first address in the code section stored in the instruction pointer. Due performance issues the opcode gets interpreted by index to function pointer translation. In combination with the two nibbles (1 Byte) to identify the correct function it needs two steps and $512 \times 32|64$ Bit to identify an opcode target.

The values stored in the nibbles to identify the param types are used to read the parameter values from the instruction code.

Together with the opcode, the Execution Unit calculates the result of this operation and then sets the next instruction pointer.

VII. EVALUATION

The Apate Hypervisor needs to process the custom code inside a hook in an efficient way. Performance tests should assure that Apate is able work under productive usage. The most important factor is the overhead of the hypervisor and the processing of common used code patterns. To evaluate the performance of Apate a common command in *Nix system is hooked.

The experimental setup is shown in Table III.

TABLE III: Experimental setup

Host System	
CPU	2 x XEON
Memory	64 GB Ram
Ethernet	Bridged at 1 GBit Nic
Virtualisation	ESXI
Measurement System	
CPU	2 x VMWare CPU
Memory	8GB Ram
HDD	30 GB Backed by ESXI
HDD Format	ext4

The test scenario is a clone of the cp-command. Aside from command parameter processing, the command uses the commands in Listing 3.

Listing 3: Example code for cp-command

```
for (;;) {
        readed = read(filenosrc,buffer,buffer_size
            );
        if (!readed) {
                //eof
                break;
        }
        written = write(filenodst,buffer,(size_t)
            readed);
}
```

This piece of code reads `buffer_size` bytes from file `filenosrc` and writes `readed` bytes back to file `filenodst`. The variable `buffer_size` is one of the performance variables under *Nix Systems and corresponds to the copybuffer.

The performance test generates a file with 100MB and random data. Then this file gets copied with the cp-command to another file. As reference, a measurement without any hooking has been done. This reference is further called $m1$. Each measurement has been done with different values of `buffer_size`.

Let $l_n \times 1024$ be the value of `buffer_size`. l_n starts with 0. However, a `buffer_size` of 0 is not usable. In this case the `buffer_size` is set to 1.

$$l_n = \begin{cases} l_{n-1} + 4 & \text{if } l_n < 100 \\ l_{n-1} + 100 & \text{if } 100 \leq l_n \\ & \wedge l_n < 1,000 \\ l_{n-1} + 1,000 & \text{if } 1,000 \leq l_n \\ & \wedge l_n \leq 10,000 \end{cases} \quad (2)$$

Each size of l_n has been tested 10 times.

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

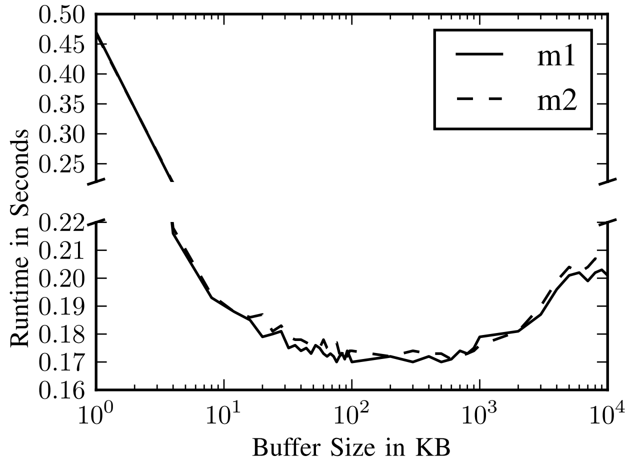

Figure 12: Performance Measurement m1 against m2

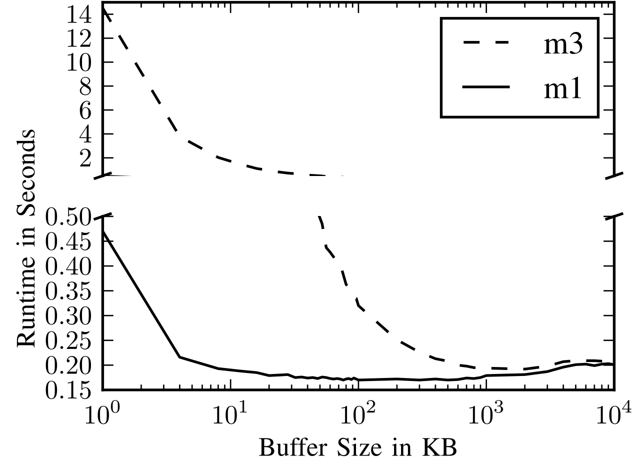

Figure 13: Performance Measurement m1 against m3

To test the overhead the parent function (in this case the function that inherits the code in Listing 3) gets hooked by Apate. The custom code inherits 2500 compare statements. Hence, those statements reflects 50 string compares with 50 chars each. This test is further called $m2$.

Figure 12 compares the reference $m1$ with the overhead of the hypervisor in $m2$.

This measurement shows that the `buffer_size` value has an impact to the overall performance. The outstanding performance for reading and writing (with over 500MB/sec) can be explained with heavy caching due the ESXI Host system. This measurement also shows that the overhead for the hypervisor is not significant.

The next measurement ensures that the hypervisor is able to server even a high amount of custom code calls. For this the `read` function has been hooked with the same custom code than before. Dependent on the `buffer_size` more or less hooks are called by the cp-command. This measurement is further called $m3$. Figure 13 shows the difference between $m1$ and $m3$.

This measurement shows that Apate is able to serve custom code even with a high amount 100.000 custom code calls.

Table IV concludes all 3 measurements.

TABLE IV: Performance Description m1,m2,m3

Measurement	m_1	m_2	m_3
Measurements	4,390	4,390	4,390
min(runtime sec)	0.16	0.17	0.19
max(runtime sec)	0.48	0.48	15.73
mean(runtime sec)	0.186	0.188	0.810
sd(runtime sec)	0.042	0.042	2.017
Throughput(MB/s) mean	536.320	530.566	303.411

In conclusion, the measurement shows that the amount of hooks does not interfere with the performance of one single hook. The Standard deviation is also in an expected range. The throughput row shows that even under high load, and when every system call to —open— is hooked even under worst scenario (`buffer_size==0`), the throughput rate is better than the throughput rate from a standard HDD.

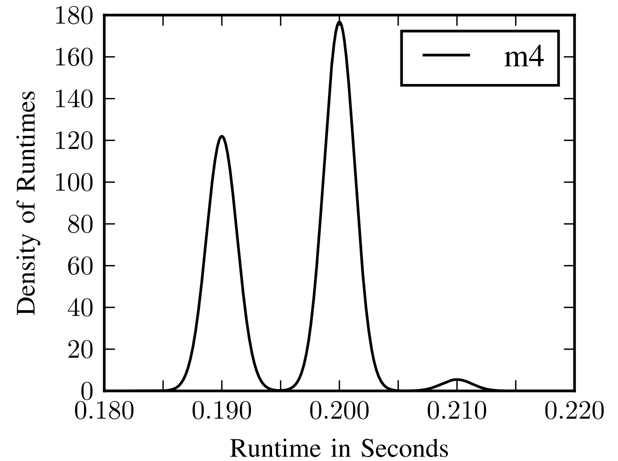

Figure 14: Density of Runtime at m4

The performance test should also show the influence of the amount of operations that are running in the hypervisor.

For this the cp-command has been tested 1000 times with a `buffer_size=8` and a file size of 100MB.

Figure 14 shows the reference measurement. The measurement $m4$ shows the reference measurement without any hypervisor influence. The x-axis shows the runtime in seconds. The y-axis shows the density of each runtime in all measurements. The measurement shows, that the measurement is only in a small range of runtimes, which means a stable runtime for a given buffer and file size to copy. Figure 15 shows the throughput of the reference measurement in MB/s. The throughput is generated with a weighted exponential moving

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

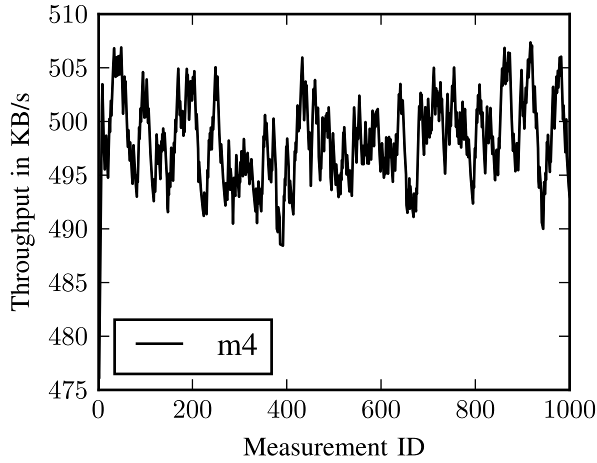

Figure 15: Throughput behavior m4

average over 20 measurements for more clearance. It also shows that the performance is stable through all measurements.

Table V describes the m4 measurement data.

TABLE V: Performance Description m4

Measurement	$time$	$throughputMB/s$
Measurements	1000	1000
min	0.19 sec	465.029762
max	0.21 sec	513.980263
mean	0.196170	498.168027
standard deviation	0.005221	13.303990

The standard deviation shows that with a given buffer and file size, the command has a stable performance behavior.

The measurement $m5$ uses the same setting than in measurement $m4$, but a hook which calls the hypervisor with a custom code is called once. The hypervisor custom code inherits 2500 compare statements, like in measurement $m2$. Figure 16 shows the density of the measurement $m5$.

Compared to the reference measurement, the data from $m5$ shows that the hypervisor has only a small influence on performance. However, it also shows that the performance of the hypervisor is stable along 1000 measurements. Figure 17 shows the throughput of measurement $m5$ in MB/s.

The throughput is generated with a weighted exponential moving average over 20 measurements. This illustration of data shows that the performance is stable over all measurements. It also shows that it is possible to keep the throughput rate, even with the hypervisor enabled.

Table VI describes measurement $m5$.

The standard deviation shows, that the hypervisor has a stable performance behavior.

In measurement $m6$, the behavior of a productive honeypot scenario is shown. In this case a honeypot, exploitable by timing attacks and with the ability to appear with an HDD with

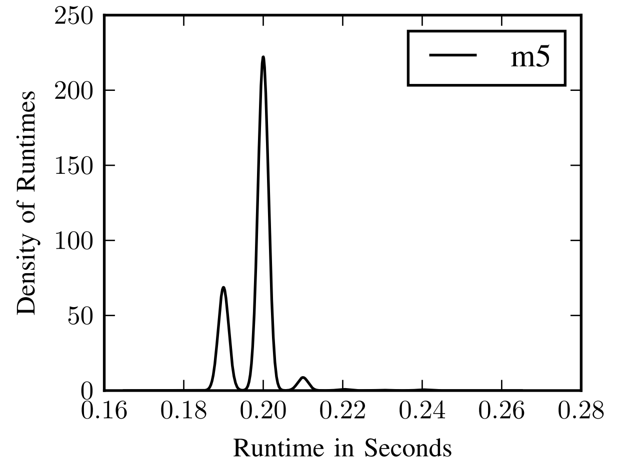

Figure 16: Density of Runtime at m5

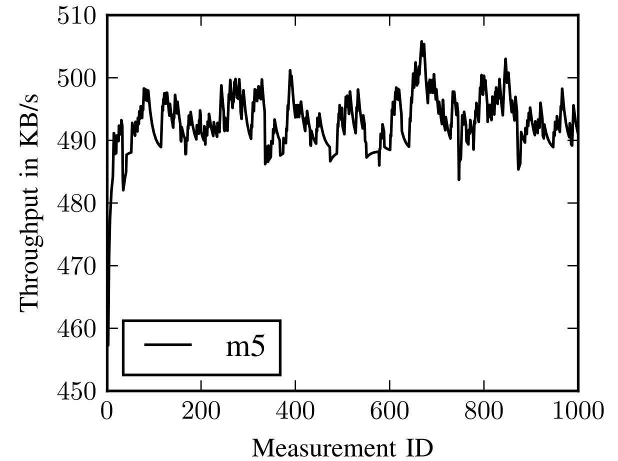

Figure 17: Throughput behavior m5

TABLE VI: Performance Description m5

Measurement	$time$	$throughputMB/s$
Measurements	1000	1000
min	0.19 sec	406.901042
max	0.24 sec	513.980263
mean	0.198180	513.980263
standard deviation	0.005265	12.897778

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

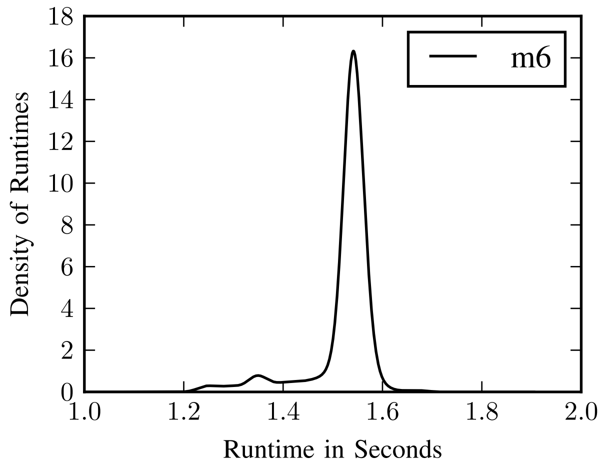

Figure 18: Density of Runtime at m6

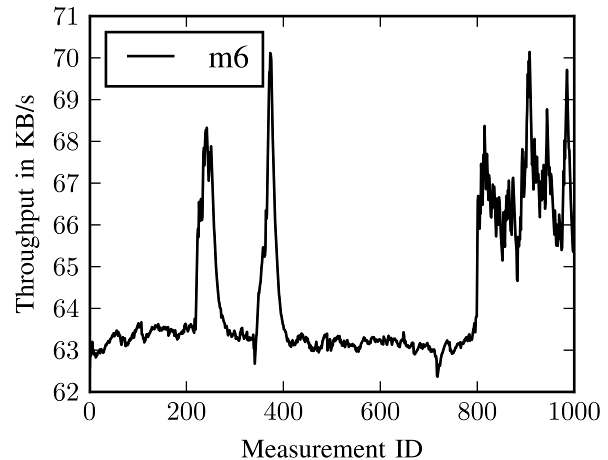

Figure 19: Throughput behavior m6

another throughput rate is built with Apate. The hypervisor should provide a throughput rate of 65 MB/s, instead of the real throughput rate from $m4$. The scenario is the same like in $m4$ and $m5$. A random generated file of 100MB gets copied and the time gets measured. This measurement has been repeated 1000 times.

Let t_{real} be the time to copy one KB on the real system. The real system is the honeypot system. Let t_{honey} be the time to copy one KB on a fictional honeypot system. In the case of this measurement, it is a system with a HDD that provides a throughput rate of 65MB/sec. Let b be the buffer size, used by the cp-command. The hook with the hypervisor is bound to the `read`-function. To decoy the attacker, the hook needs to sleep for a time t_{sleep}, such that:

$$t_{sleep} = b \times (t_{honey} - t_{real}) \tag{3}$$

The calculation $t_{honey} - t_{real}$ is a constant t_{wait} to describe the honeypot system.

The custom code, processed by the hypervisor, reads the variable `buffer_size`. Then, it multiplies this variable with the sleeping rate constant t_{wait}. The result is used to trigger the sleeping function. The full functionality in Apate-IL is shown in Listing 4.

Listing 4: Sleeping function in Apate-IL

```
fpull r1
fpull r2
mul r1 r2
sleep r1
exit
```

The first and second line reads the `buffer_size` and the constant t_{wait} and stores them in register `r1` and `r2`. In the third line, both values stored in the registers gets multiplied. The result gets stored in the most left register `r1`. The sleeping command uses this value to sleep. The exit command quits the hypervisor processing.

Figure 18 describes the density of $m6$.

Compared to $m4$ the measurements show that the sleeping function is able to generate a completely different runtime scenario. It also shows that the performance is stable along all measurements.

Figure 19 shows the throughput of measurement $m6$ in MB/s.

The throughput is generated with a weighted exponential moving average over 20 measurements. This illustration also shows that the throughput is stable over all measurements. It also describes that the target rate of 65 MB/s is stable over all measurements with a small deviation.

Table VII describes the data for measurement $m6$.

TABLE VII: Performance Description m6

Measurement	$time$	$throughputMB/s$
Measurements	1000	1000
min	1.23 sec	58.128720
max	1.68 sec	79.395325
mean	1.521950	64.290778
standard deviation	0.063464	3.020391

This description shows that the mean throughput is very near the expected throughput rate. It also shows that the standard deviation is low, so that the honeypot is able to provide a stable throughput rate.

VIII. CONCLUSION

This paper presents Apate, a hypervisor for custom code to hook any function in a *Nix Kernel or userland-program. With the possibilities of the proposed solution to detect timing attacks, it is possible to identify the best place to hook a function and to inject the honeypot component. Apate works on a function level, and is able to log, block or manipulate functions. The evaluation shows that Apate has only a low performance overhead and can be used in productive scenarios. The evaluation also shows that Apate is able, with only four

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

commands, to build a honeypot for timing attacks, and to lure an attacker with timings from a completely different hardware system, without any installation, compilation or any other time consuming configuration. As future work, we will implement more commands for usage in honeypot systems. We also plan to include multiprocessing and a more advanced code section. At moment Apate is in a beta status, whenever it is more stable (the assembler does need some tweaking in error detection), Apate will be open source under github.

ACKNOWLEDGMENT

This work is part of the project "Sichere Entwicklung und Sicherer Betrieb von Webanwendungen" of the Munich IT Security Research Group funded by the Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst.

REFERENCES

[1] C. Pohl and H.-J. Hof, "The all-seeing eye: A massive-multi-sensor zero-configuration intrusion detection system for web applications," in *SECURWARE 2013, The Seventh International Conference on Emerging Security Information, Systems and Technologies*, 2013, pp. 66–72.

[2] C. Pohl, M. Meier, and H.-J. Hof, "Apate-a linux kernel module for high interaction honeypots," in *The Ninth International Conference on Emerging Security Information, Systems and Technologies – SECURWARE 2015*, 2015, pp. 133–138.

[3] C. Pohl, A. Zugenmaier, M. Meier, and H.-J. Hof, "B. hive: A zero configuration forms honeypot for productive web applications," in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 267–280.

[4] A. Patcha and J. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, no. 12, pp. 3448–3470, 2007, retrieved 2013-04-11. [Online]. Available: www.scopus.com

[5] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, p. 251261, retrieved 2013-04-11. [Online]. Available: http://doi.acm.org/10.1145/948109.948144

[6] G. Liepens and H. Vaccaro, "Intrusion detection: Its role and validation," *Computers & Security*, vol. 11, no. 4, pp. 347–355, Jul. 1992, retrieved 2013-04-11. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016740489290175Q

[7] D. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 222–232, 1987, retrieved 2013-04-11.

[8] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft, "Structural analysis of network traffic flows," in *Proceedings of the joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '04/Performance '04. New York, NY, USA: ACM, 2004, p. 6172, retrieved 2013-04-11. [Online]. Available: http://doi.acm.org/10.1145/1005686.1005697

[9] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, ser. IMW '02. New York, NY, USA: ACM, 2002, p. 7182, retrieved 2013-04-11. [Online]. Available: http://doi.acm.org/10.1145/637201.637210

[10] F. Silveira and C. Diot, "URCA: pulling out anomalies by their root causes," in *2010 Proceedings IEEE INFOCOM*, 2010, pp. 1–9, retrieved 2013-04-11.

[11] H. Javitz and A. Valdes, "The SRI IDES statistical anomaly detector," in *1991 IEEE Computer Society Symposium on Research in Security and Privacy, 1991. Proceedings*, 1991, pp. 316–326, retrieved 2013-04-11.

[12] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, p. 66, retrieved 2013-04-11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267555

[13] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998, retrieved 2013-04-11.

[14] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, "A sense of self for unix processes," in *1996 IEEE Symposium on Security and Privacy, 1996. Proceedings*, 1996, pp. 120–128, retrieved 2013-04-11.

[15] A. Frossi, F. Maggi, G. L. Rizzo, and S. Zanero, "Selecting and improving system call models for anomaly detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, U. Flegel and D. Bruschi, Eds. Springer Berlin Heidelberg, Jan. 2009, no. 5587, pp. 206–223, retrieved 2013-04-11.

[16] W. Masri and A. Podgurski, "Application-based anomaly intrusion detection with dynamic information flow analysis," *Computers & Security*, vol. 27, no. 56, pp. 176–187, Oct. 2008, retrieved 2013-04-11. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404808000369

[17] L. Feng, X. Guan, S. Guo, Y. Gao, and P. Liu, "Predicting the intrusion intentions by observing system call sequences," *Computers & Security*, vol. 23, no. 3, pp. 241–252, May 2004, retrieved 2013-04-11. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404804000732

[18] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *2006 IEEE Symposium on Security and Privacy*, 2006, pp. 15–62, retrieved 2013-04-11.

[19] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "LIFT: a low-overhead practical information flow tracking system for detecting security attacks," in *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39*, 2006, pp. 135–148, retrieved 2013-04-11.

[20] C. Kruegel, G. Vigna, and W. Robertson, "A multi-model approach to the detection of web-based attacks," *Computer Networks*, vol. 48, no. 5, pp. 717–738, Aug. 2005, retrieved 2013-04-11. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128605000083

[21] Honeynet Project, "Know Your Enemy: Sebek," 2003, retrieved: 07, 2015. [Online]. Available: https://www.honeynet.org/papers/sebek

[22] E. Balas, "Sebek: Covert Glass-Box Host Analysis," *login: THE USENIX MAGAZINE*, no. December 2003, Volume 28, Number 6, pp. 21–24, 2003.

[23] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005.* IEEE, 2005, pp. 29–36.

[24] M. Dornseif, T. Holz, and C. Klein, "NoSEBrEaK - Attacking Honeynets," in *Proceedings of the 2004 IEEE Workshop on Information Assurance and Security*, Jun. 2004, pp. 123–129.

[25] C. Song, B. Ha, and J. Zhuge, "Know Your Tools: Qebek – Conceal the Monitoring — The Honeynet Project," retrieved: 07, 2015. [Online]. Available: http://www.honeynet.org/papers/KYT_qebek

[26] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, "Virtual machine introspection in a hybrid honeypot architecture," in *Proceedings of the 5th USENIX Conference on Cyber Security Experimentation and Test*, ser. CSET'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 5–13.

[27] X. Jiang and X. Wang, ""Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots," in *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2007, pp. 198–218.

[28] NSA (Initial developer), "Selinux," 2009, retrieved: 07, 2015. [Online]. Available: https://www.nsa.gov/research/selinux/index.shtml

[29] Open Source Security, "grsecurity," 2015, retrieved: 07, 2015. [Online]. Available: https://grsecurity.net

[30] PAX Team, "Pax," 2015, retrieved: 07, 2015. [Online]. Available: https://pax.grsecurity.net

[31] M. Fox, J. Giordano, L. Stotler, and A. Thomas, "Selinux and grsecurity: A case study comparing linux security kernel enhancements," 2009.

[32] "OpenCms, opencms homepage," http://www.opencms.org, retrieved 2013-04-11. [Online]. Available: http://www.opencms.org

[33] "OpenVZ openvz linux containers," http://openvz.org, retrieved 2013-04-11. [Online]. Available: http://openvz.org

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[34] "Apache Tomcat apache tomcat," http://tomcat.apache.org, retrieved 2013-04-11. [Online]. Available: http://tomcat.apache.org

[35] "Owasp Top Ten," https://www.owasp.org/index.php/Category: OWASP_Top_Ten_Project, retrieved 2013-04-11. [Online]. Available: https://www.owasp.org/index.php/Category: OWASP_Top_Ten_Project

[36] S. Marlow, "Haskell 2010 language report," 2010, retrieved: 07, 2015. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[37] OpenBSD, "Pf: The openbsd packet filter," 2015, retrieved: 07, 2015. [Online]. Available: http://www.openbsd.org/faq/pf/

2016, © Copyright by authors, Published under agreement with IARIA - www.iaria.org