DECAF++: Elastic Whole-System Dynamic Taint Analysis

Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin
University of California, Riverside
{adava003,zqi020,yuq, hengy}@ucr.edu

Abstract

Whole-system dynamic taint analysis has many unique applications such as malware analysis and fuzz testing. Compared to process-level taint analysis, it offers a wider analysis scope, a better transparency and tamper resistance. The main barrier of applying whole-system dynamic taint analysis in practice is the large slowdown that can be sometimes up to 30 times. Existing optimization schemes have either considerable baseline overheads (when there is no tainted data) or specific hardware dependencies. In this paper, we propose an elastic whole-system dynamic taint analysis approach, and implement it in a prototype called DECAF++. Elastic whole-system dynamic taint analysis strives to perform taint analysis as least frequent as possible while maintaining the precision and accuracy. Although similar ideas are explored before for process-level taint analysis, we are the first to successfully achieve true elasticity for whole-system taint analysis via pure software approaches. We evaluated our prototype DECAF++ on nbench, apache bench, and SPEC CPU2006. Under taint analysis loads, DECAF++ achieves 202% speedup on nbench and 66% speedup on apache bench. Under no taint analysis load, DECAF++ imposes only 4% overhead on SPEC CPU2006.

1 Introduction

Dynamic taint analysis (also known as dynamic information flow tracking) marks certain values in CPU registers or memory locations as *tainted*, and keeps track of the tainted data propagation during the code execution. It has been applied to solving many program analysis problems, such as malware analysis [17, 28, 29], protocol reverse engineering [5], vulnerability signature generation [24], fuzz testing [26], etc.

Dynamic taint analysis can be implemented either at the process level, or at the whole system level. Based on process-level instrumentation frameworks such as Pin [19], Valgrind [23], and StarDBT [3], process-level taint analysis tools like LibDft [16], LIFT [25], Dytan [7] and Minemu [4] keep track of taint propagation within a process scope. Whole-system taint analysis tools (e.g., TaintBochs [6], DECAF [12] and PANDA [10]) are built upon system emulators (e.g., Bochs [20] and QEMU [2]), and as a result can keep track of taint propagation throughout the entire software stack, including the OS kernel and all the running processes. Moreover, whole-system dynamic taint analysis offers a better transparency and temper resistance because code instrumentation and analysis are completely isolated from the guest system execution within a virtual machine; in contrast, process-level taint analysis tools share the same memory space with the instrumented process execution.

However, these benefits come at a price of a much higher performance penalty. For instance, the most efficient implementation of whole-system taint analysis to our knowledge, DECAF [12], incurs around 6 times overhead over QEMU [12], which itself has another 5-10 times slowdown over the bare-metal hardware. This overhead for tainting is paid constantly no matter how much tainted data is actually propagated in the software stack.

To mitigate such a performance degradation introduced by dynamic taint analysis, some systems dynamically alternate between the execution of program instructions and the taint tracking ones [13, 25]. For instance, LIFT [25] is based on the idea of alternating execution between an original target program (fast mode) and an instrumented version of the program containing the taint analysis logic. Ho et al. proposed the idea of *demand emulation* [13], that is, to perform taint analysis via emulation only when there is an unsafe input.

Despite the above, there are still some unsolved problems in this research direction. First, LIFT [25] works at the process level, which means LIFT has the aforementioned shortcomings of process level taint analysis. Moreover, LIFT still has to pay a considerable overhead for checking registers and the memory in the fast mode. Second, the demand emulation approach [13] has a very high overhead in switching between the virtualization mode and the emulation mode [13]. Third, some optimization approaches depend on specific hardware features for acceleration [4, 16, 25].

In this paper, we propose solutions to solve these problems, and provide a more flexible and generally applicable

dynamic taint analysis approach. We present DECAF++, an enhancement of DECAF with respect to its taint analysis performance based on these solutions. The essence of DECAF++ is *elastic* whole-system taint analysis. Elasticity, here, means that the runtime performance of whole-system taint analysis degrades gracefully with the increase of tainted data and taint propagation. Unlike some prior solutions that rely on specific hardware features for acceleration, we take a pure software approach to improve the performance of whole-system dynamic taint analysis, and thus the proposed improvements are applicable to any hardware architecture and platform.

More specifically, we propose two independent optimizations to achieve the elasticity: elastic taint status checking and elastic taint propagation. DECAF++ elasticity is built upon the idea that if the system is in a safe state, i.e., there is no data from taint sources, there is no need for taint analysis as well. Henceforth, we access the shadow memory to read the taint statuses only when there is a chance that the data is tainted. Similarly, we propagate the taint statuses from the source to the destination operand only when any of the source operands are tainted.

We implemented a prototype dubbed DECAF++ on top of DECAF. Our introduced code is around 2.5 KLOC including both insertions and modifications to the DECAF code. We evaluated DECAF++ on nbench, SPEC CPU2006, and Apache bench. When there are tainted bytes, we achieve 202% (18% to 328%) improvement on nbench integer index, and on average 66% improvement on apache bench in comparison to DECAF. When there are no tainted bytes, on SPEC CPU2006, our system is only 4% slower than the emulation without instrumentation.

Contributions In summary, we make the following contributions:

- We systematically analyze the overheads of whole system dynamic taint analysis. Our analysis identifies two main sources of slowdown for DECAF: taint status checking incurring 2.6 times overhead, and taint propagation incurring 1.8 times overhead.
- We propose an elastic whole-system dynamic taint analysis approach to reduce taint propagation and taint status checking overhead that imposes low constant and low transition overhead via pure software optimization.
- We implement a prototype based on elastic tainting dubbed DECAF++ and evaluate it with three benchmarks. Experimental results show that DECAF++ incurs nearly zero overhead over QEMU software emulation when no tainted data is involved, and has considerably lower overhead over DECAF when tainted data is involved. The taint analysis overhead of DECAF++ decreases gracefully with the amount of tainted data, providing the elasticity.

2 Related Work

2.1 Hardware Acceleration

Related works on taint analysis optimization focus on a single architecture [4, 16, 25]. Henceforth, they utilize the capabilities offered by the architecture and hardware to accelerate the taint analysis. LIFT uses x86 specific LAHF/SAHF instructions to accelerate the context switch between the original binary code and the instrumented code. Minemu uses X86 SSE registers to store the taint status of the general purpose registers, and fails if the application itself uses these registers. libdft uses multiple page size feature on x86 architectures to reduce the Translation Lookaside Buffer (TLB) cache miss for the shadow memory. In this work, we stay away from these hardware-specific optimizations, and rely only on software techniques to make our solution architecture agnostic.

2.2 Shadow Memory Access Optimization

Minimizing the overhead of shadow memory access is crucial for the taint analysis performance. Most related works reduce this overhead by creating a direct memory mapping between the memory addresses and the shadow memory [4, 16, 25]. This kind of mapping removes the lookup time to find the taint status location of a given memory address. The implementation of the direct mapping requires a fixed size memory structure. This fixed size structure to store the taint status is practical only for 32-bit systems; to support every application, such an implementation requires 32 TB of memory space on 64 bits systems [16]. Even on 32-bit systems, the implementation usually incurs a constant memory overhead of around 12.5% [16]. Minemu furthers this optimization and implements a circular memory structure that rearranges the memory allocation of the analyzed application. The result is that it quickly crashes for applications that have a large memory usage [4]. In this work, we aim to follow a dynamically managed shadow memory that can work not only for applications with large memory usage but also for 64-bit applications.

In addition to the above, LIFT [25] coalesces the taint status checks to reduce the frequency of access to the shadow memory. To this end, LIFT needs to know ahead of time what memory accesses are nearby or to the same location. This requires a memory reference analysis before executing a trace. LIFT scans the instructions in a trace and constructs a dependency graph to perform this analysis. LIFT reports that this optimization is application dependent (sometimes no improvement), and depends on what percentage of time the taint analysis is required for the application. LIFT is a process level taint analysis tool, and in case of whole system analysis, we expect the required taint analysis percentage for a program to be very low in comparison to the size of the system. Henceforth, we do not expect this optimization to

be very useful for whole system analysis given the constant overhead of performing memory reference analysis for the entire system.

2.3 Decoupling

Several related works reduce the taint analysis overhead by decoupling taint analysis from the program execution [14, 15, 18, 21, 22]. ShadowReplica [14] decouples the taint analysis task and runs it in a separate thread. TaintPipe [22] parallelizes the taint analysis by pipelining the workload in multiple threads. StraightTaint [21] offloads the taint analysis to an offline process that reconstructs the execution trace and the control flow. LDX [18] performs taint analysis by mutating the source data and watching the change in the sink. If the sink is tainted, the change in the source would change the sink value. LDX reduces the overhead by spawning a child process and running the analysis in the spawned process on a separate CPU core. RAIN [15] performs on-demand dynamic information level tracking by replaying an execution trace when there is an anomaly in the system. RAIN reduces the overhead by limiting the replay and the analysis to a few processes in the system (within the information flow graph) based on a system call reachability analysis. Our work complements these works as we aim to separate the taint analysis from the original execution.

2.4 Elastic Tainting

Elastic taint propagation Similar ideas to elastic tainting have been explored in the previous works [13, 25]. Qin et al. introduced the idea of fast path optimization [25]. Fast path optimization is based on the notion of alternating execution between a target program and an instrumented version of the program including the taint analysis logic. The former is called check execution mode, and the latter is called track execution mode. Qin et al. presented this idea for process level taint analysis. We build upon this idea for whole system analysis. While the intuition behind both elastic tainting and fast optimization is the same, our work advances Qin et al.'s work for the whole system.

Our first novelty is that we reduce the overhead in the check mode. LIFT [25] checks registers and memory locations of every basic block regardless of the mode. Note that this overhead in system level analysis is a major issue because it affects every process (and the kernel) as we show in §3.3. We reduce the overhead by releasing DECAF++ from checking registers in the check mode and instead monitor the taint sources, data from input devices or memory locations, directly. Combining this with our low overhead taint checking, we reduce the overhead in the check mode to nearly zero as we show in §6.4. Our second novelty is that, unlike LIFT [25], we implement elastic tainting in an architecture agnostic way. Meeting this requirement while obtaining a low overhead is technically challenging. As an example, LIFT uses a simple jump to switch modes while such a jump in our case would panic the CPU since a single guest instruction might break into several host binary instructions that need to be executed atomically.

Finally, the effectiveness of elastic taint propagation for whole system taint analysis has not been investigated before. As we show in §6, this optimization for whole-system taint analysis is application dependent and needs to be accompanied with a taint status checking optimization. Our work is the first that shows the elastic property through comprehensive evaluations, and provides a means to compare the elastic taint propagation with elastic taint status checking.

Elastic taint status checking Ho et al. present the idea of demand emulation [13] that has elements in common with our elastic taint status checking. The demand emulation idea is to perform taint analysis via emulation only when there is an unsafe input e.g. network input in their case. Otherwise, the system is virtually executed without extra overhead. Demand emulation idea looks enticing because the virtualization overhead is usually lower than emulation. However, as Ho et al. state, the transition cost between virtualization and emulation is quite high and possibly offsets the speedup gained through the virtualization. In contrast, as we show in §6.4, our elastic tainting incurs almost zero transition overhead. Further, Ho et al. had to modify the underlying target operating system to provide efficient support for demand emulation. In contrast, in this work, we implement elastic tainting using only emulation without any modifications to the target systems, and show substantial improvements in real world applications of taint analysis.

3 Whole-System Dynamic Tainting

In this section, we introduce a basic background knowledge on DECAF, mainly focusing on its taint analysis functionality related to this work. For further details on QEMU, the underlying emulator, we refer the readers to Appendix A.

3.1 Taint Propagation

DECAF defines how instructions affect the taint status of their operands. Going to the details of the rules for every instruction is out of the scope of the current work; an interested reader can refer to [27]. Just to give an idea, *mov* instructions in x86 result in a corresponding *mov* of the taint statuses from the source to the destination (see Figure 1). What is important about the DECAF taint propagation is that DECAF inserts a few instructions before every Tiny Code Generator (*TCG*) IR instruction that do the following:

- Read the taint status of the source operand. The taint status depending on the operand type can be in the shadow registers, temporary variables or in the shadow memory.

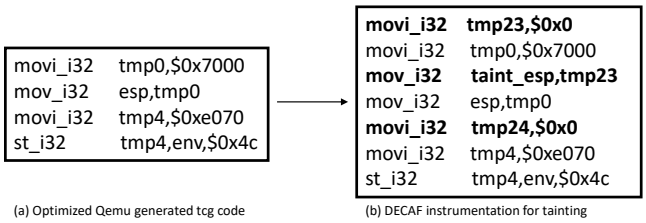


Figure 1: DECAF tcg instrumentation to apply tainting for the instruction *mov $0x7000, %esp*. (a) shows the tcg IR after translating the guest *mov* instruction, and (b) shows the tcg IR after applying the taint analysis instrumentation.

- Decide the taint status of the destination operand based on the instruction tainting rule. To implement the tainting rule, a few TCG IRs are inserted before each IR to propagate the taint status.
- Write the taint status of the destination operand to its shadow variable.

3.2 Shadow Memory

DECAF stores the taint statuses of the registers and the memory addresses respectively in global variables and in the shadow memory (allocated from heap). DECAF does not make any assumptions about the memory and can support any application with any memory requirements. The shadow memory associates the taint statuses with *guest physical addresses*. This is a key design choice because the taint analysis is done at the system level, and hence, virtual addresses point to different memory addresses in different processes.

DECAF stores the taint statuses in a two-level tree data structure. The first level points to a particular page. The second level stores the taint statuses for all the addresses within a physical page. This design is based on the natural cache design of the operating systems, and hence makes use of the temporal and spatial locality of memory accesses.

DECAF instruments QEMU memory operations to maintain the shadow memory. Instrumentation is in fact on the Tiny Code Generator (TCG) Intermediate Representation (IR). DECAF instruments the IR instruction for memory load, *op_qemu_ld**, and the IR instruction for memory store, *op_qemu_st**. For load, DECAF loads the taint status of the source operand of the current instruction along with the memory load operation. For store, DECAF stores the taint status of the destination operand of the current instruction to its corresponding shadow memory along with the memory store operation. In both cases, the load (or store) is to (or from) a global variable named *temp_idx*.

3.3 Taint Analysis Overhead

We analyzed the current sources of slowdown in DECAF. There are three sources of slowdown in DECAF:

- The QEMU emulation overhead that is not inherent to the DECAF taint analysis approach but rather an inevitable overhead that enables dynamic whole system analysis. That said, QEMU is faster than other emulators like Bochs [20] by several orders of magnitude [2].
- The taint propagation overhead as explained in §3.1.
- The taint status checking as explained in §3.2.

After applying DECAF tainting instrumentation, the final binary code is on average *3 times* the original QEMU generated code according to Table 1. Clearly, the additional inserted instructions (after the instrumentation) impose an overhead.

Table 1: The statistical summary of the blownup rate after the instrumentations. The numbers show the ratio of the code size to a baseline after an instrumentation. QEMU baseline is the guest binary code, and DECAF baseline is QEMU IR code. DECAF increases the already inflated QEMU generated code size around 3 times on average.

System	Component	Min	Median	*Mean*	Max
QEMU	lifting binary to tcg	3.33	6.75	7.13	28.00
DECAF	taint checking & propagation	1.25	3.12	2.94	5.14

We systematically analyzed the overheads of DECAF framework, i.e., taint propagation and taint status checking. To measure each overhead, we isolated the codes from DECAF that would cause the overhead by removing other parts. For taint propagation overhead measurement, we removed the shadow memory operations by disabling the memory load and store patching functionality of DECAF that adds the shadow memory operations. For taint status checking overhead measurement, we removed the taint propagation functionality from DECAF by deactivating the instrumentation that implements the taint propagation rules.

We measured the performance of the isolated versions of DECAF using nbench benchmark on a windows XP guest image with a given 1024MB of RAM. The experiment was performed on an Ubuntu 18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. Figure 2 illustrates the result of our analysis. Figure 2 reports the geometric mean of the nbench reported indexes normalized using a baseline. The baseline is the DECAF without the taint analysis functionality outright, that is, DECAF with only Virtual Machine Introspection (VMI). The result shows that on nbench, taint analysis slows down the system 400%. But more important than that, Figure 2 shows that taint propagation alone slows down the system about 1.8 times while taint status checking alone adds a 2.5 times slowdown.

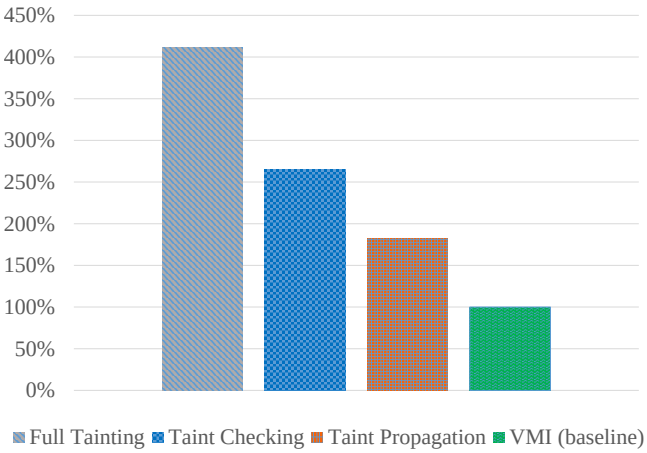


Figure 2: Breakdown of overhead given the DECAF VMI as baseline.

4 Elastic Taint Propagation

4.1 Overview

Elastic taint propagation aims to remove the taint propagation overhead whenever possible. It is based on the intuition that taint analysis can be skipped if the taint analysis operation does not change any taint status value. Taint analysis operations can possibly result in a change only when either of the source or the destination operand of an instruction is tainted.

Two modes Based on the above intuition, we define two modes with and without the taint propagation overhead. We name the mode with taint propagation operations *track mode*, and the mode without taint propagation operations *check mode*. At any given time, the execution mode depends on whether any CPU register is tainted.

Mode transition When the system starts, no tainted data exists in the system, so the system runs in the check mode. The execution switches to the track mode when there is an input from a taint source. The taints propagate in the track mode until the propagation converges and the shadow registers are all zero (clear taint status). At this point, the execution switches back to the check mode. Finally, either based on an input or a data load from a tainted memory address the execution switches back to the track mode. Figure 3 shows when transition occurs between the track and the check mode.

4.2 Execution Modes

An execution mode determines the way a block should be instrumented. Each mode has its set of translation blocks. Further, each mode has its own cache tables. The execution in a mode can flow only within the same mode translation blocks. This means that blocks only from the same mode would be

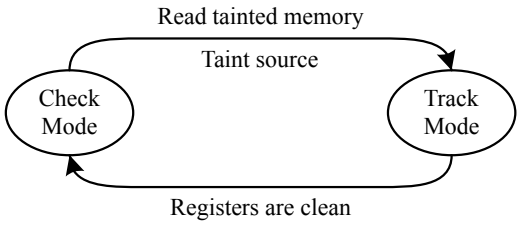


Figure 3: State transitions between check mode and track mode

chained together. We determine the mode using a flag variable. Based on the mode, the final code would be instrumented with or without the taint propagation instructions. The generated code will be reused for execution based on the execution mode unless invalidated. A cache invalidation request invalidates the generated codes regardless of the mode. The set of translation blocks and code caches virtually form an exclusive copy of the translated code for the execution mode.

Check mode The generated code in the check mode is the original guest code (program under analysis) plus the instrumentation code for memory load and memory store. For memory load, shadow memory is checked, and if any byte is tainted, the mode is switched to the track. In §5, we further explain how we efficiently perform this checking. For memory store, the destination operand taint status will be cleaned because any propagation in this mode is safe.

Track mode Code generated in track mode is the same as the one generated in the original DECAF. Readers can refer to §3.1 and §3.2 for further details.

4.3 Transition

A key challenge after having two execution modes in place is to decide when and how the transition between the two modes should occur. The transition between the two modes should affect neither the emulation nor the taint analysis correctness. The execution should immediately stall in the check mode and resume in the track mode when there is a data load from a taint source. Further, this mode switch should happen smoothly without panicking the CPU.

We need to monitor data flow from the taint sources and the shadow registers for timely transition between the modes. In the check mode, we only monitor the taint sources. This is a key design choice for performance because monitoring registers for timely transition is very costly. Note that to implement register monitoring in the check mode, we would need to check the register taint statuses before *every instruction*. Thus, in the check mode, to reduce the overhead, we only monitor the taint sources without loosing the precision. In the track mode, we can check the registers less often because

longer execution in this mode neither affects the taint analysis precision nor the emulation correctness.

Input devices monitoring The taint sources are generally memory addresses of the *input* devices like keyboards or network cards. For the input devices, DECAF++ relies on the monitoring functionality implemented in DECAF. However, we slightly modify the code to raise an exception whenever there is an input. This exception tells the system that it is in an unsafe state because of the user input and the track mode should be activated if not before.

Memory monitoring In addition to the input devices, we should also monitor the *memory load* operations. This is because after processing the data from an input device, the data might propagate to other memory locations and pollute them. We need to track the propagation in the check mode as soon as a tainted value is loaded from memory for further processing. Efficient design and implementation of memory monitoring is a key to our elastic instrumentation solution. We elaborate on how we do this efficiently in §5.

Registers monitoring Monitoring the registers is a key to identifying when we can stop the taint propagation. If none of the registers carry a tainted value, no machine operations except the memory load can result in a tainted value. Henceforth, we can safely stop the execution in the track mode and resume the execution in the check mode while carefully monitoring the memory load operations as explained earlier.

We point out that monitoring registers in the track mode has low overhead. This is because in the track mode, we can tolerate missing the exact time that the registers are clean without affecting either the safety or the precision (we would propagate zero). Therefore, DECAF++ can check the cleanness of the registers in the track mode at block (instead of instruction) granularity.

We check the registers' taint status either after the execution of a chain of blocks, or when there is an execution exception (including the interrupts). Our experiments confirm that this is a fine granularity given its lower overhead comparing to an instruction level granularity approach. If all the registers have a clean taint status, we resume the execution for the next blocks in the check mode.

Transition from check mode to track mode Unlike the transition from the track mode to the check mode (always in the beginning of a block), the transition from the check to the track can happen anywhere in the block depending on the position of an I/O read or a load instruction. However, the execution of a single guest instruction should start from the beginning to the end, note that a single guest instruction might be translated to several TCG instructions [1]; otherwise, the result of the analysis would be both invalid and unsafe. This is because the block code copies in each mode are different and the current instruction might have dependency on the former instructions that are not executed in the current mode. To have a smooth transition, the following steps should be followed:

(1) Restore CPU state: before we switch the code caches, we need to restore the CPU state to the last successfully executed instruction. We need to restore the CPU state to avoid state inconsistency. Since the corresponding execution block in the other mode is different, we can not resume the execution from the same point; the same point CPU state is not consistent with the new mode block instructions. To restore the CPU state, we re-execute the instructions from the beginning of the block to the last successfully executed guest instruction. This will create a CPU state that can be resumed in the other mode.

(2) Raise exception: after restoring the CPU state, we emulate a custom exception: *mem_tainted*. We set the exception number in the *exception_index* of the emulated CPU data structure. After that, we make a long jump to the QEMU main loop (*cpu-exec* loop).

(3) Switch mode: in the QEMU main execution loop, we check the *exception_index* and change the execution mode if the exception is *mem_tainted*. We switch the mode by changing mode flag value that instructs us how we should instrument the guest code (track or check).

After switching the modes, QEMU safely resumes the corresponding block execution in the new mode because we restored the CPU state in the step 1.

5 Elastic Taint Status Checking

The main idea behind reducing the taint status checking overhead is to avoid unnecessary interactions with the shadow memory. In DECAF, the taint status checking happens for every memory operation. However, we can avoid the overhead per memory address if we perform the check for a larger set of memory addresses. Thus, if the larger set doesn't contain any tainted byte we can safely skip the check per address within that set. The natural sets within a system are physical memory pages.

DECAF++ scans physical pages while loading them in TLB, and decides whether or not to further inspect the individual memory addresses. We modified the TLB filling logic of QEMU according to Figure 4. The modifications are highlighted. The figure illustrates that if the page contains any tainted byte, DECAF++ sets a shadow memory handler for

[1] For instance, a single x86 "ADD m16, imm16" instruction will be translated to three TCG IR instructions; one load, one add and one store

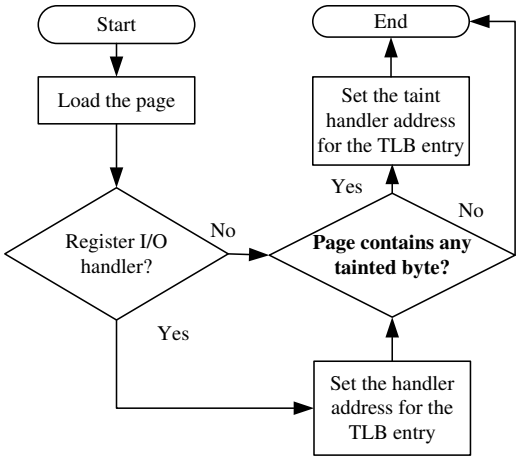


Figure 4: Fill TLB routine

the page through some of the TLB entry control bits. Afterwards, whenever this page is accessed, DECAF++ redirects the requests to the shadow memory handler. In the following paragraphs, we explain how we handle memory load and store operations separately because of their subtle differences.

Memory load During memory load operations, we should load the taint status of the source memory address operands as well. We load the taint status value from the shadow memory only when the TLB entry for the page contains the shadow memory handler. In other cases, we can safely assume that the taint status is zero. Based on this notion, we modify the QEMU memory load operation logic as shown in Figure 5a. In particular, two cases might occur:

- If the TLB entry control bits for the page contain the shadow memory handler, the address translation process for the memory load operations results in a TLB miss. In the TLB miss handler routine, if the control bits indicate that the shadow memory handler should be invoked we do so and load the taint status for the referenced address from the shadow memory. If the execution is not already in the track mode, and the loaded status is not zero we quickly switch to the track mode.
- If the address translation process for the load operation results in a TLB hit, we check whether we are in the track mode, and if so we load zero as the taint value status. Otherwise, we don't need to load the taint status since it will not be used for taint propagation.

Based on the locality principle, a majority of accesses should go through the fast path shown in the Figure 5a. Our elastic taint status checking is designed not to add overhead to this fast path, and hence a performance boost is expected.

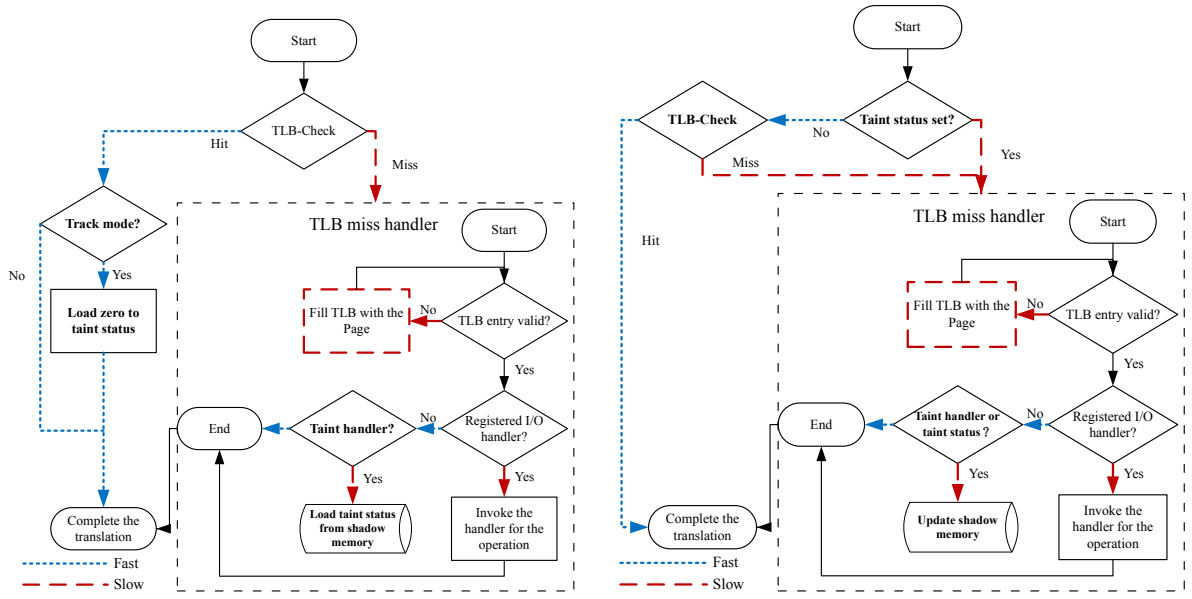
Memory store During memory store operations, we should store the taint status of the source operand to the shadow memory address of the destination referenced memory address. Similar to memory loads, we perform the shadow memory store operation only when the TLB entry control bits for the referenced address page indicate so. That said, there is a subtle difference that makes memory stores costlier than memory loads. Figure 5b shows how we update the shadow memory alongside the memory store operations. In particular, there will be three cases:

- If the source operand for the store operation has a zero taint status, and the page TLB entry does not indicate a tainted page, we do nothing. This happens both in the check mode all the time, and in the track mode when the page to be processed does not contain any tainted byte.
- If the page TLB entry flags us to inspect the shadow memory, we check the TLB control bits in the TLB miss handler and update the shadow memory if the shadow memory handler is set (see Figure 5b).
- If the taint status of the source operand is not zero, even if the page is not registered with a shadow memory handler, we still update the shadow memory. This can only happen in the track mode, because in the check mode there is no taint propagation, and hence the source operand taint status is always zero. We update the TLB entry when this is the first time there is a non-zero taint value store to the page. Since the page now contains at least a tainted byte, all the next memory operations involving this page should go through the shadow memory handler. We update the TLB entry and register the shadow memory handler for the future operations.

Propagation of non-tainted bytes A special case of the taint status store is when a tainted memory address is overwritten with non-tainted data. This case happens when the TLB entry for the page flags shadow memory operation even when the source operand is not tainted. In such a case, the memory address taint status would be updated to zero but the page is still processed as unsafe; the memory operations still will go through the taint handler. For performance reason, we do not immediately reclaim the data structure containing the taint value, but rely on a garbage collection (see Appendix B) mechanism that would be activated based on an interval. The page remains unsafe until the garbage collector is called. The garbage collector walks through the shadow memory data structure and frees the allocated memory for a page if no byte within the page is tainted. After this point, Fill TLB routine will not set the taint handler for the page anymore, and any processing involving the page will take the fast path.

6 Evaluation

In this section, we evaluate DECAF++, a prototype based on the elastic whole system dynamic taint analysis idea. DECAF++ is a fork of DECAF project including the introduced



(a) QEMU load memory operation

(b) QEMU store memory operation

Figure 5: Elastic shadow memory access workflow

optimizations. Overall, our changes (insertion or deletion of code) to develop our prototype on top of DECAf does not exceed 2.5 KLOC. We defined two compilation options that selectively allows activating elastic taint propagation or elastic taint status checking. We evaluate DECAF++ to understand:

1. How effective each of our optimization, i.e., elastic taint propagation and elastic taint status checking and altogether is in terms of performance.

2. Whether our system achieves the elastic property for different taint analysis applications, that, is a gradual degradation of performance based on the increase in the number of tainted bytes.

3. What the current overheads of DECAF++ are and whether we can address the shortcomings of the previous works [13, 25], i.e., reducing the overhead in the check mode and in the transition between the two modes.

6.1 Methodology

We measure the performance metrics using standard benchmarks under two different taint analysis scenarios in §6.2 and §6.3. In both scenarios, a virtual machine image is loaded in DECAF++ and a benchmark measures the performance of the virtual machine while the taint analysis task is running.

To answer (1), we measure the performance of DECAF++ with different optimizations, i.e., with elastic taint propagation dubbed as *Propagation*, with elastic taint status checking dubbed as *Memory* and with the both dubbed as *Full* and compare them. To answer (2) and evaluate the elastic property, we introduce a parameter in our taint analysis plugin that adjusts the number or the percentage of tainted bytes. Plotting the performance trend based on this parameter values allows answering question (2). Finally to answer (3), we measure the overhead of the frequent or costly tasks in our implementation. In the the rest of this section, we describe the details and answer (1) and (2) in §6.2 and §6.3. In §6.4, we answer (3).

6.2 Intra-Process Taint Analysis

In this scenario, we track the flow of information within a single process. For this experiment, we use nbench benchmark [11]. We track the flow of information within the nbench programs using a taint analysis plugin we developed for DECAF. The goal is to be able to report the performance indexes measured by nbnech while the taint analysis task is running. In the next paragraphs, we explain the configurations for the experiment, and at the end of this section we report the results.

nbench Understanding nbench is important since our taint analysis plugin instruments it. nbench has 10 different programs. These programs implement a popular algorithm and measure the execution time on their host. Although the underlying algorithms are different, they all follow the same pattern regarding loading the initial data. They all create an array of random values and then run the algorithm on that array. The arrays are allocated from the heap, and the random generator is a custom pseudo random generator.

Taint analysis plugin The taint analysis plugin instruments nbench programs and taints a portion of the initial input data based on a given parameter. Although this is not trivial, we do

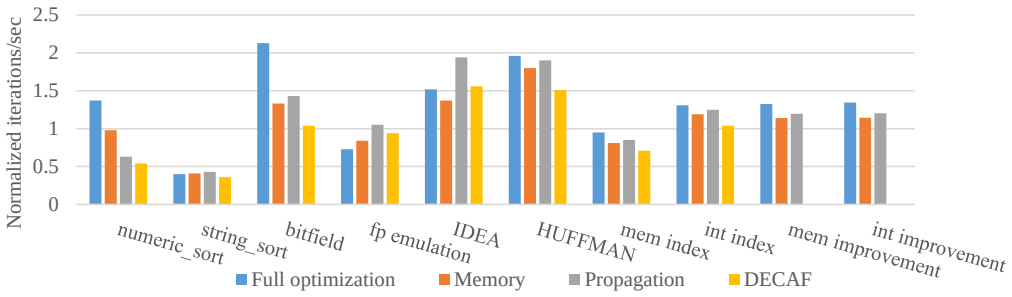


Figure 6: Comparing DECAF and DECAF++ performance.

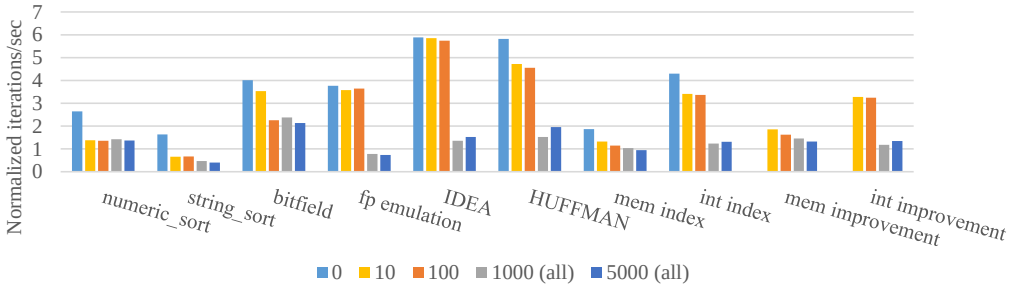


Figure 7: Evaluation of DECAF++ with full optimization under different number of tainted bytes; performance values are normalized by nbench based on a AMD k6/233 system.

not go into the details. We just mention that we record the address of the allocated array from the heap, and taint a portion of the array right after the random initialization. The portion size depends on an input parameter that we call *taint_size*. For instance, *taint_size*=100 means that 100 bytes of the array (from the beginning) used in the running programs of nbench are tainted using the plugin. After the instrumentation, DECAF automatically tracks the propagation from the the taint sources to other memory locations.

Experiments setup We measure the performance of each solitary optimization feature (and together) of DECAF++ using nbench[2] on a loaded windows XP guest image. The image is given 1024MB of RAM. The experiment was performed on an Ubuntu18 i686 host with a Core i7 3.5GH CPU and 8GB of RAM. The reported indexes by nbench are the mean result of many runs (depending on the system performance). Further, nbench controls the statistical reliability of the results and reports if otherwise. Finally, note that since all the measurements are conducted on the VM after it is loaded, the VM overhead would be a constant that is the same for all the measurements.

Result Figure 6 shows the performance of different optimization in DECAF++ in comparison to DECAF. The results in this figure answers question (1) for this scenario. Overall, when the entire program inputs are tainted, combining elastic taint propagation and elastic taint status checking (full optimization) achieves the best performance. However, the performance is application dependant and sometimes a single optimization can achieve better performance than both combined, e.g. HUFFMAN and IDEA.

Figure 7 shows the performance of the DECAF++ when both optimization are activated for varied taint_size values. This figure answers question (2): DECAF++ has the elastic property, that is, the performance degrades based on the number of tainted bytes. *On average, in comparison to DECAF, DECAF++ achieves on average 55% improvement (32% to 86%) on the nbench memory index and 202% (18% to 328%) on the nbench integer index.*

Further, we can see from Figure 7 that results for taint_size={10,100} are similar and differ from the result for taint_size={1000,5000}. For taint_size={10,100}, since the tainted bytes are adjacent, the track mode activates for a sequence of bytes and then quickly switches back to the check mode. Also since taint_size={10,100} is well below a page size, the shadow memory access penalty would be low because often the tainted bytes will be within a single page. However for taint_size={1000,5000}, almost the entire nbench programs input array is tainted that results in frequent execution in the track mode and shadow memory access penalty.

In addition to the above, an interesting observation is the performance degradation for the sort algorithms. The performance degrades abruptly while taint_size changes from zero

[2]Three programs Fourier, NEURAL NET and LU DECOMPOSITION from the nbench did not reflect any change in their reported numbers so we removed them from analysis. Also due to cross compilation, Assignment test did not work on Windows XP.

to greater values. This is because of the behavior of the sort algorithm, that is, frequently moving an element in the array. This behavior results in polluting the entire array quickly and hence degrading the performance abruptly.

6.3 Network Stack Taint Analysis

In this scenario, the taint analysis tracks the flow of information from the network throughout the entire system and every process that accesses the network data. Performing taint analysis in this level is only possible using whole-system taint analysis tools like DECAF. Since the taint analysis affects the entire system, the need of having an elastic property would be more necessary.

Honeypots are an instance of the applications that can greatly benefit from the elastic property. Previous studies show that the likelihood of the malice of a network traffic can be predetermined [8]. Therefore, a honeypot can adopt a policy to achieve taint analysis only for network traffic that are expected to be malicious. Elastic property helps such systems to boost their performance based on their policy.

We measure transfer rate and throughput based on a parameter called taint_perc (instead of taint_size in the previous experiment) that defines the percentage of network packets to be tainted. This is because percentage, here, better represents the real applications. For instance, for honeypots, the taint_perc can be easily derived based on the taint policy.

Taint analysis plugin Our taint analysis plugin taints the incoming network traffic based on the taint_perc parameter. We implemented this plugin using the callback functionality of the DECAF. Our plugin registers a callback that is invoked whenever the network receive API is called. Then, based on the taint_perc parameter, our plugin decides whether to taint the payload or not.

Experiments setup The experiments were performed on an Ubuntu16.04 LTS host with a Core i7 6700 3.40GHz×8 CPU and 16GB of RAM. The guest image was Ubuntu 11.10 and it was given 4GB of RAM. For throughput measurement, we use Apache 2.2.22. We isolate the network interface between the server (guest image running Apache) and the client (the host machine) to reduce the network traffic noise that might perturb the results. That said, there is still a large deviation in the throughput because of the non-deterministic interrupt processing behavior of the system. We rely on significantly different values considering the standard deviation to draw conclusions.

We use netcat to measure the transfer rate. Our measurement is based on the transfer rate for 200 netcat requests of size 100KB. We use apache bench [1] to measure the throughput of an apache web server on the guest image. We execute Apache bench remotely from the host system with a fixed 10000 request parameter. Apache bench sends 10000 requests and reports the average number of completed requests per second. For both transfer rate and the throughput, we repeat the experiments for each taint_perc parameter value 5 times and report the average and the relative standard deviation.

Table 2: Network transfer rate of solitary features of DECAF++ (and together as Full) on Netcat; the throughput is the mean of 5 measurements for a range of tainted bytes.

Tainted Bytes	Implementation	Transfer Rate (MB/S)	Standard Deviation
40KB - 50KB	Full	3.57	18%
	Memory Propagation	3.60	10%
		3.43	8%
20KB - 30KB	Full	5.70	3%
	Memory Propagation	4.25	14%
		3.70	12%
0KB	Full	5.31	5%
	Memory Propagation	5.24	7%
		4.12	8%
0KB - 50KB	DECAF	3.70	9%
0KB - 50KB	QEMU	6.00	3%

Transfer rate result Table 2 reports the result of our transfer rate measurement using netcat. The results show the transfer rate for three taint_perc parameter values: when every packet is tainted (40KB - 50KB tainted bytes), when half of the number of packets are tainted (20KB - 30KB) and finally when no packet is tainted. Note that although every request is 100KB, only a portion of the packet is payload, and not the entire 100KB payload would be live in the system at the same time; this is why eventually only around 50KB is tainted. *The results show a substantial 54% improvement when only half of the incoming packets are tainted*. This is only 5% less than the QEMU transfer rate that is the maximum we can achieve. There is no improvement when every packet is tainted but this is expected because taint propagation and taint status checking have to be constantly done.

Throughput result Figure 8 illustrates the result of our throughput measurement for apache server using apache bench. The figure shows that the full optimization achieves the best throughput. Answering (1), full optimization and elastic taint status checking outperform DECAF for all values of taint_perc, and elastic taint propagation outperforms DECAF when taint_perc is below 1%. Figure 9 shows the performance of the full optimization based on the percentage of the tainted bytes. Although DECAF++ has the elastic property and there is improvement in all cases (answering (2)), it is more tangible for taint_perc values less than 1%. We point out two points on why taint_perc values seem very small. First, the number of

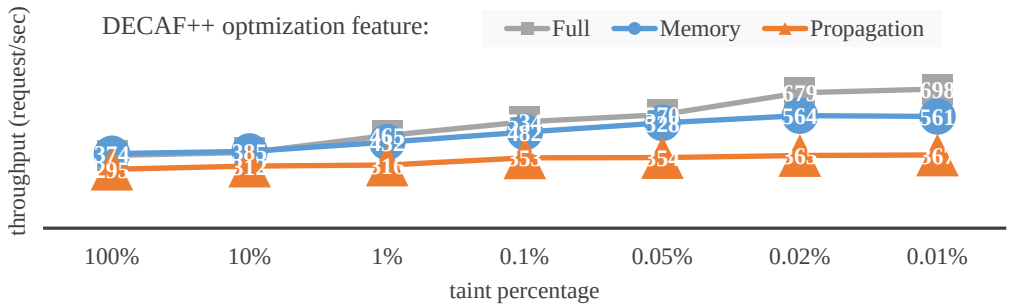


Figure 8: Throughput of solitary optimization features (and together) of DECAF++ in. The reported numbers are the mean of 5 measurements and the relative standard deviation are in range of [2%,18%] for Full, [1%,15%] for Mem and [1%,14%] for the Propagation. DECAF and QEMU 1.0 throughput are 320 and 815 request/sec.

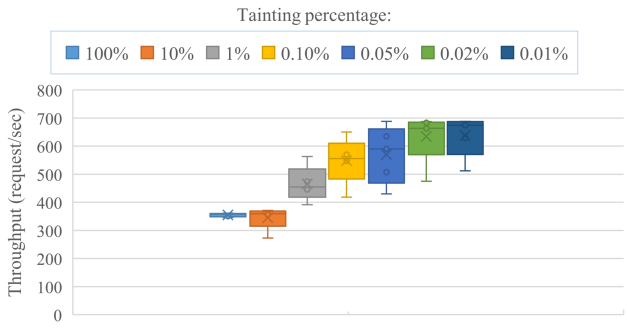


Figure 9: Evaluation of the DECAF++ on Apache bench. Each candlestick shows 5 measurements of throughput (request/sec) for a percentage of tainted packets.

tainted bytes do not linearly decrease with taint_perc. Second, these even seemingly small taint_perc values represent the real world scenarios. For instance for security applications, the attacks are anomaly cases and the percentage of suspicious packets are well below 1%. Overall, DECAF++ achieves an average (geometric) 60% throughput improvement in comparison to DECAF. When there are no tainted bytes, our system is still around 18% slower than QEMU because of the network callbacks.

6.4 Elastic Instrumentation Overhead

In this section, we evaluate DECAF++ to answer (3) and understand whether we could address the shortcomings of the previous works that are high overhead in the check mode of LIFT [25] or high transition overhead of [13].

Check mode overhead Elastic taint analysis imposes an overhead even when there are no tainted bytes. In case of LIFT [25], it's the registers taint tag check at the beginning of every basic block and further memory tag checks before memory instructions. For DECAF++, our evaluation using SPEC CPU2006 and nbench illustrated in Figure 10 and Figure 11 shows that DECAF++ imposes around 4% overhead even when there is no taint analysis task, that is, running only in the check mode. This overhead is in comparison to when tainting functionality is completely disabled. DECAF++ introduces a few overheads in comparison to this case. These overheads are:

Table 3: The nbench evaluation of DECAF++ by removing the potential overheads

Procedure	Memory index	Relative STD	Integer Index	Relative STD
Baseline	4.04	1%	4.36	1%
Full	3.86	2%	4.17	2%
Mode checking	4.04	2%	4.34	3%
Load operations patching	3.87	1%	4.23	1%
Page check in TLB fill	3.74	1%	4.14	1%

- Checking the mode before the code translation and in the memory operations
- Patching the memory load operations in the track mode
- Checking the status of the page throughout the TLB filling process to register the taint status handler

We measured the effect of each of these overheads on indexes reported by nbench by removing the code snippets attributed to these functionalities. The results of these measurements are listed in Table 3. Removing none of the overheads except the mode checking has a substantial effect on the performance. This is because mode checking is frequently done along with every memory load and store operation. It goes unsaid that this overhead is inevitable.

Transition Overhead The transition from the check mode to the track mode imposes an overhead as discussed in section 4.3. This overhead is the major issue with [13]. However, our measurement shows that this overhead is negligible for DECAF++. We measured the transition overhead by recording

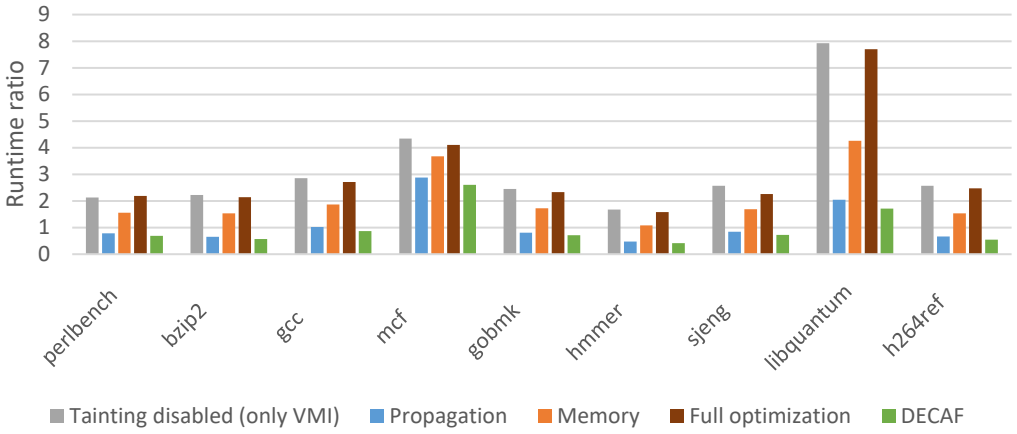


Figure 10: SPEC CPU2006 for different implementations

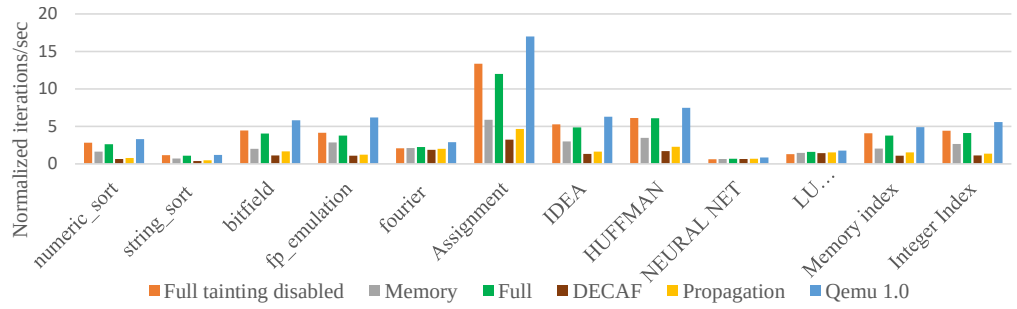


Figure 11: DECAF++ check mode overhead on nbench

the time it takes to change the mode and execute the same instruction that was executing before the transition occurred. Our measurement was performed during nbench execution, and every input byte was tainted. We repeated the measurement 10 times. The average transition time is 0.031% of the overall benchmark execution time with 0.007% relative standard deviation.

7 Conclusion

In this work, we introduced elastic tainting for whole-system dynamic taint analysis. Elastic tainting is based on elastic taint propagation and elastic taint status checking that accordingly address DECAF taint propagation and taint status checking overhead by removing unnecessary taint analysis computations when the system is in a safe state. We successfully designed and implemented this idea on top of DECAF in a prototype dubbed DECAF++ via pure software optimization. We showed that elastic tainting helps DECAF++ achieve a substantial better performance even when all inputs are tainted. Further, we showed how elastic taint propagation and elastic taint checking optimization each and together contribute to the performance improvement for different applications.

Our elastic tainting addresses the shortcomings of the previous works that are either high overhead when there's no tainted bytes or high transition cost when there is some. As a result, DECAF++ has an elastic property for both information flow within a process and information flow of a network input throughout the system. We believe whole-system dynamic taint analysis applications like intrusion detection systems and honeypots can greatly benefit from the elastic property. On one hand, this is because these systems are constantly online and taint analysis affects the entire system constantly. On the other hand, these systems can filter benign traffic and focus on the taint analysis of a small portion of the traffic that are likely to be malicious.

Acknowledgement

We thank the anonymous reviewers for their insightful comments on our work. This work is partly supported by Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATC '05)*, pages 41–41, 2005.

[3] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code generation and Optimization (CGO '06)*, pages 333–345. IEEE Computer Society, March 2006.

[4] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium On Recent Advances in Intrusion Detection (RAID'11)*, pages 1–20, Berlin, Heidelberg, September 2011. Springer.

[5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[6] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, pages 321–336, August 2004.

[7] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 196–206, New York, NY, USA, July 2007. ACM.

[8] Ali Davanian. Effective granularity in internet badhood detection: Detection rate, precision and implementation performance. Master's thesis, University of Twente, August 2017.

[9] Peter J Denning. The locality principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, pages 43–67. World Scientific, 2006.

[10] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5)*, page 4. ACM, December 2015.

[11] Cornelia Cecilia Eglantine. Nbench. 2012.

[12] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 248–258. ACM, July 2014.

[13] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 29–41, New York, NY, USA, 2006. ACM.

[14] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. Shadowreplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 235–246. ACM, 2013.

[15] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 377–390, New York, NY, USA, October 2017. ACM.

[16] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12)*, volume 47, pages 121–132. ACM, 2012.

[17] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, pages 1691–1708, New York, NY, USA, October 2017. ACM.

[18] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 503–515. ACM, March 2016.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200. ACM, June 2005.

[20] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA'08, Beijing*, page 32, 2008.

[21] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*, pages 308–319. IEEE, August 2016.

[22] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*, pages 65–80, August 2015.

[23] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, volume 42, pages 89–100. ACM, June 2007.

[24] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, volume 5, pages 3–4, 2005.

[25] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 135–148. IEEE, 2006.

[26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS'17)*, February 2017.

[27] LK Yan, A Henderson, X Hu, H Yin, and S McCamant. On soundness and precision of dynamic taint analysis. *Dep. Elect. Eng. Comput. Sci., Syracuse Univ., Tech. Rep. SYR-EECS-2014–04*, 2014.

[28] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[29] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 116–127. ACM, 2007.

A QEMU

DECAF is built on top of QEMU [2]. Therefore, to understand DECAF and DECAF++, Qemu knowledge is required. QEMU provides binary instrumentation functionality in an architecture agnostic way via emulation. Qemu emulates the execution of a target binary e.g. a virtual machine image. This means CPU, memory and other hardware are emulated for the target binary. CPU is represented using a *vcpu* data structure that contains all the CPU registers. Providing Memory Management Unit (MMU) is more complicated but the idea is to emulate memory for the target through a software approach known as software MMU (softMMU). We elaborate on softMMU at the end of this section.

Binary translation Qemu first needs to translate the target binary to understand how to emulate it. QEMU loads a guest executable, and translates the binary one block at a time. QEMU translates the binary into an Intermediate Representation (IR) named Tiny Code Generator (*tcg*). The result of this translation is stored in a data structure called *Translation Block* (TB).

Code generation After the translation, Qemu generates an executable code from the translation block. This generated code is written to a data structure called *code cache*. In essence, code cache is an executable page that allows dynamic execution of code. After code generation, Qemu executes the code and updates emulated CPU and memory.

Cache table Qemu stores the result of code translation and generation in a cache to speedup the emulation. Then, before future execution of the same program counter, the cache table would be consulted and the request would be resolved using cache.

Block chaining In addition to the above, Qemu employs another optimization to speed up the emulation. We mentioned that the unit of translation and execution for Qemu is basic blocks. However, after translation of consecutive code blocks, Qemu chains them together and forms a trace. This process is known as block chaining and is implemented by placing a direct jump from the current block to the next one. A trace can be executed without interruption for translation.

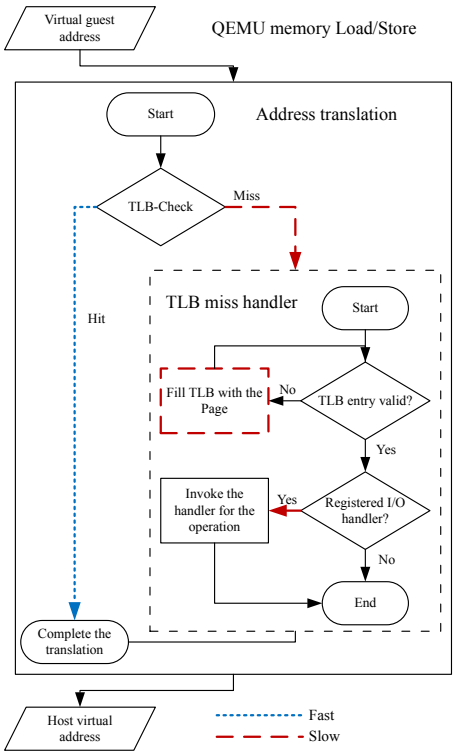


Figure 12: QEMU software memory management through op_qemu_ldlst* operations.

Cache invalidation The generated code blocks may be invalidated to stop further fast execution using cache table. There are different reasons for this. Two main reasons that a code cache would be invalidated are (a) the code cache is full, and (b) the code has been modified and the previous generated code is not valid for re-execution.

Software MMU QEMU software memory management unit (softMMU) translates Guest Virtual Addresses (GVA) to the Host Virtual Address (HVA). This process happens at runtime and within guest memory operations. QEMU generates load and store IR operations for machine level memory operations. QEMU tcg IR store operation, i.e., *op_qemu_st*, stores the content of a register to a given virtual guest address. Qemu tcg load operation, i.e., *op_qemu_ld*, loads the content of a memory address to a given register.

QEMU implements a software Translation Look-aside Buffer (TLB). Hardware TLB maps virtual pages within a process to their corresponding physical pages in the memory. In the Qemu software TLB, the mapping is indeed between the GVA to the HVA. Through address translation and software TLB, QEMU ensures that every guest virtual address (regardless of the process) is addressable in the host QEMU process space.

Figure 12 depicts what happens at runtime in the QEMU memory load and store operations. QEMU needs to make sure that the software TLB contains a valid entry for the following page that will be accessed. To this end, it checks whether the page index portion of the address is valid in the software TLB. If yes, and the page is not registered as a memory mapped I/O, it can be safely (without page fault) accessed. This is the fastest case, and it is expected that based on the *locality principle* [9], a majority of memory operations go through this path. If the page is not present, or the page is registered as memory mapped I/O then the memory operation is much slower.

B DECAF Garbage Collection

A memory address may become tainted, and then may be overwritten through a non-tainted data propagation. In such a case, the shadow memory can be de-allocated. DECAF does not reclaim it immediately for performance reasons. Immediate reclaiming requires DECAF to explore all the leaves (memory addresses) of a parent node (a physical page) for every memory operation that is very costly.

DECAF relies on a garbage collector to reclaim the unused shadow memory. DECAF will handle the unused memory by calling the garbage collector based on an interval. The garbage collector is called in the QEMU *main_loop* that runs in a separate thread. Experimentally, garbage collector is set to walk the shadow memory pages every 4096 times the main loop runs.

The garbage collector walks through the shadow memory and checks every parent and all its leaves. It will return an unused leaf to a memory pool. This pool will use the leaf for another taint status storage. Finally, the garbage collector will return a parent to the memory pool if none of its children is in use.