


# PDIFT: A Practical Dynamic Information-Flow Tracker

Michael Kiperberg<sup>1</sup> <sup>a</sup>, Aleksei Rozman<sup>1</sup>, Aleksei Kuraev<sup>1</sup> and Nezer Zaidenberg<sup>2</sup>

<sup>1</sup>*Department of Software Engineering, Shamoon College of Engineering, Beer-Sheva, Israel*

<sup>2</sup>*Department of Computer Sciences, Ariel University, Ariel, Israel*

Keywords: Virtualization, Hypervisor, Emulator, DIFT.

Abstract: We present PDIFT, a hybrid dynamic information-flow tracker. PDIFT is based on a thin hypervisor, which tracks information in coarse granularity but has a negligible performance overhead. PDIFT switches to its internally embedded emulator when the information is accessed and needs to be tracked with finer granularity. Using the combination of a thin hypervisor with an embedded emulator, we achieved a significant improvement in the overall performance. We believe that PDIFT can be used as an extension to the Android base DIFT systems that currently struggle with native code tracking.

## 1 INTRODUCTION

Computer systems are exposed to constant threats targeting their confidential information. An attacker can use multiple vectors to achieve her goal, from introducing new code that alters the system's behavior to the exfiltration of confidential information. While widely-available antivirus programs concentrate on software analysis to detect malicious intent, sufficiently sophisticated malware (e.g., metamorphic (You and Yim, 2010)) can evade them. On the other hand, data leakage prevention systems attempt to prevent unauthorized data movement by analyzing the data being transferred through certain critical junctions, e.g., between applications, over the network, or external flash drives. They operate by preventing the transfer of data that violates a predefined set of rules. The rules can target the context, or the content of the data (Kiperberg, 2021). These security systems attempt to prevent the infiltration of malware and the exfiltration of sensitive information.


Unfortunately, the widely available security systems are unaware of the history of the information being analyzed and, in particular, its origin. The origin can form the basis for simple and precise rules regarding data movement. For example, execution permissions can be expressed in terms of the code origin: only code that was downloaded from a specific domain can execute on the computer. Data leakage prevention rules can be expressed similarly: data that is

affected by the content of a sensitive file cannot be sent over the network. Of course, additional rules may apply. Without the origin, it may be challenging to devise precise rules for the prevention of data infiltration and exfiltration.

Dynamic information-flow tracking (DIFT) (Chen et al., 2021) is a technique for tracing data throughout any transformations. DIFT can establish the origin of a particular value, or more precisely, DIFT can reason about whether a particular memory location has been affected by another memory location. DIFT can be implemented in hardware (Chen et al., 2008; Venkataramani et al., 2008) and software. Software implementations, implemented using emulators (Yan and Yin, 2012; Xue et al., 2018) or instrumentation (Kemerlis et al., 2012), degrade the performance by several factors. The severe performance degradation prohibits DIFT deployments on commodity systems.

The only area in which DIFT can be partially deployed is Android. Since much of the code executing on Android is written in Java, DIFT can be implemented efficiently by analyzing the Java bytecode inside the Java Virtual Machine (Dalvik (Bornstein, 2008)). Unfortunately, the tracing functionality of such DIFT implementations cannot cross the Java boundary. Tracing of data in the native code invoked by Java requires additional means (Yan and Yin, 2012; Xue et al., 2018) implemented using the slow emulation or instrumentation.

The realization that DIFT can be enabled selectively, only during the times that the information is tracked, led to a hybrid DIFT approach (Ho et al.,

<sup>a</sup>  <https://orcid.org/0000-0001-8906-5940>

2006). A hybrid DIFT consists of a hypervisor and an emulator. The hypervisor performs virtualization until the system accesses the information to be tracked. Then, the hypervisor switches to emulation, which continues while the analyzed code manipulates the tracked information.

This paper introduces Practical Dynamic Information-Flow Tracker (PDIFT), a hybrid DIFT system. PDIFT's design makes it suitable for deployment on Android devices supporting virtualization. We anticipate the performance of DIFT on these devices to be optimal since direct emulation is required only when native code manipulates tracked information. In other cases, when Java code runs or when native code does not manipulate the tracked information, the performance degradation will be minimal.

PDIFT is a thin hypervisor (Shinagawa et al., 2009) with an embedded emulator. The hypervisor provides hypercalls for tainting the memory regions to be tracked and for querying whether a memory region is tainted. The hypervisor remains passive while the tainted memory region is not accessed, thus allowing it to incur only a negligible overhead (1.5%). When the tracked program accesses a tainted memory region, the hypervisor activates its embedded emulator. The performance benefits of embedding the emulator inside the hypervisor rather than placing it in a different environment (virtual machine) are twofold. First, we can spare the extra transition from the hypervisor to the different environment. Second, we can spare the costly copying of the tracked program's memory since the emulator can access it directly.

In this paper, we make the following contributions:

- We describe PDIFT, a novel DIFT system that is based on a thin hypervisor and an embedded emulator.
- We analyze the performance of the PDIFT and compare it with other DIFT systems.
- We outline the design of an efficient DIFT system for Android devices which supports both Java and native code.

## 2 BACKGROUND

Intel introduced hardware-assisted virtualization almost 20 years ago. Their VT-x technology introduces a new execution mode to the x86 architecture, the *root mode*. Software executing in the root mode, the hypervisor, has higher privileges than the operating system, has higher precedence in interrupt handling,

and can intercept many events in the operating system. Upon an occurrence of an event, the processor switches to the hypervisor. The hypervisor handles the event and continues the execution of the operating system. The transitions to the hypervisor are called *vm-exits*, and the transitions to the operating system are called *vm-entries*. The hypervisor can run multiple operating systems in isolated environments, called *virtual machines* or *guests*.

In order to improve the memory management performance, Intel introduced the Extended Page Table (EPT) mechanism, also known as a secondary-level page table (SLAT), which allows the hypervisor to configure the correspondence between the physical addresses as perceived by the guests to the actual physical addresses. Similarly to the regular (process) page table, the EPT configures the mapping and access rights. When a guest attempts to access a memory location in violation of the set rights, a VM-exit occurs, thus allowing the hypervisor to handle the event.

Hypervisors can run atop an operating system (e.g., Oracle VirtualBox (Oracle, 2022), VMware Workstation (VMware, 2022b), KVM (Deshane et al., 2008)) or without a hosting operating system (e.g., VMware ESXi (VMware, 2022a), Xen (Deshane et al., 2008)). In the second case, the hypervisor must boot directly from the firmware, similarly to the operating system itself. PDIFT is embedded in an EFI (Zimmer, 2009) application. The EFI firmware loads this application, which in turn initializes the hypervisor. When the EFI application exits, the firmware continues to the next EFI application, the operating system bootloader.

## 3 SYSTEM DESIGN

PDIFT is a hybrid dynamic information-flow tracking system. Figure 1 presents the high-level design of PDIFT. In essence, PDIFT provides means for determining whether a particular memory region was affected by another memory region during a particular calculation. The main idea is to augment the instruction-set architecture with the ability to track an information flow.

We have augmented the instruction-set architecture as follows. For every memory address (byte), we have added a bit to determine whether the memory address contains information that needs tracking. We say that such memory addresses are tainted. The process of information flow tracking can be divided into three phases. In the first phase, the input memory addresses are tainted. In the second phase, the calcu-

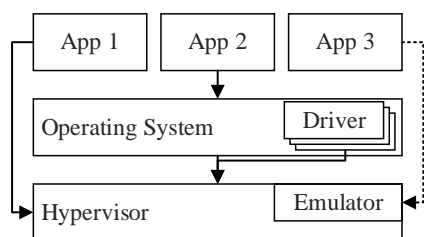


Figure 1: The design of PDIFT. The hypervisor executes with higher privileges than the operating system. The hypervisor executes a single virtual machine containing the operating system and the user applications. The applications can communicate with the operating system through system calls (the arrow from App 2 to the Operating System). The applications, the operating system, and the drivers can communicate with the hypervisor through hypercalls (all other solid arrows). The emulator is embedded in the hypervisor and can execute (emulate) application code.

lation is performed in the augmented instruction-set architecture. Finally, in the third phase, the tainted memory addresses are determined.

The main idea is to track the taints in the granularity of a single instruction. For this to work correctly, not only memory locations need to be taintable, but also the registers. For example, a `MOV` instruction can copy 4 bytes of information from memory to a register and then copy from this register to another memory address. In this case, if the source address is tainted, then the first instruction will propagate the taint to the destination register, and the second instruction will propagate the taint to the destination memory address.

In general, augmentations of existing instruction-set architectures are performed either by direct hardware modifications or by modifying an existing emulator of the instruction-set architecture. We have chosen the second path since it is more readily available. There are multiple available emulators (Bochs (Lawton, 1996), QEMU (Bellard, 2005), *libx86emu* (wfeldt, 2022)) that differ in performance, complexity, and the completeness of the emulated instruction-set architecture. However, they all are orders of magnitude slower than virtualization-based solutions.

On the other hand, virtualization-based solutions are incapable of modifying the behavior of instructions since those are executed directly by the CPU. Therefore, there is a need for a hybrid solution in which virtualization will be used most of the time in order to provide the best possible performance, and emulation will be used sporadically for taint propagation.

Our system is based on our thin hypervisor, which boots from a UEFI application before the operating system. The hypervisor creates a single virtual machine, also called the “guest”, under the hypervisor’s

full control — the operating system boots inside the newly created virtual machine.

Our hypervisor configures the EPT with the access rights of different pages, thus allowing it to intercept accesses to certain memory regions. In addition, the hypervisor provides communication means with the guest through hypercalls, similar to the communication means provided by operating systems. Upon an occurrence of an event that was selected for the interception, a hypervisor can observe the guest’s state, modify it and resume the guest’s execution.

We have added two new hypercalls to our hypervisor:

- `set_taint(addr, size, val)` — the hypercall sets or clears a taint (depending on the setting of the “val” parameter) of a memory region defined by the “addr” and “size” parameters,
- `get_taint(addr, size)` — the hypercall returns true iff at least one of the bytes of a memory region defined by the “addr” and “size” parameters is tainted.

These hypercalls allow an entity (kernel module or user-mode application) to set and check the tainting of a particular memory region. These hypercalls alter and query the taint repository, a data structure that keeps track of the tainted memory regions. In addition to modifying the taint repository, the `set_taint` hypercall configures the EPT to intercept accesses to the pages containing the tainted range of addresses.

When the hypervisor gains control in response to access to such a page, it begins the emulation process. The emulation involves copying the guest’s state to the emulator, running the emulator until the emulation can be disabled, and then copying the emulator’s state to the guest. The emulator executes inside the context of the hypervisor in order to eliminate unnecessary transitions. Therefore, memory accesses require the emulator to translate the guest’s virtual addresses to the actual physical addresses. In our current implementation, we do not cache the translation information. Such a cache can potentially have a significant effect on the overall system performance. During emulation, taints propagate from the memory to the registers and vice versa. The emulation proceeds while at least one register is tainted.

The current implementation of PDIFT does not include any optimizations to prevent costly transitions to the emulator and back. Neither does the current implementation include caches of compiled blocks of instructions. PDIFT includes of compiled *libx86emu*, a tiny x86 user-space emulator (wfeldt, 2022). Since *libx86emu*, unlike Bochs (Lawton, 1996) or Qemu (Bellard, 2005), does not have external dependencies, it could be easily embedded in our thin hypervisor.

On the other hand, *libx86emu* performs naïve emulation and does not incorporate even basic optimization techniques, which makes its performance suboptimal, to say the least.

The *libx86emu* emulator consists of an infinite loop. On each iteration, the emulator reads the next instruction from the location pointed by the instruction pointer. Then, the read instruction is parsed and executed. The instruction execution may involve access to the registers and the memory. The emulator stores the registers in a special object that describes the current execution context. The context object is allocated during the initialization of the hypervisor. We have extended this object to include a taint bit for each general-purpose register (see Figure 2). When the emulation process begins, the values of the actual registers are loaded to the context object, and the taint bits are cleared. When the emulation ends, the values of the registers are copied back.

```
typedef struct
{
    union {
        I32_reg_t I32_reg;
        I16_reg_t I16_reg;
        I8_reg_t I8_reg;
    } val;
    int tainted; // added
} i386_general_register;
```

Figure 2: A data structure that describes a single general-purpose register with the added “tainted” member.

The *libx86emu* emulator emulates each instruction by a dedicated function. In rare cases, several instructions are handled by a single function. Some instructions do not modify the taints of memory regions or registers; others do. We have modified the emulation of those instructions to propagate the taint as appropriate. Figure 3 presents an excerpt from the *x86emuOp\_mov\_word\_R\_RM* instruction emulation; only the handler of the *MOV reg, [mem]* variant is presented. The taint propagation functionality is marked with a comment. In this example, the destination register is set to be tainted iff one of the bytes in the source memory region is tainted. The example demonstrates two of the four macros that implement the taint propagation functionality.

The taint propagation functionality is realized by four macros:

- `MEM_TYPE_GET(ADDR, SIZE)` — evaluates to `TRUE` if at least one of the memory bytes in the range `[ADDR, ADDR+SIZE)` is tainted.
- `MEM_TYPE_SET(ADDR, SIZE, VAL)` — depending on the value of `VAL`, sets (if 1) or resets (if

0) the taint of the memory bytes in the range `[ADDR, ADDR+SIZE)`.

- `REG_GET_MARK( REG )` — evaluates to `TRUE` if the register pointed by `REG` is tainted
- `REG_SET_MARK( REG, VAL )` — depending on the value of `VAL`, sets (if 1) or resets (if 0) the taint of the register `REG`. In addition, this macro modifies a special field of the context object that counts the number of currently tainted registers.

Figure 4 presents an excerpt from the *x86emuOp\_mov\_byte\_RM\_R* instruction emulation. It demonstrates the usage of the `MEM_TYPE_SET` and the `REG_GET_MARK` macros. The memory taint is set iff the source register has a taint.

## 4 EVALUATION

The evaluation of the proposed system was performed on a system whose configuration appears in Table 1. We have split the evaluation into three parts: (a) correctness, (b) hypervisor performance, (c) overall performance.

Table 1: Evaluation System Configuration.

CPU	Intel Core i7-8665U
Memory	16GB
Operating System	Windows 7 x64

We demonstrate the firm information-flow tracking abilities using an encryption program that uses the RC4 cipher for encryption. The program reads its two inputs from files: the encryption key and the plaintext. The program schedules the key, encrypts the plaintext, and stores the ciphertext in a file. We have added two hypercalls to the program. The first hypercall is issued right after the inputs are read. The hypercall requests the hypervisor to taint the memory location that stores the key. The second hypercall is issued before the output file is written. This hypercall queries whether the ciphertext is tainted.

In the future, we envision these hypercalls to be implemented by the operating system in response to read and write requests. In this implementation, the operating system can mark files (or their regions) tainted by extending the filesystem. After reading from a tainted file, the operating system can request the hypervisor to taint the memory locations containing the read data. When writing to a file, the operating system can check whether the memory being written is tainted and mark the filesystem appropriately.

We assessed our implementation’s correctness by performing the encryption on the publicly available



```
static void x86emuOp_mov_word_R_RM(x86emu_t *emu, u8 op1)
{
    ...
    fetch_decode_modrm(emu, &mod, &rh, &rl);
    ...
    if(MODE_DATA32) {
        dst32 = decode_rm_long_register(emu, rh);
        OP_DECODE(",");
        addr = decode_rm_address(emu, mod, rl);
        *dst32 = fetch_data_long(emu, addr);
        REG_SET_MARK(dst32, MEM_TYPE_GET(addr, 4)); // <- taint propagation
    }
    ...
}
```

Figure 3: An excerpt from the x86emuOp\_mov\_word.R\_RM instruction emulation.

```
static void x86emuOp_mov_byte_RM_R(x86emu_t *emu, u8 op1)
{
    ...
    addr = decode_rm_address(emu, mod, rl);
    OP_DECODE(",");
    src = decode_rm_byte_register(emu, rh);
    store_data_byte(emu, addr, *src);
    MEM_TYPE_SET(addr, 1, REG_GET_MARK(src)); // <- taint propagation
    ...
}
```

Figure 4: An excerpt from the x86emuOp\_mov\_byte.RM.R instruction emulation.

RC4 test vectors. As was explained, we tainted the encryption key before the encryption. Then, we verified that the ciphertext was correct and tainted.

We used the PCMark (Sibai, 2008) benchmarking tool to measure the performance overhead of our hypervisor and compare it to a full hypervisor. PCMark runs several applications belonging to various categories and outputs the score obtained in each category. We executed PCMark in three different configurations: (a) without a hypervisor, (b) with our thin hypervisor, (c) in a full hypervisor (we used the same operating system in the guest). From the results in table 2, we can see that the thin hypervisor is at least an order of magnitude faster than a full hypervisor. The reason for this is that PDIFT intercepts only EPT violations, hypercalls (implemented as CPUIDs), and other events that induce VM-exits unconditionally, while full hypervisors intercept all the interrupts.

To measure the performance of the tainting functionality, we executed the encryption program ten times with and without requesting the hypervisor to taint the encryption key. Figure 5 presents the results. On average, the execution time increased by 26%.

We continued to a more detailed analysis of the execution times. Table 3 presents the number of emulation transitions that were performed during each

Table 2: Performance degradation in percents with respect to the configuration without a hypervisor.

Category	Thin Hypervisor	VirtualBox
App start-up	2.29	44.92
Web browsing	2.66	32.75
Spreadsheet	1.18	39.91
Writing	1.54	42.66
Photo editing	-0.24	38.35
Video editing	1.77	28.02

Table 3: Transitions number and handling time.

Run	Transitions	Total Ticks
1	2071	410227
2	2085	414746
3	2069	412529
4	2022	403379
5	2095	416943
6	2070	418251
7	2009	401620
8	2020	402534
9	2055	410003
10	2088	427738

run. In addition, the table presents the total time that was spent in the hypervisor during these transitions. The time is given in CPU ticks. On average, the program performed 2058 transitions. The handling took

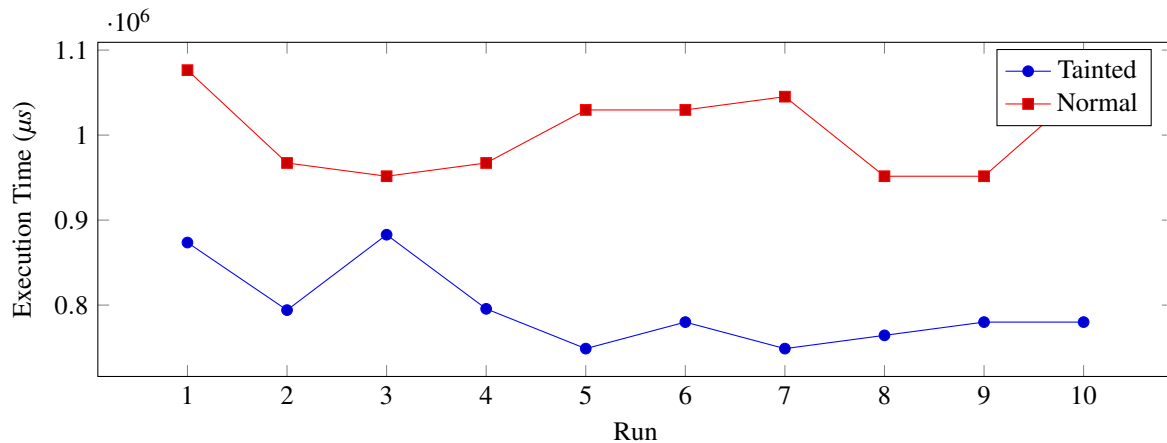


Figure 5: The execution times of the encryption program with and without tainting.

412K ticks on average, 200 ticks per transition. We note that depending on the CPU, the transition itself takes several hundreds of ticks. Therefore, we can conclude that PDIFT’s performance can be improved by reducing the number of transitions rather than improving the emulation performance.

## 5 RELATED WORK

While taint tracking can be performed statically, in native programs, it usually leads to tainting the entire program memory (Slowinska and Bos, 2009). However, it is still applicable to programs written in managed languages, like Java (Machiry, 2017). Dynamic taint tracking, which can be performed using instrumentation (Kemerlis et al., 2012), or emulation (Ho et al., 2006), suffers from high performance degradation, which is generally prohibitive for real-time security monitoring. Hardware solutions (Chen et al., 2008; Venkataramani et al., 2008) are costly and probably cannot be widely deployed. Another exciting direction is hybrid pruning, in which dynamically collected data is injected into the static taint analysis system to improve its precision (Das et al., 2022).

TaintDroid (Enck et al., 2014) is a widely adopted solution for DIFT in managed portions of Android applications. Attempts to extend this solution to native code, like DroidScope (Yan and Yin, 2012) and NDroid (Xue et al., 2018), which are based on emulation using QEMU, resulted in high performance degradation. We believe that better results can be obtained using the hybrid approach of PDIFT.

Ho et al. (Ho et al., 2006) was the first to propose the hybrid approach. Their implementation included a full hypervisor and an efficient emulator (Qemu). The emulator and the analyzed code were executed in two separate virtual machines, which degraded the perfor-

mance significantly. In contrast, PDIFT is based on a thin hypervisor with an embedded emulator.

Using hypervisors in security applications is not new. Full and thin hypervisors can be used in different scenarios ranging from malware detection (Seshadri et al., 2007; Leon et al., 2019) to device protection (Shinagawa et al., 2009; Kiperberg et al., 2020). PDIFT is another example of the thin hypervisors’ versatility.

## 6 FUTURE WORK

As we have stated previously, the performance of DIFT solutions is prohibitively high for native code and surprisingly low for managed code. Therefore, it would be interesting to integrate PDIFT into TaintDroid, for example. Such integration will require porting PDIFT to the ARM architecture and switching to a different emulator that supports ARM.

While initially, we thought that PDIFT could benefit from an optimized emulator, like QEMU, our evaluation clearly shows that the performance highly depends on the number of transitions. In addition, porting such a large emulator with multiple dependencies into a restricted environment is challenging.

In the future, we plan to analyze further whether several transitions can be grouped. It is possible that, in some cases, it would be more beneficial to continue that costly emulation rather than switching to virtualization for a short period and then switching back.

## 7 CONCLUSIONS

In this paper, we presented a Practical Dynamic Information-Flow Tracker (PDIFT) implemented as a

thin hypervisor with an embedded emulator for the Intel architecture. We showed that PDIFT has better performance than previous solutions. We believe that PDIFT can form the basis for the native DIFT on Android platforms, thus providing a complete solution for the DIFT problem in these environments.

## REFERENCES

- Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA.
- Bornstein, D. (2008). Dalvik vm internals. In *Google I/O developer conference*, volume 23, pages 17–30.
- Chen, K., Guo, X., Deng, Q., and Jin, Y. (2021). Dynamic information flow tracking: Taxonomy, challenges, and opportunities. *Micromachines*, 12(8):898.
- Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P. B., Mowry, T. C., Ramachandran, V., Ruwase, O., Ryan, M., and Vlachos, E. (2008). Flexible hardware acceleration for instruction-grain program monitoring. *ACM SIGARCH Computer Architecture News*, 36(3):377–388.
- Das, D., Bose, P., Machiry, A., Mariani, S., Shoshitaishvili, Y., Vigna, G., and Kruegel, C. (2022). Hybrid pruning: Towards precise pointer and taint analysis. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer.
- Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., and Rao, B. (2008). Quantitative comparison of Xen and KVM. *Xen Summit, Boston, MA, USA*, pages 1–2.
- Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29.
- Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. (2006). Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41.
- Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. (2012). libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132.
- Kiperberg, M. (2021). Preventing malicious communication using virtualization. *Journal of Information Security and Applications*, 61:102871.
- Kiperberg, M., Yehuda, R. B., and Zaidenberg, N. J. (2020). Hyperwall: A hypervisor for detection and prevention of malicious communication. In *International Conference on Network and System Security*, pages 79–93. Springer.
- Lawton, K. P. (1996). Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7–es.
- Leon, R. S., Kiperberg, M., Zabag, A. A. L., Resh, A., Al-gawi, A., and Zaidenberg, N. J. (2019). Hypervisor-based white listing of executables. *IEEE Security & Privacy*, 17(5):58–67.
- Machiry, A. (2017). The need for extensible and configurable static taint tracking for c/c++. <https://machiry.github.io/blog/2017/05/31/static-taint-tracking>.
- Oracle (Accessed Nov. 2022). VirtualBox. <https://www.virtualbox.org/>.
- Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350.
- Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., et al. (2009). Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130.
- Sibai, F. N. (2008). Evaluating the performance of single and multiple core processors with PCMARK® 05 and benchmark analysis. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):62–71.
- Slowinska, A. and Bos, H. (2009). Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74.
- Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M. (2008). Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184. IEEE.
- VMware (Accessed Nov. 2022a). VMware ESXi. <https://www.vmware.com/il/products/esxi-and-esx.html>.
- VMware (Accessed Nov. 2022b). VMware Workstation Pro. <https://www.vmware.com/products/workstation-pro.html>.
- wfeldt (2022). libx86emu. <https://github.com/wfeldt/libx86emu>.
- Xue, L., Qian, C., Zhou, H., Luo, X., Zhou, Y., Shao, Y., and Chan, A. T. (2018). Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828.
- Yan, L. K. and Yin, H. (2012). {DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *21st USENIX security symposium (USENIX security 12)*, pages 569–584.
- You, I. and Yim, K. (2010). Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE.
- Zimmer, R. (2009). Hale, “UEFI: From Reset Vector to Operating System,” Chapter 3 of *Hardware-Dependent Software*.