MAYALOK: A Cyber-Deception Hardware Using Runtime Instruction Infusion

Preet Derasari, Kailash Gogineni, Guru Venkataramani
The George Washington University, Washington, D.C., USA
{preet_derasari,kailashg26,guruv}@gwu.edu

***Abstract*—Rapid rise in malware attacks has added significant costs to cyber operations. As adversaries evolve, there is a growing need for fast, targeted defenses that effectively guard computer systems against these cyber-attacks. Cyber-deception is an increasingly adopted defense strategy with its ability to continually engage with adversaries and deploy counter-measures proactively by manipulating the malware program execution flow to non-useful states for the attacker.**

This paper introduces MAYALOK, a novel hardware-based cyber-deception framework to combat malware through runtime instruction infusion. MAYALOK employs hardware deception primitives to transparently insert or skip malware program instructions during runtime and deliver the attackers a deceptive view of the system state. We evaluate and demonstrate the deception efficacy of the MAYALOK framework on malware samples representing various attack vectors: Ransomware, InfoStealers, Buffer overflow, and Side-channels.

***Index Terms*—Computer Security, Cyber-Deception, Hardware Security**

I. INTRODUCTION

Newer security threats have been steadily rising, fueled partly by the growing complexity of the computing environments and the increased connectivity between them. To safeguard the computer system, popular defense strategies typically rely on either detect-and-defend type solutions [4] or system security-hardening frameworks [14]. Despite the successes of these prior solutions against many attacks, it remains practically infeasible to accurately predict all of the threat vectors and prepare defenses *ahead in time* to act against adversaries and effectively protect the critical system assets.

Cyber-deception is a proactive defense strategy that not only traps malicious actors but creates an illusion of successful penetration for the attackers, while *misleading* them with false information and *learning* their behavior [2]. Prior studies [25] have shown its strong potential for future-generation system security even in zero-trust architectures [18]. It maintains deep visibility into the activities and interactions between applications and users (via program instrumentation or system activity profiling). It can also proactively break adversarial system intrusion attempts through the dynamic deployment of deceptive decoys (that drain the adversary's resources).

Current software-based cyber-deception leverages honeypot techniques to trap an attacker with counterfeit information using specialized honey servers and observe its interactions with the honeypots [17]. Similar software cyber-deception techniques are vulnerable to advanced attackers that can perform signature analysis and surveillance for creating binaries to subvert such deception frameworks [13]. As software layers (including the OS and runtime) become active adversarial targets, support for cyber-deception from trusted hardware layers is necessary to boost its effectiveness and enhance adoption among the cyber-protection systems. As such, there remains an urgent need to investigate mechanisms that support cyber-deception within the hardware layers of the computer system to handle evolving adversaries effectively.

In this paper, we introduce **MAYALOK**[1], a cyber-deception hardware to enable the deployment of runtime instruction infusion against a wide variety of security attacks. Since hardware directly supports deception, the MAYALOK cyber-deception operations are transparent (not visible) to even the OS kernel during runtime. Our hardware cyber-deception helps achieve multiple goals simultaneously: Provide an illusion to the attacker that it successfully penetrated the system while effectively enhancing the system monitor's ability to silently observe malicious actions without provoking them.

To evaluate MAYALOK, we studied malware samples representing different attack vectors: Ransomware, InfoStealers, Buffer overflows, and Side-channels. Our experimental analysis demonstrates the efficacy of MAYALOK in achieving the same level of security with significantly lower (negligible) performance overheads when compared to state-of-the-art Moving Target Defense designs like Morpheus [9].

In summary, the key contributions of our paper are:

- We present MAYALOK, a cyber-deception hardware that *transparently* infuses the malware instructions during program runtime, impeding an adversary's ability to adapt to the dynamically deployed cyber-deception strategy.
- We design and demonstrate our deception infusion logic that can dynamically alter the system view of the program state to an adversary during its operation. We note that our deception infusion logic may be used individually or in composition with other defenses to create deception strategies that can ultimately thwart several malware vectors.
- We show experimental analysis of MAYALOK's effectiveness in executing deception strategies using four case studies, including a ransomware [1], a buffer overflow benchmark [23], an InfoStealer [21], and a timing side-channel [24]. We also evaluated the performance impact of the MAYALOK hardware and observed that it incurs neg-

[1]*Mayalok* translates to an illusionary place in the Sanskrit language.

DOI 10.1109/ASAP57973.2023.00019

ligible performance overheads due to the targeted defense strategy on malware.

II. Threat Model and Assumptions

This work considers attacks targeting memory corruption and information leakage through timing side-channels. Notable examples of such attacks include manipulating file storage, exploiting memory vulnerabilities in legacy programs to hijack their control flow, stealing sensitive information, and creating covert communication channels. The attackers usually employ malware like ransomware, spyware, Remote Access Trojans (RATs), remote code executions, and others that entail executing malicious code on a victim system. We target the core functionalities of such malware to mount our deception primitives. For example, ransomware cannot successfully encrypt files without performing repetitive reads/writes on user files.

We make the following assumptions about the attackers: *1)* they have sufficient privileges on the system that allows them to access and modify the directories and their files, *2)* they can locate memory addresses inside the victim programs through address reverse mapping or disclosure vulnerabilities, and *3)* they can measure the timing of specific code regions with reasonable accuracy. These capabilities allow the adversaries to effectively intrude into the system and launch information leakage or memory data corruption attacks.

We also assume that MAYALOK can leverage existing zero-day malware detection frameworks similar to the one proposed by Zolotukhin et al. [27] that use n-gram models to extract opcode sequences from known malware and benign applications. Another body of research has shown the effectiveness of DNN-based malware detection techniques using features like raw machine byte values to analyze un-trusted applications [22]. These prior works report above 97% accuracy in classifying malware and may be used before program load-time to formulate a deception strategy. The MAYALOK hardware is invoked on binaries marked as malicious by such highly accurate detectors. Any false positives during this process can be attributed to the malware detector's capability.

III. The Mayalok Deception Hardware

In this section, we describe the design of our MAYALOK deception hardware, including modifications needed in the processor pipeline. MAYALOK realizes deception on malicious instructions with the help of *Hardware Deception Cache* and *Deception Trigger* that tag the deception instructions (described in Section III-A), and the *Deception Infusion Logic* that modifies the tagged instructions (described in Section III-B). Figure 1 presents an overview of our architecture that shows the additional hardware required by our framework, along with their data/control and signal paths integrated into an out-of-order processor pipeline.

A. Deception Tagging

Deception tagging aims to trigger deception-related operations on specific instructions and minimize the impact on all other instructions. We illustrate our design on an x86-like core (widely used and targeted by most malware). We note that without loss of generality, a similar or a subset of hardware instrumentation features of our proposed design may also be adapted to other microarchitectures. The MAYALOK deception tagging involves interaction between the following structures:

1) Hardware Deception Table/Cache: Our design includes a *Deception Table* that stores the parameters needed by the MAYALOK primitives (referred to as deception parameters or $\mathrm{D_{params}}$ for short) during program execution in secure memory locations accessible only to trusted system software (e.g., system hypervisor). An illustration is shown in Figure 2a.

When a potentially spurious process is loaded, MAYALOK assumes that a system hypervisor uses a program load-time utility (e.g., LD_PRELOAD) to include fake (honey) code and data into the malware memory space (including those from dynamically linked libraries). Previous works [7] have employed disassemblers to find potentially malicious instructions using a pre-defined set of rules (for example, opcodes with memory accesses) and output that information to a user-defined structure. We use this mechanism to populate the deception table entries and initialize a cache block deception bitmap (Blk D-bit) in hardware to tag the cache blocks containing instruction addresses specified in the deception table.

Frequent access to the deception table may present prohibitive runtime overheads. To alleviate this problem, we incorporate a *Hardware Deception Cache* for the deception table that acts as a fast access mechanism for $\mathrm{D_{params}}$ of instructions frequently targeted by the MAYALOK hardware. This cache resembles a Translation Lookaside Buffer that maintains frequently accessed page table entries. Program counter (PC) addresses are currently used to tag deception table entries. We note that other instruction attributes, such as the data (memory) addresses, instruction operation types (opcodes), or target/source registers, may also be used as tags in future deception table designs.

To launch the MAYALOK framework against a malware process, the deception table (managed by a system hypervisor) is initialized with three sets of fields as shown in Fig. 2a: 1) the instruction addresses (<$\mathrm{D_{PC}}$>) of targeted instructions, 2) the deception mode (<mode>) to select the infusion functionality, which also acts as a format specifier for 3) the deception parameters (<$\mathrm{D_{params}}$>) required by the infusion logic. At decode time, our deception table facilitates two operational modes:

Insert:(01) Insert the *n* deception-related instructions ($\mathrm{D-inst_n}$) into the pipeline before the specified instruction address ($\mathrm{D_{PC}}$).

Skip:(10) Remove instructions from the pipeline starting at a specified instruction address until the number of instructions mentioned in $\mathrm{D_{params}}$ as $\mathrm{N_{inst}}$.

2) Deception Trigger: When a malicious application is executing on a MAYALOK-enabled cyber-deception framework, the *Deception Trigger* mechanism gates the instructions being fetched and decoded from the I-cache based on the Blk D-bit value of their source cache block. A Blk D-bit value of

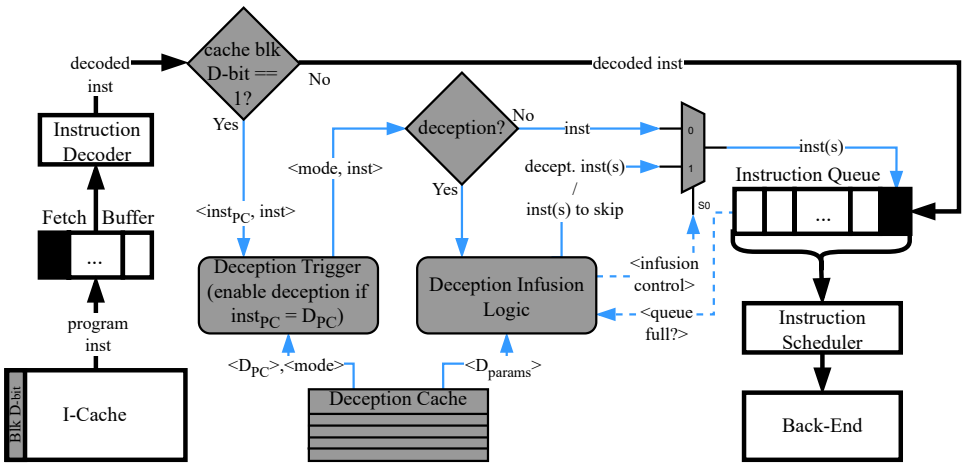
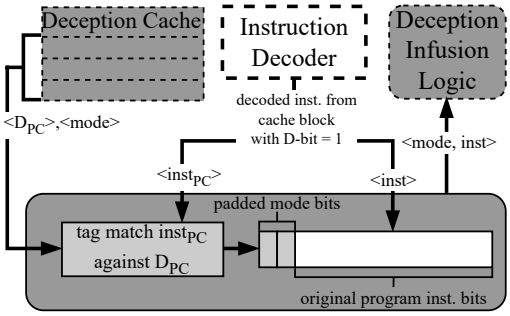


Fig. 1: The MAYALOK design integrated with an out-of-order processor pipeline. Deception-related hardware is highlighted in gray, which includes the *Deception Infusion Logic* along with the *Deception Trigger* mechanism. The *Deception Cache* supplies the <D_{PC}>, <D_{params}> values to these hardware units.

Deception Infusion Functionality	**Format Specifier (<D_{PC}>,<mode>,<D_{params}>)**
Insert	<PC>,<01>,<D-inst$_n$>
Skip	<PC>,<10>,<N$_{inst}$>

(a) Format specifiers for deception table/cache entries



(b) Deception trigger operation

Fig. 2: (a) The format specifiers for the deception table/cache entries indicating the <D_{PC}> tag, the target instruction's <mode> of deception, and the <D_{params}> needed to deceive it. (b) The *Deception Trigger* operation which pads a program instruction with the mode bits of a deception cache entry if the entry's <D_{PC}> tag matches with the <$inst_{PC}$>.

'1' informs the deception trigger that the corresponding cache block is occupied by instruction(s) requiring deception. The hardware sets the Blk D-bit values for instructions critical to the malware in achieving an attacker's goals. Binary analysis tools like Angr [20] are used pre-runtime to reveal a malware's vital code blocks. We note that the Blk D-bit optimization helps reduce the deception trigger's tag lookups to instructions within the cache blocks that have the Blk D-bit enabled. Thus, most instructions may bypass the deception trigger and execute normally down the pipeline.

Figure 2b represents the deception trigger's operations. Upon encountering an instruction block decoded with the D-bit on, the deception trigger logic matches the PC values of each instruction in the block (<$inst_{PC}$>) against the deception cache entries that are indexed using a target instruction's PC value (<D_{PC}>). Upon a tag match, the deception trigger pads the program instructions (<inst>) with the mode bits (<mode>) to guide the deception infusion logic. The default value is 00, indicating no deception is required for that instruction. From the evaluation presented in Section IV, we observed that $< 1\%$ of the total instructions on average trigger deception due to the targeted nature of MAYALOK. Most instructions do not invoke the deception infusion logic, thus reducing the overall execution time impact.

B. Deception Infusion Logic

The MAYALOK *Deception Infusion Logic* is incorporated into the front end of the pipeline to seamlessly integrate its functions before the instructions reach the instruction scheduler. DISE [6] has shown the feasibility of dynamically editing instruction streams in the front-end stages of the pipeline.

Upon encountering a program instruction tagged for an **insert (01)** mode of deception, the infusion logic invokes an exception-like mechanism to save contextual information (block address, PC address, size, etc.) about the existing instructions in the pipeline. Subsequently, it begins inserting

the pre-decoded deception instructions ($\mathrm{D-inst_n}$ supplied by $\mathrm{D_{params}}$) into the Instruction Queue (IQ).

In CISC pipelines (e.g., x86), each instruction may decode to a variable number of *micro-op instructions*. This irregularity necessitates appropriate handling of situations where the infusion logic is in the middle of inserting a deception instruction sequence, and the IQ is already full. To accommodate such cases, we added a `<queue full?>` signal from the IQ to the infusion primitive as seen in Figure 1, which *pauses* the insertion of deception instructions via the `<infusion control>` signal to the fetch stage. The insertion is *resumed* in the following clock cycles when the back-end pipeline stages drain the deception micro-ops after execution and *stopped* when all deception instructions are infused into the IQ to achieve the necessary deception. Finally, the saved context is restored, and the program and infusion-tagged instructions continue execution. This pause-resume mechanism is also leveraged for precise exception handling and during interrupts.

For program instructions that are tagged in **skip (10)** mode, the infusion logic accesses the deception cache as seen in figure 1 to look up the number of instructions ($\mathrm{N_{inst}}$ from $\mathrm{D_{params}}$) to be skipped, starting from the PC address ($\mathrm{D_{PC}}$). The instruction skip logic nullifies the marked instructions from the IQ until $\mathrm{N_{inst}}$. Like instruction insertion, the skip operations are also frozen when the IQ is empty and resumed when the fetch logic fills the instruction buffer.

IV. Experimental Results

We conducted experiments on the x86 build of Gem5 version 21.0.0.0 [3]. The MAYALOK deception framework is built on top of the existing Gem5 DerivO3 CPU model based on an out-of-order processor. All of our simulations are single process context threads execution of Gem5 configured with a 2GHz single core CPU, an L1 I-cache of 32KB, an L1 D-cache of 64KB, both 2-way set associative, and a 2MB 8-way set associative, inclusive L2 cache.

A. Security Analysis – Case Studies

To evaluate the efficacy of MAYALOK, four malware samples were selected among some of the major contemporary cyber-security attacks. We formulate cyber-deception strategies that help thwart these attacks and assume system defenders to do the same in a real hardware implementation. Our test-bed consists of one ransomware [1], a buffer overflow benchmark [23], an InfoStealer [21], and a timing-based side-channel attack on the RSA cryptography algorithm [24]. To present functional examples of our deception primitives, we used them in *three* deception strategies:

- *Diversion* involves redirecting the attacker away from sensitive to honey resources.
- *Exhaustion* is achieved by pinning down the attacker in an infinite loop and draining their performance and energy right after they strike a well-crafted honey resource.
- *Fabrication* entails misrepresenting certain sensitive information that the attacker desires by modifying their perception of the victim system.

Malware 1 RSA Timing Attack

```
d is the private decryption key
N is the product of two primes
C is the ciphertext to decrypt
procedure MODULAR_EXPONENTIATION(d, C, N)
    n ← size of d in bits; x ← C; j ← 1
    for each j less than n do
        x ← x^2 mod N
        if d_j equals 1 then
            x ← x * C mod N
        end if
        if d_j equals 0 then
            x * C mod N
        end if
        j ← j + 1
    end for
    return x
end procedure
```

Certain deception strategies can be applied to malware categories that share common interaction patterns with the victim system. For example, ransomware and InfoStealers can be categorized as malware that manipulate filesystems. Similar consolidations would reduce the pre-load-time analysis complexity in creating deception strategies.

The experimental results for security evaluation are shown in Table I that list the different *malware families*, their *attack categories*, *deception strategies*, and the *infusion mode* applied. We also report the penalties to the attacker and the outcomes of our security evaluation. *Attacker penalty* indicates the overhead impact on a malicious program when the MAYALOK hardware deceives them. *Security outcomes* show the sensitive assets are protected by MAYALOK when targeted by the corresponding malware.

The following describes our case studies and how the MAYALOK hardware helped accomplish deception that thwarted the corresponding attacks.

*1) **Side-Channel – RSA Timing Attack**:* An *RSA Timing Attack* [24] works by siphoning the timing difference caused by a compute-intensive modular exponentiation algorithm used to encrypt/decrypt a message with the respective public/private keys in an RSA-based encryption scheme. The algorithm uses two prime numbers *p* and *q* to compute $N = p*q$ and selects two random exponents *e* and *d* that satisfy the condition $e*d = 1 \bmod (p-1)*(q-1)$. Then it uses a public key *(e,N)* to encrypt and a private key *(d)* to decrypt a message *M*. The encrypted message is stored in $C = M^e \bmod N$, where *C* is the resulting ciphertext.

Malware 1 performs a modular exponentiation function using the Chinese Remainder Theorem, where a ciphertext, *C*, is decrypted using the binary value of the private key *d*. The attack exploits a timing difference in the computation of $M = C^d \bmod N$, where $x = x^2 \bmod N$ is the only calculation done for *'0'* bits and due to the additional $x = x*C \bmod N$ for each *'1'* bit of *d* the attacker can precisely time this

TABLE I: Summary of the security evaluation of MAYALOK on four contemporary malware samples

Malware family	Attack category	Deception strategy	Infusion mode	Attacker penalty	Security outcomes
Credential Stealer	InfoStealer	Fabrication	Insert	Fake data supply	Credentials protected
Petya	Ransomware	Exhaustion	Insert & Skip	∞ (time)	Files protected
Return-To-Libc	Buffer overflow	Diversion	Insert	Failed overflow	Pointer protected
RSA Timing Attack	Side-channel	Fabrication	Insert	No side-channel	Keys protected

calculation using the `RDTSC` timer functionality. By analyzing the timing difference of multiple decryption operations of some carefully crafted ciphertexts, the attacker can retrieve the secret key value *d* with reasonably high accuracy.

We propose a *Fabrication* deception strategy to insert deception instructions that would perform a dummy version (with no architectural side-effects) of the missing multiplication and reduction ($x{*}C \bmod N$) operation for each *'0'* bit of the secret key *d* to equalize the execution time of this function for all of *d*'s bits. We would also need to discard the result so that it does not modify the x value. We obfuscated the timing difference of this attack by deploying the MAYALOK infusion logic to silently insert the deception instructions corresponding to the $x{*}C \bmod N$ operation into the fetch queue without disturbing the original program instructions.

The deception instructions are inserted at the end of the modular exponentiation loop by creating deception table entries specifying the PC address of the `JMP` instruction (<$\mathrm{D_{PC}}$>), and the <$\mathrm{D_{params}}$> that supply $\mathrm{D - inst_n}$ comma-separated hexadecimal machine byte values equivalent to 0xe0458b, 0x01c083, 0xd84589, 0xe4458b, 0xcc45af0f, 0x99, 0xc47df7, that will be infused into the instruction pipeline by the MAYALOK infusion logic in insert mode (01) to perform an additional $x * C \bmod N$ operation for every 0 bit in the RSA exponentiation.

To evaluate the effectiveness of *Fabrication* on the RSA timing attack, we added `clock_gettime()` function calls inside the modular exponentiation routine to measure the computational timing difference for each bit of the secret key *d*. To simulate an attacker guessing the private key, we executed the sample ten times with the same values of *p* and *q* but with different ciphertexts. Without MAYALOK, our timer reported the average execution time for the modular exponentiation function as $130ns$ with a difference of $40ns$ between 0 and 1 bits of the *d* value. Upon enabling MAYALOK with the $\mathrm{D_{params}}$ for infusion, we observed less than a $5ns$ timing difference for each bit with no apparent pattern.

In summary, our experiments demonstrate how MAYALOK can be helpful to *dynamically* insert noise (transparent to the attacker) using hardware that can protect valuable secrets (private decryption keys) in the widely used RSA algorithm without the need to modify the source code or instrument the binary application files.

2) ***Ransomware – Petya****:* The *Petya* [1] ransomware family targets a victim's Master Boot Record (MBR) and encrypts all of the files. Its evaluation sample opens a user-specified directory and reads the entries to find unencrypted files. Each target file is duplicated with a custom extension, and the original files are deleted after their contents are encrypted (using OpenSSL-EVP) and stored in the duplicates. This ransomware sample repeats the directory read process until it observes complete encryption of previously missed or newly added files.

Malware 2 Ransomware (Petya)

```
procedure ENCRYPT_FILES(target_dir)
    for each file in target_dir do
        if not file.encrypted then
            fileptr ← fopen(file)
            fread(fileptr, buf, sizeof(buf))
            buf ← encrypted buf
            encfileptr ← create file.encrypted
            fwrite(encfileptr, buf, sizeof(buf))
            remove(file)
        end if
    end for
    for each file in target_dir do
        if not all file.encrypted then
            encrypt_files(target_dir)
        end if
    end for
end procedure
```

We propose an *Exhaustion* strategy for this sample; (1) redirect sensitive file-opens to a honey file and (2) prevent file removals which trap it in an infinite loop. The ransomware's `encrypt_files()` function opens the target directory, and for each unencrypted `file`, its contents are encrypted and written to their copies in `file.encrypted`. The original files are then deleted by calling `remove(file)`. To ensure the encryption of missed or newly added files, the ransomware reads `target_dir`'s entries again and calls `encrypt_files()` if it finds any unencrypted files. To achieve the Exhaustion deception strategy, we need to:

- Insert instructions to modify the `open(file)` calls for all files inside `target_dir` such that the target file names are switched with a honey file before they are opened.
- Skip the `remove(file)` function calls for the honey file to trap the ransomware in a repeated encryption cycle every time it reads the directory and finds that all target files along with the honey file are unencrypted.

For our *first* deception target, among all instructions formulated by the compiler for an `fopen()` call, `MOVUPS xmm2/mem128, xmm1` was the instruction responsible for moving the file name values to program memory locations.

Particularly, the instruction moves 128 bits of a single-precision floating-point data value from the `xmm1` register to either the `xmm2` register or a 128-bit memory location (`mem128`) [11]. The `MOVUPS xmm2/mem128, xmm1` instructions were decoded into two `STFP mem64, xmm1` micro-ops by our Gem5 simulated x86 core. The data values held by the `xmm1` registers of the repetitively executed `MOVUPS` instructions are altered via deception to store a honey file name instead of the ransomware-targeted files. Deception is executed for all targeted `MOVUPS` instructions (<D_{PC}>) by creating deception table entries in an insert mode (01) of infusion with <D_{params}> that supply $D - inst_n$ instructions to switch the original file name bytes with that of a honey file within the `xmm1` registers.

For our *second* deception target, we observed that during a `remove(file)` function call, before the `CALLQ <addr@remove>` instruction, the compiler emitted additional `MOV` instructions that moved the `file` to function argument registers. Deception is executed by creating deception table entries in a skip mode (10) of infusion for all target `CALLQ` instructions. Since infusion is implemented at the front end of the pipeline, MAYALOK can prevent any deception-tagged instruction(s) from being inserted into the instruction queue.

To evaluate the efficacy of an exhaustion-based deception strategy on this sample, we tested a directory with over a hundred files as a ransomware target. It finished execution with deception OFF within $200ms$. A honey file was placed in the same directory, and the deception table was initialized with the entries required for infusing specific `MOVUPS` and `CALLQ` instructions. The sample was executed again with deception turned ON. The ransomware did not exit execution since it was trapped in an infinite loop, ending up encrypting the honey files repeatedly for all of the files in the target directory. Our observations conclude that MAYALOK can perform an *Exhaustion*-based deception strategy by trapping an adversary for ∞ while draining their resources (time) and protecting the valuable assets (sensitive files within the directory).

3) ***Return-to-Libc – Buffer Overflow****:* We use the *RIPE Buffer Overflow Benchmark* [23] to evaluate a return-to-libc attack that exploits stack buffer overflow to invoke the `system()` function and launch an unauthorized command. We build a test sample according to RIPE's configuration directives that allow a vulnerable code to perform buffer overflow on itself. The resulting sample tries to overwrite a function pointer on the stack by abusing the `memcpy()` function's flaw to move contents between two buffers without performing bounds check at the destination buffer. We propose a *Diversion* deception strategy that can thwart this adversary by switching the address value of its malicious code location with a honey address during the movement of bytes within `memcpy()` from a payload to the unsafe stack buffer.

From the buffer overflow example in Malware 3 it can be seen that the sample misuses `memcpy(buffer, payload, size)` in an attempt to overwrite the target address, `&some_function()` with the glibc address,

Malware 3 Return-to-Libc

```
procedure HONEY_FUNCTION(str)
    Perform redundant operations that confuse the attacker
end procedure
procedure PERFORM_ATTACK
    stack_func_ptr ← &some_function()
    initialize buffer[len]
    target_addr ← &stack_func_ptr
    overflow_ptr ← &system()
    size ← target_addr - buffer + 8
    payload ← ['A'*(size - 8)] + overflow_ptr
    memcpy(buffer, payload, size)
    call stack_func_ptr
end procedure
```

`&system()` that is stored in an `overflow_ptr`. The payload's `size` is calculated as the number of bytes required to fill the stack space between `buffer` and `target_addr` plus an additional eight bytes for the `overflow_ptr`. The attacker usually pads the `payload` with `size - 8` number of garbage bytes. Once `memcpy()` is called, the `overflow_ptr` is copied to the end of the payload, which overwrites the `target_addr`. The sample calls `stack_func_ptr`, which now holds the address of `system()` instead of `some_function()` after a successful overflow.

To deploy a diversion-based deception strategy, we modify the `overflow_ptr` bytes being moved inside `memcpy()` from the payload to the stack buffer. The move instructions inside `memcpy()` were realized through the `MOVUPS xmm2/mem128, xmm1` instructions that wrote `system()` function's address bytes to the `stack_func_ptr`. If the 8 bytes of attacker pointer value in `xmm1` are changed to the address of a `honey_function()` before the `MOVUPS` instruction is completed, the attacker will be diverted to the fake function, thus nullifying the overflow attempt. Our results from the *Petya* case study show that infusion in an insert mode of operation can effectively replace a byte sequence inside an `xmm1` register before they are stored in memory. For this case study, we formulated a similar deception table entry with a set of <D_{params}>, containing instructions to modify the `xmm1` register value with that of the `honey_function()`. In this example, the fake function invokes a dummy shell with the lowest level of privileges.

To evaluate the efficacy of our deception strategy, we measured the sample execution time for a return-to-libc attack when the deception was disabled. Then, we populated the deception table with appropriate parameters to re-execute the program with deception enabled. Our experiments on Gem5 showed that the sample took $0.1ms$ to successfully overwrite the `stack_func_ptr` with the glibc system address when MAYALOK was turned OFF. With MAYALOK turned ON, the sample could not invoke a shell using the `system()` call and thus resulted in a *failed overflow* attempt for the attacker by a simple yet effective usage of the infusion hardware.

Malware 4 Credential Stealer

```
initialize browser_pass ← "path/to/password/file"
if browser_pass exists then
    fopen(browser_pass,"r")
    fread(passwords from browser_pass)
    create new file browser_pass_copy
    fwrite(copied passwords to browser_pass_copy)
    compress browser_pass_copy
    exfiltrate browser_pass_copy to C2 server
end if
```

4) ***InfoStealer – Credential Stealer****:* We studied the *Credential Stealer* [21] malware in the InfoStealer family. This malware infiltrates a victim system and exfiltrates important user credentials like passwords, browser cookies, history, and encryption keys. The credential stealer sample used in this evaluation targets browser password files stored at predetermined locations inside victim systems. This stealer confirms the presence of the password file and reads all the information, which is duplicated to another file, compressed, and sent to a C2 (Command & Control) server. Prior works like SODA [19] have established HoneyThings (honey files, honey registries, honey configurations) to misinform InfoStealers as an effective deception scheme. Similarly, we formulated a *Fabrication* deception strategy to counter this class of malware. The goal is to misrepresent credential data from sensitive files when requested by the malware.

We demonstrate an example of the Malware 4, where this sample opens the browser password file that stores the user's passwords on the system. Since these files are read-only for user space applications, the attacker copies the data to a new file for ensuing compression (saves storage space) and transmission to the C2 server. The sample used in this evaluation calls `fread()` to extract `browser_pass` contents and writes them to another file, `browser_pass_copy`. To fabricate the data read by this sample, we needed to fake the password bytes moved `browser_pass` to a buffer inside the `fread()` function call.

Upon an instruction trace analysis of this sample, we found that the `MOVUPS xmm1, xmm2/mem128` instructions inside the `fread()` call are responsible for reading data bytes from a memory location (`mem128`) to the `xmm1` register. For a fabrication strategy, we used the MAYALOK infusion logic to insert memory-modifying instructions that fake the data stored at `mem128` before the corresponding `MOVUPS` instruction was executed. Similarly, multiple `MOVUPS` instructions emitted to read the entire password file were deceived using this strategy. The D_{param} required for each deception table entry contained $\mathrm{D-inst_n}$ instructions along with the PC, $\mathrm{D_{PC}}$ of each target `MOVUPS` instruction.

To study the effectiveness of deception in this sample, we executed the credential stealer without deception, where it could read a set of passwords from the browser user files and clone them. Upon re-executing the sample with deception enabled, we observed that it created a copy of the password file, but its values were completely different from the original version. The *Hamming Distance* was calculated between the two files as a percentage difference of bytes between them using $\%diff = (1 - \frac{\text{Hamming Distance}}{\text{File Size}}) * 100$. The result was a 100% change in the character bytes between the actual and duplicate password files. This proves that the *Fabrication* deception strategy effectively defends against an InfoStealer type of attack.

B. Overhead Analysis

With MAYALOK deception hardware being incorporated into the front end of a processor pipeline to infuse instructions, a small 2-way fully associative 2KB deception cache is used by our design to store the deception table with $\mathrm{D_{params}}$ for up to 20 individual $\mathrm{D_{PCs}}$. Since deception is a targeted defense strategy, we note that this small cache is sufficient for most of this work's malware case study applications. Cache spill/fill may be used with the instruction cache and the rest of the cache hierarchy for applications with more extensive deception code needs. Besides the deception cache, we note that the deception trigger and infusion logic incur negligible area overheads.

Comparison with related efforts – Morpheus [9] is a hardware-based Moving Target Defense (MTD) technique that proposes a frequently triggered churn-based approach that re-randomizes the program code, data, and pointers using a combination of displacement and encryption. Their performance analysis reported on the RIPE buffer overflow benchmark [23] showed an average of 13% overhead for a continuous churn period [9]. For the same benchmark, MAYALOK – inserted instructions to replace the attacker code pointer with a honey function's address – incurs negligible overhead ($< 1\%$) due to the targeted manipulation of data values for a limited number of instructions as compared to Morpheus, which needs to continuously displace and encrypt all of the code, data, and pointer values.

V. RELATED WORK

Most research in the cyber-deception field can be divided into two categories, the first kind of which targets network-based techniques like virtual honeypots to create a simulated environment or a virtual machine for an attacker to interact [17]. However, counter-attacks can potentially sense these deception strategies via artifact discovering techniques [16]. The second class of research focuses on active cyber-deception techniques at the system level to tackle memory corruption and information manipulation attacks. PhantomFS [5] proposed a decoy-file and kernel-based interface, which is futile if the attacker compromises the kernel altogether. Due to a lack of hardware support, software-based API-hooking enabled deception against ransomware, InfoStealers, and RATs typically report very high execution time overheads (2 minutes for Soda [19]) to detect and stop the malware.

Alternatively, hardware-based asset obfuscation tactics utilize moving target defense [9]. While effective, they either

require system-wide state changes (e.g., churns used by Morpheus [9]) or are designed to handle specific attacks [15]. Camouflage [26] presented a hardware solution to mitigate memory side channels using memory request/response rate modulation that may be detected by attackers auditing the memory bus for anomalies in memory traffic. Compared to these works, the MAYALOK cyber-deception hardware presents a dynamic and low overhead platform that provides flexibility to different malware types, as demonstrated by our security evaluation.

Cyber-deception has been studied from a behavioral aspect by Landsborough et al. [12] suggesting a strategic approach that combines autonomic computing, game theory, and cognitive and behavioral psychology to deploy decoys effectively. Additionally, Foreseer [10] introduced an LSTM-based framework to synthesize malware events that significantly decreases the detection time of front-end malware detectors by up to 40%. MAYAVI [8] proposed a memory load/store address modifying deception engine against malware as a parallel effort towards hardware-assisted cyber-deception.

VI. CONCLUSION

This paper presents MAYALOK, a cyber-deception supporting hardware. The MAYALOK hardware design incorporates a programmable table to store the deception-related parameters assisted using a hardware cache and a trigger logic to invoke deception for specific instructions in a flexible manner. We propose an infusion design integrated into the processor pipeline that can seamlessly *insert* pre-decoded instructions or *remove* program instructions from the malicious process' dynamic execution flow. We also illustrate MAYALOK's security effectiveness by evaluating it on various malware samples comprising memory corruption and information exfiltration attacks like Ransomware, Buffer overflows, InfoStealers, and Side-channels. Our evaluation shows that MAYALOK is capable of protecting sensitive assets and presents a high cost to the attacker. We envision MAYALOK as a next-generation design for a secure architecture that can effectively protect sensitive resources by increasing the attack cost through cleverly orchestrated deception strategies.

ACKNOWLEDGMENT

This research is based on work supported by the U.S. Office of Naval Research under grant N00014-21-1-2520.

REFERENCES

[1] J. Aidan, H. Verma, and L. Awasthi, "Comprehensive survey on petya ransomware attack," in *IEEE ICNGCIS*, 2017, pp. 122–125.

[2] E. Al-Shaer, J. Wei, W. Kevin, and C. Wang, "Autonomous cyber deception," *Springer*, 2019.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[4] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware," in *IEEE/ACM MICRO*, 2014, pp. 216–228.

[5] J. Choi, H. Lee, Y. Park, H. K. Kim, J. Lee, Y. Kim, G. Lee, S.-W. Shim, and T. Kim, "Phantomfs-v2: Dare you to avoid this trap," *IEEE Access*, vol. 8, pp. 198 285–198 300, 2020.

[6] M. L. Corliss, E. C. Lewis, and A. Roth, "Dise: A programmable macro engine for customizing applications," in *IEEE ISCA*, 2003, pp. 362–373.

[7] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *ACM ASIA CCS*, 2013, pp. 299–310.

[8] P. Derasari, K. Gogineni, and G. Venkataramani, "Mayavi: A cyber-deception hardware for memory load-stores," in *ACM GLSVLSI*, 2023.

[9] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco *et al.*, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *ACM ASPLOS*, 2019, pp. 469–484.

[10] K. Gogineni, P. Derasari, and G. Venkataramani, "Foreseer: Efficiently forecasting malware event series with long short-term memory," in *IEEE SEED*, 2022, pp. 97–108.

[11] Guide. (2022, Apr) Intel® 64 and ia-32 architectures software developer's manual. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[12] J. Landsborough, L. Carpenter, B. Coronado, S. Fugate, K. Ferguson-Walter, and D. Van Bruggen, "Towards self-adaptive cyber deception for defense." in *HICSS*, 2021, pp. 1–10.

[13] S. Morishita, T. Hoizumi, W. Ueno, R. Tanabe, C. Gañán, M. J. van Eeten, K. Yoshioka, and T. Matsumoto, "Detect me if you… oh wait. an internet-wide view of self-revealing honeypots," in *IFIP/IEEE IM*, 2019, pp. 134–143.

[14] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *ACM POMACS*, vol. 2, no. 2, pp. 1–30, 2018.

[15] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[16] K. Papazis and N. Chilamkurti, "Detecting indicators of deception in emulated monitoring systems," *Springer SOCA*, vol. 13, no. 1, pp. 17–29, 2019.

[17] M. T. Qassrawi and Z. Hongli, "Deception methodology in virtual honeypots," in *IEEE NSWCTC*, vol. 2, 2010, pp. 462–467.

[18] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," *NIST Report*, 2020.

[19] M. S. I. Sajid, J. Wei, B. Abdeen, E. Al-Shaer, M. M. Islam, W. Diong, and L. Khan, "Soda: A system for cyber deception orchestration and automation," in *ACM ACSAC*, 2021, pp. 675–689.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE S&P*, 2016.

[21] P. Tavares. (2021) Redline stealer malware: Full analysis. [Online]. Available: https://resources.infosecinstitute.com/topic/redline-stealer-malware-full-analysis/

[22] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46 717–46 738, 2019.

[23] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *ACM ACSAC*, 2011, pp. 41–50.

[24] W. H. Wong, "Timing attacks on rsa: revealing your secrets through the fourth dimension," *ACM XRDS*, vol. 11, no. 3, pp. 5–5, 2005.

[25] L. Zhang and V. L. Thing, "Three decades of deception techniques in active cyber defense-retrospect and outlook," *Elsevier Computers & Security*, vol. 106, p. 102288, 2021.

[26] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, "Camouflage: Memory traffic shaping to mitigate timing attacks," in *IEEE HPCA*, 2017, pp. 337–348.

[27] M. Zolotukhin and T. Hämäläinen, "Detection of zero-day malware based on the analysis of opcode sequences," in *IEEE CCNC*, 2014, pp. 386–391.