Metering Graphical Data Leakage with Snowman

Qiuyu Xiao
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
qiuyu@cs.unc.edu

Brittany Subialdea
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
britsubi@cs.unc.edu

Lujo Bauer
Carnegie Mellon University
Pittsburgh, PA, USA
lbauer@cmu.edu

Michael K. Reiter
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA
reiter@cs.unc.edu

ABSTRACT

A long-standing technique to interfere with theft of sensitive data by its intended users is permitting these insiders only remote access to the data via a thin client. Even allowing only remote access is inadequate, however, to counter an insider willing to reconstruct the data from the graphical output, in the limit by photographing the data on-screen and applying automatic character recognition to these photographs offline. In this paper we propose and evaluate a system, called Snowman, that accurately monitors the amount of sensitive data output to a client. To conduct this monitoring without slowing the interactive user session, leakage is concurrently tracked in a replica of the application execution. This, in turn, introduces a key technical challenge that Snowman solves, namely identically replicating execution of an unmodified Linux binary while also performing efficient multi-label taint-tracking on it. We show through empirical measurements with a word processor, a spreadsheet program, and a code editor that Snowman induces little overhead on interactive user sessions and easily differentiates data-access patterns induced by normal usage and sufficiently aggressive data theft with reasonable responsiveness.

ACM Reference Format:
Qiuyu Xiao, Brittany Subialdea, Lujo Bauer, and Michael K. Reiter. 2020. Metering Graphical Data Leakage with Snowman. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies (SACMAT '20), June 10–12, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3381991.3395598

1 INTRODUCTION

Data theft by insiders is a threat that is especially difficult to prevent, as it involves misuse of permissions that the insider presumably must be given to perform his/her duties in the organization. It is thus not surprising that such data thefts are so common; e.g., in healthcare, insider threat is the most common cause of data leakage, accounting for 58% of incidents [47]. And, of course, insiders were behind some of the highest profile data breaches of U.S. government data, with the Manning [52] and Snowden [49] cases being two exemplars. All U.S. executive agencies and military departments are now required "to monitor user activity on all classified networks in order to detect activity indicative of insider threat behavior" [39, §H.1], and a National Insider Threat Task Force has been established "to develop a Government-wide insider threat program for deterring, detecting, and mitigating insider threats" [2].

An approach to address data theft by insiders is to make sensitive data available to users only by secure remote access. In this approach, programs execute on sensitive data only on computers trusted by the organization, while users are permitted to interact with those programs/data only remotely, perhaps from a less-trusted (even user-owned) computer or a "thin" client having no persistent storage of its own. While this approach has been practiced for decades in various forms (e.g., [54]), a current product embodying this approach is Citrix Virtual Apps and Desktops [1], formerly marketed as XenApp and XenDesktop. The deployment of XenDesktop by Osaka Gas [3] provides an illustrative example of the data security benefits that this approach can offer.

Despite the data-protection benefits of remote-only access, this approach is fundamentally limited by the possibility that the user screenshots sensitive content or even photographs it using another device. Distribution of such images can be discouraged through the introduction of watermarks (e.g., by varying screen luminescence [23]). However, the ability to attribute the data leak to an individual after the data is already leaked might be ineffective in deterring the leak. Moreover, since these watermarks must not interfere with the user experience, text data can be recovered, sans watermarks, by applying optical character recognition to the images [14]. For small text data files (e.g., a Word file that fits on one or two screens), it seems that there is little hope for defending further against such insiders.

The premise of this paper, however, is that for *large* amounts of sensitive text—e.g., large documents, databases, or codebases—an effort to quickly display significant amounts of that data to a computer screen to record it (e.g., using another device) is likely to induce patterns of accessing that data that departs from the norm for interacting with it for legitimate purposes. Combined with remote-only access, this observation might be leveraged to detect data theft *as it is occurring*. In this paper, we propose a system, called Snowman, to accomplish this goal. Roughly speaking, Snowman monitors the transmission of sensitive data by a program to a remote client via a graphical user interface, and raises an alert when the rate of transmission exceeds what is typical for interacting with that data.

SACMAT '20, June 10–12, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7568-9/20/06.
https://doi.org/10.1145/3381991.3395598

A central challenge in this approach is how to measure the amount of sensitive data transmitted to a remote client. While the total volume of GUI data transmitted to the remote client is an upper bound, such a bound is very coarse. For example, numerous ways of interacting with a program—e.g., scrolling a document a few lines, up and down repeatedly—would result in an ever growing estimate of leakage, even though only the same few lines of data are being rendered to the user's screen. Snowman therefore employs taint analysis (e.g., [15, 16, 37]) to track which sensitive bytes taint each byte output to the remote user. By tracking the cumulative set of sensitive bytes that taint the output to the remote user, Snowman can improve the accuracy of this upper bound considerably, and even determine *which* sensitive bytes might have been leaked.

Unfortunately, multi-label (i.e., per sensitive byte) taint analysis on unmodified binaries is very expensive, incurring a typical overhead of 7× or more [29]. To ensure that this overhead does not interfere with the user experience, Snowman thus performs taint analysis only on a *replica* of the program that replays the program's execution alongside the program instance that interacts with the user. This replica lags behind the user-facing instance due to the overhead of taint analysis, but as we will show, it need not lag by much for typical user behavior. Herein lies a core technical challenge that we solve in Snowman, namely how to efficiently conduct taint analysis on a replica while forcing the replica to execute *identically* to the original, user-facing execution.

Replication systems (e.g., [8, 40]) need to record asynchronous signals and scheduling events, and to replay them to the replica at the exact same execution points as in the original execution. Those systems rely on CPU hardware performance counters to measure the number of instructions executed by the program to locate the right execution points at which to replay them. However, conventional taint analysis tools [11, 29] use dynamic binary instrumentation to insert analysis routines into the original code blocks, which would break the measurement of the replica's execution and so cause the replica to diverge from the original. Snowman employs a novel architecture that conducts taint analysis in the kernel without disturbing the performance counter measurements. To reduce the overhead of taint analysis, Snowman employs various optimizations, such as excluding application basic blocks from in-kernel analysis where it can be inferred that they have no tainted operands, caching instruction decoding results, copy-on-write taint propagation, and garbage collection of taint tags corresponding to already-leaked data.

We have implemented Snowman to work on unmodified x86-64 Linux binaries and off-the-shelf hardware. We evaluated Snowman on three widely used GUI programs: LibreOffice Writer (a word processor), LibreOffice Calc (a spreadsheet program), and Gedit (a code editor). Our evaluation shows that Snowman introduces only moderate overhead on common user actions and performs better than the Pin "null tool", which serves as a baseline for many previous taint analysis systems [26, 29, 33, 34] and any other Pin-based [32] solutions. The evaluation also demonstrates that Snowman can easily distinguish normal user behaviors from ones reflecting data copying in all tested programs, by analyzing the sensitive data leakage patterns.

To summarize, the contributions of our paper are as follows:

- To our knowledge, we provide the first system designed to detect copying of graphical output to reconstruct sensitive data, e.g., to exfiltrate it later. Rather than focusing on watermarking to assign responsibility for the theft after the released data is recovered, Snowman instead seeks to detect the copying while it is occurring.
- We detail the design of Snowman, which performs multi-label taint-tracking only on a replica of the user-facing execution, to minimize the performance impact to the interactive user experience. In doing so, Snowman simultaneously achieves exact replication of the user-facing execution while performing multi-label taint tracking on it, without modification to the program binary. Central to its efficiency are a variety of optimizations that render it far more lightweight than straightforward solutions (e.g., based on Pin [32]).
- We show through evaluations of Snowman on a fully-featured word processor, spreadsheet program, and code editor, that sufficiently aggressive copying can easily be differentiated from normal usage examples based on the rate of GUI leakage of sensitive file output. We also show that Snowman supports copying detection with minimal penalties to the responsiveness observed by the user, and with modest delays from the time at which the leakage occurs.

The rest of this paper is organized as follows. We summarize related work in Sec. 2 and describe the design and implementation of Snowman in Sec. 3. In Sec. 4, we evaluate Snowman in terms of its performance on various user actions and its capability to differentiate malicious data-access patterns from normal ones. We discuss remaining challenges and possible extensions in Sec. 5, and conclude the paper in Sec. 6.

2 RELATED WORK

Thin-client systems. In a thin-client system, the server renders graphical data from data source and transmits the generated graphical data to the client over the network. The client doesn't have persistent storage and can only display graphical data and take user inputs. Baratto et al. [4] proposed a high-performance thin-client architecture by directly exposing the video hardware to the display system. Yang et al. [54] measured the performance of six popular thin-client systems and explained the performance impact of the underlying remote display protocols. Shi et al. [44] gave a survey on various 3D rendering thin-client systems. Orthogonal to the previous work, our paper focus on improving security of the thin-client system by offering a generic solution to monitoring the sensitive bytes leaked to the client.

Anomaly detection. Here we treat a malicious insider exfiltrating a large volume of sensitive data from an organization as an anomalous behavior to be detected using anomaly-detection techniques, of which many have been proposed (see, e.g., [13]). In anomaly-detection systems, various events of the user and the running programs are logged. The logs might include, e.g., system calls, shell commands, file reads and writes, and others. These logged data are then provided to the feature-based detection algorithms, which can be based on machine learning [38], data mining [53], statistics [30], or information theory [31], to identify anomalies.

The main contribution of Snowman is offering a novel system architecture that (i) restricts user's interaction with the sensitive files to the GUI interface in a thin client and (ii) accurately monitors the sensitive bytes leaked to the user. Snowman can generate logs containing the indices of the leaked bytes and the timestamps of the leakage events, which can be used as features by the anomaly-detection algorithms. This fine-grained sensitive data leakage pattern gives more insight into the user's intent compared with coarser patterns of access to files or file blocks. In this sense, Snowman provides a new type of feature for anomaly-detection systems to analyze, though our goal here is not to develop new anomaly detection algorithms ourselves.

Taint analysis. The conventional approach to monitor sensitive information flow is taint analysis (e.g., [15, 16, 37]). Taint analysis systems attach taint tags to the memory and register locations whenever the program consumes sensitive data. Along with the execution of the program, the taint tags are propagated from one location to another. Since the taint propagation rules are dictated by the instruction semantics, taint analysis can accurately monitor how the sensitive data is transformed and transferred, and whether it is leaked through the program's execution.

Taint analysis can be implemented with dynamic binary instrumentation (e.g., [11, 26, 29, 33, 34, 43]) or virtualization (e.g., [15, 24, 42]). In inlined taint analysis systems [11, 15, 24, 29, 42, 43], the taint analysis logic is interleaved with the analyzed program's execution flow and so introduces substantial overhead. For example, libdft, a state-of-art taint analysis system, imposes 7.06× slowdown to the Firefox browser even after employing various optimization techniques [29]. Some recent systems [26, 33, 34] aim to reduce overhead of the analyzed program by decoupling taint analysis from the program's execution. These systems record the control flow and memory access information with Pin, a dynamic binary instrumentation tool, and run the taint analysis logic in a separate thread with the recorded information. Snowman also decouples taint analysis from the execution of the monitored program, by replicating the program's execution and conducting taint analysis only on the replica. Since Snowman doesn't do heavyweight binary instrumentation, it adds less overhead to the monitored execution compared with the Pin "null tool" (as will be shown in Sec. 4) and hence all other Pin-based tools.

Some systems implement taint analysis in hardware [17, 28, 48, 51] and usually have better performance than the software based systems. However, custom hardware for this purpose is not widely deployed. Snowman has better applicability since it works on off-the-shelf hardware and unmodified binaries.

Replicated execution. Replicating the execution of a multi-threaded program in multicore systems is a challenging problem. Many sources of nondeterminism can lead to divergence of the replicated execution. The first type of nondeterminism is caused by the program's communication with the system or other programs via systems calls, e.g., `read()`, or instructions, e.g., `RDTSC`. Replication systems (e.g., [8, 40]) usually address this type of nondeterminism by recording the nondeterministic inputs to the original execution and replaying the recorded values to the replica. The second type of nondeterminism is caused by shared-memory interactions among threads or processes. To address this type of nondeterminism, the approaches taken by replication systems include replicating all shared-memory accesses [9, 41]; scheduling one thread or process at a time and replicating the scheduling decisions [35, 40, 46]; assuming the program is race-free and replicating the synchronization events by instrumenting the synchronization library [5, 6, 18]; or applying a deterministic scheduling algorithm to remove the nondeterminism in shared-memory interactions [7, 8, 19].

Replicating the program's execution can be also achieved by running the program in a virtual machine and replicating the execution of an entire virtual machine [12, 21, 22]. However, doing so introduces unnecessary overhead of replicating the execution of the operating system and other programs. There are also replication systems [25, 36] relying on custom hardware to reduce overhead. We adapted RR [40], an open-source tool from Mozilla, to implement Snowman's replication subsystem. We chose RR because it works on unmodified binaries and off-the-shelf hardware. Additionally, RR doesn't assume race-freedom of the replicated program, and so programs with data races (e.g., programs using lock-free data structures [50]) can be directly replicated by RR.

Replication and dynamic analysis. Several previous works [18, 20, 27, 41] explored the idea of combining replication and dynamic analysis. Unlike Snowman, which replicates and analyzes the execution of the program concurrently with the original execution, these systems can replicate the previously recorded execution of the program only after the program exits, to conduct analysis for debugging, auditing, or attack provenance. In addition, the high overhead caused by replicating every shared-memory interaction [41] or replicating a virtual machine [20], as well as assuming race-freedom [18, 27], do not fit our use cases.

3 SYSTEM DESIGN AND IMPLEMENTATION

3.1 Overview

In Snowman, GUI programs with permissions to access sensitive files run in a remote server. A user employs her personal computer to interact with the GUI programs over the network. The GUI programs take user inputs, such as mouse clicks and keystrokes, do computations, and deliver graphical outputs to the user computer. In our threat model, the user computer is not trusted and the user might intend to steal sensitive information. However, we assume the server computer and all software running on the server are trusted. The only channel where a user can get sensitive information is the graphical outputs generated by the remote GUI programs. Our system aims to accurately monitor the amount of sensitive file information flowing from the remote server to the user computer without affecting the normal usage of the GUI programs.

To accomplish this, Snowman replicates the execution of the analyzed program and conducts taint analysis on the replica, to minimize the impact of that analysis on the performance of the original execution. Our system works on legacy x86-64 binaries without requiring custom hardware or recompilation. Fig. 1 shows the overall architecture of our system. The graphical user interfaces of the program are displayed in the user computer (a thin client). In the remote server, the replication engine creates and maintains a replica for every thread and process of the monitored program. The replicas maintain the exact same execution states as the original

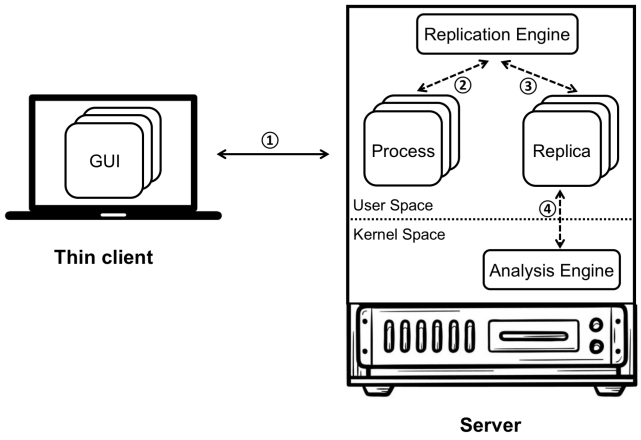


Figure 1: Snowman architecture: (1) communicate user inputs and graphical outputs (over the network); (2) record program execution; (3) replicate program execution; (4) monitor sensitive data leakage

threads and processes, including memory values, register values, and control flows. The analysis engine conducts taint analysis on the replicated processes by attaching and propagating taint tags to monitor the sensitive information flow. There are many challenges in efficiently performing taint analysis and, at the same time, faithfully replicating the original program's execution. We will give detailed descriptions of our system in the following sections.

3.2 Replication engine

The implementation of the replication engine is based on RR [40], which is designed to record and replay multi-threaded Linux programs with low overhead, and is useful in debugging concurrency bugs. Instead of replaying the whole execution after the program exits, we adapt RR to run a replicated program side-by-side along with the original program execution throughout its lifetime.

The core problem solved by RR is faithfully replicating the execution of a multi-threaded program. In principle, if all non-deterministic inputs (from the operating system to the application) and events of the original execution are recorded and replayed to the replica, the replicated execution should be the exact same as the original one. RR runs in user-space and monitors the target program via the `ptrace` system call. RR can observe various events of the monitored program including system calls and signals. Whenever a monitored process enters or exits a system call, it is suspended and RR is notified. For system calls that spawn a new process or thread, like `fork()` and `exec()`, RR creates a corresponding replicated process or thread and copies the original memory and register state to the replica. For the system calls that consume non-deterministic inputs from the operating system, such as `read()` and `gettimeofday()`, RR records the inputs to the system call from the original process and replays them to the replicated process without actually executing the system call. RR also deals with inputs from non-deterministic instructions, including `RDTSC` and `RDRAND`, by emulating or rewriting those instructions.

Besides non-deterministic inputs to system calls and from non-deterministic instructions, RR also needs to record and replay the non-deterministic events. The first type of non-deterministic events is scheduling events. Multiple threads of the same process run concurrently and do computations on shared data, and so different thread schedules could lead to different outcomes of the program. Without replaying the scheduling events, the replica's execution could diverge. Another type of non-deterministic events is signals. Signals usually interrupt the normal execution flow of the program. If a signal handler is registered, it will be called to handle the arrived signal. Since the signal handler could compute on data shared with normal program code, similar to scheduling events, we have to record and replay the signals to avoid divergence.

RR acts as a scheduler to the monitored program and only schedules one thread at a time. By using a deterministic performance counter of the Intel CPU, RR tracks the number of retired conditional branches (RCB) and uses the RCB counts to mark the progress of the program's execution. Whenever a non-deterministic event happens in the original program, RR records the timing, measured by the RCB count, of that event, and replays this event in the exact same execution point of the replica. RR instructs the CPU to fire an interrupt after a specified number of conditional branches are retired by the replica to control the timing of the event replay. If it is a scheduling event, RR preempts the replicated threads to replay the schedule. If it is a signal, RR emulates the execution of the signal handler without delivering a real signal to the replica.

During the original execution of the program, RR records the aforementioned data. Consumption of the recorded data is sometimes slower than its generation because conducting taint analysis on the replicated program could slow down its execution. The replication engine buffers the recorded data in the file system, so that the original program execution can advance normally without having to wait for the replica.

3.3 Analysis Engine

If the replication engine maintains a replicated program execution that progresses exactly as the original one, conducting taint analysis on the replica should expose the same sensitive information flows as occurred in the original. The analysis engine's goal is thus to track which sensitive file bytes taint GUI outputs of the replica execution (and so of the original execution, to the client computer) without causing the replica's execution to diverge from the original. libdft [29] and other similar tools (e.g., [11, 16]) use dynamic binary instrumentation to transform the original code blocks to semantically equivalent ones intertwined with the taint analysis logic. We don't take such an approach to implement the analysis engine since adding additional instructions in the replica would confuse the RCB counts measured by the replication engine, which might lead to divergence of the replicated execution due to the non-deterministic events being inserted at the wrong execution points. As such, the analysis engine in Snowman takes a different approach, which we summarize in this section.

3.3.1 Architecture. Snowman defines each memory or register byte as an individual taint unit, to which it associates taint tags dynamically. The instructions executed by the program dictate how the

taint tags should be propagated. For example, if the program executes "`mov ebx, eax`", which moves four bytes of data from the `eax` register to the `ebx` register, the analysis engine should move the taint tags on each byte of the `eax` register to the corresponding byte of the `ebx` register. Strictly speaking, the analysis engine is required to perform this type of analysis on every instruction executed by the program.

In Snowman, the analysis engine is implemented as a Linux kernel module, and all taint analysis operations are done in kernel space, which don't interfere with the RCB counts of the user-space replica. For the replica, the analysis engine maintains shadow memory address spaces and shadow registers that store taint tags for the corresponding memory and register bytes. Different threads of the same process share the same shadow memory address space but have their own shadow registers. A strawman approach to implement taint analysis is running the replica in single-step mode. After the execution of each instruction, the CPU traps to kernel mode and transfers control to the analysis engine. The analysis engine then decodes the binary instruction and decides how to propagate taint tags based on the instruction's opcode (e.g., `mov`) and operands (either memory or register operands). This approach works but is too slow since each instruction triggers a CPU context switch from user mode to kernel mode.

One source of optimization is the observation that we need to analyze an instruction only if its operands have taint tags. Leveraging this observation, the analysis engine checks, at the beginning of each basic block, whether the register operands of the instructions in the basic block contain taint tags by inspecting the shadow registers. If any register operand has taint tags, the analysis engine sets the CPU to single-step mode for this basic block. If not, the analysis engine sets a breakpoint at the last instruction of the basic block, which lets the analysis engine seize control and check the next basic block. This design often induces one context switch per basic block instead of per instruction, which could be a big performance gain if only a small percentage of instructions touch tainted registers.

Figuring out whether the memory operands of a basic block contain taint tags is a more complicated task. Since some memory operands can be indirectly addressed—e.g., in "`mov eax, [ebx]`", where the address of the memory operand is the value of the `ebx` register—we may not know the address of the memory operand at the beginning of the basic block. As such, we cannot decide whether the basic block should run in single-step mode by only inspecting the shadow memory. To solve this problem, the analysis engine changes each memory page that contains tainted memory to kernel-only pages by modifying its page table protection bit. Whenever an instruction in the program accesses those protected pages, the CPU generates a page fault and transfers control to the analysis engine. The analysis engine then changes the accessed page to user-accessible and sets the CPU to single-step mode. After that, every instruction in that basic block will be analyzed by the analysis engine.

As shown in Fig. 2, each replicated thread is classified as being in one of four states by the analysis engine. When the replica process has not consumed any sensitive data and there are no taint tags in the shadow memory or shadow registers, all threads of the replicated process are in *Initial* state, and the analysis engine does not set breakpoints at basic-block boundaries. As soon as the replica process consumes sensitive data, all threads transition to *Undecided*. When a thread is in the *Undecided* state, the analysis engine checks whether the register operands in the current basic block contain any taint tags. If so, the thread is transitioned to the *Single-step* state and every instruction in the basic block will be analyzed individually. After the thread finishes the last instruction of the basic block, it is transitioned back to the *Undecided* state. If the register operands are not tainted, the thread transitions to the *Block-step* state, and the analysis engine sets a breakpoint at the last instruction of the basic block and changes all tainted pages to be kernel-only pages. If the thread accesses any tainted page, it is transitioned to *Single-step*. If the thread finishes the basic block without accessing any tainted pages, then it is transitioned to *Undecided*. This state machine ensures that every possible tainted data flow will be captured and analyzed by the analysis engine.

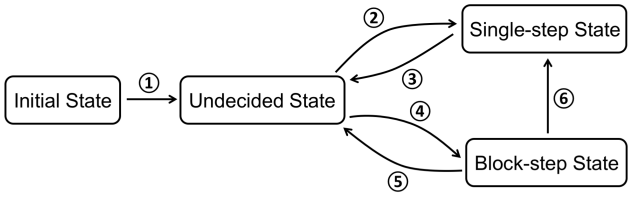


Figure 2: State transitions: (1) consume sensitive data; (2) contain taint tags; (3) finish basic block; (4) no taint tag; (5) finish basic block; (6) protection page fault

3.3.2 Taint Analysis. To monitor the amount of sensitive data leaked out of the GUI program replica through its graphical outputs, the analysis engine assigns a different taint tag to each byte of the sensitive data consumed by the replica. Each shadow memory or register byte can be attached with a list of different taint tags. When the graphical output data is sent out of the system, the analysis engine inspects what taint tags are contained in the output bytes to track which sensitive bytes have been leaked out.

During taint analysis, the analysis engine applies different analysis rules on different instructions based on their semantics. The instructions involving explicit taint propagation can be divided into four categories: movement instructions, arithmetic instructions, logical instructions, and transformation instructions. Fig. 3 gives examples of taint propagation for different types of instructions.

Movement instructions. Movement instructions move data among memory and registers, or assign immediate values to memory or registers. By this definition, `mov`, `pop`, and `push` are movement instructions. Bit shift instructions, like `shr` and `shl`, are also categorized as movement instructions since they can be considered as moving data within a register or memory location. For the movement instructions, the analysis engine first locates the source and destination operands, and then replaces the taint tags in the destination shadow memory or register, byte by byte, with the ones from the source. If the source operand doesn't contain any taint tags or is an immediate value, then the analysis engine clears the taint tags in the destination.

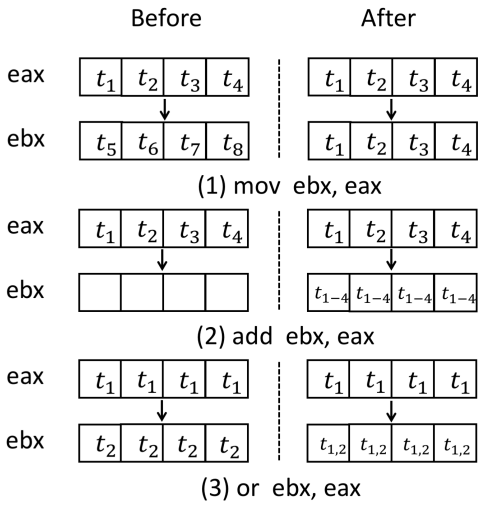


Figure 3: Examples of taint propagation

Arithmetic instructions. Arithmetic instructions do arithmetic operations on the source operands and save the result to the destination operand. `add`, `sub`, `mul`, `div`, and `inc` are common arithmetic instructions. For unary instructions like `inc`, we don't need to change the taint state of their operand. For binary instructions, the analysis engine first accumulates all the taint tags from every byte of the source operands, and then assigns this list of taint tags to every byte of the destination operand in the shadow memory or shadow register. There is a special case for `sub`, or `sbb`, instruction. If the source operands of `sub` are the same, the analysis engine clears the taint tags of the destination operand. For example in "`sub eax, eax`", the taint tags of `eax` are removed.

Logical instructions. Logical instructions do bitwise logical operations on memory or register operands. Different from the arithmetic instructions, in a logical instruction, each byte of the destination operand is affected only by the corresponding bytes of the source operands, and so the analysis engine assigns only the taint tags from those source-operand bytes to the corresponding byte in the destination operand. Special cases here are `and` and `xor`. If one of the source operands in `and` is the immediate value `0`, then the analysis engine clears the taint tags of the destination operand. If the source operands of `xor` are the same, the taint tags of the destination operand are also cleared.

Transformation instructions. Transformation instructions change the data representation of an operand from one type to another. For example, `cvtsi2sd` changes the data from an integer representation to a scalar double-precision floating-point representation and moves the data from the memory or a general register to a SIMD (single instruction, multiple data) register. Semantically, every byte of the source operand affects every byte of the destination operand. So, the analysis engine accumulates all the taint tags from every byte of the source operand, and then assigns this list of taint tags to every byte of the destination operand.

Besides propagating taint tags based on instruction semantics, we also propagate taint tags based on function semantics for selected functions involving table lookups. In table lookups, the table index is saved in the index register of the memory operand, and the value of that memory operand is the corresponding table data. There is an implicit data flow from the index register to the memory cell. To reduce false positives and false negatives, we propagate taint tags directly from the input buffer (containing table indices) to the output buffer (containing table data) for selected table-lookup functions. Currently, we have implemented such taint propagation rules for the `pango_shape_full()` function from the `libpango` library, which translates font indices to unicode characters through table lookups. More discussions about implicit data flows can be found in Sec. 5.

3.3.3 Code Caches. The analysis engine decodes each instruction to figure out 1) its opcode; 2) the names of any register operands; 3) the base register, index register, scale factor, and displacement of any memory operands; and 4) the value of any immediate operands. For each basic block, the analysis engine needs to know which registers are included in the basic block to decide whether they are tainted, to make decisions about state transitions (as described in Sec. 3.3). Decoding instructions is expensive, taking around 1000 CPU cycles per instruction. To leverage the space and time locality of code execution, similar to the CPU instruction cache, we use two in-memory software caches to speed up taint analysis.

Instruction cache. The instruction cache stores the decoded information—opcode, operands, etc.—of instructions. Whenever an instruction is executed, the analysis engine checks whether the instruction is in the cache with an index calculated by the instruction's virtual memory address. If it is not in the cache, the analysis engine decodes the instruction and caches the results. Otherwise, the cached information is directly used without decoding the instruction.

Block cache. We also use a block cache to store the names of register operands in basic blocks. At the beginning of each basic block, the analysis engine checks whether the register names are cached, with an index calculated by the starting address of the basic block. If it is a cache miss, the analysis engine has to decode each instruction in the basic block to get the register names. This decoding step can be accelerated by the instruction cache.

3.4 Implementation

We implemented the replication engine with 61 lines of C++ code on top of Mozilla RR v5.2.0. We implemented the analysis engine with 4454 lines of C code as a kernel module in Linux v4.11.12. We integrated the Zydis[1] disassembler into the analysis engine to decode x86-64 instructions. We will discuss some implementation details in this section.

Analyzed instructions and functions. We implemented taint analysis rules for 28 movement instructions, 14 arithmetic instructions, 12 logical instructions, and 5 transformation instructions. Those are commonly used instructions including a set of SIMD (single instruction, multiple data) instructions. Snowman does not propagate taint tags to the `eflags` register, and it ignores implicit data flows caused by control-flow dependencies. Many previous

[1] https://zydis.re/

works (e.g., [29, 37]) also ignore the implicit data flow to avoid over-tainting.

Some library functions are hooked and analyzed by our system. The analysis engine hooks `open()`, `close()`, and `read()` in the glibc library to add taint tags when the program opens and reads sensitive files. The analysis engine also hooks `XRenderCompositeText()` in the xrender library to inspect whether the rendered text contains taint tags.

Shadow memory and registers. Each memory and register byte (both for general registers and SIMD registers) has a corresponding "shadow byte" maintained by the analysis engine. The "shadow byte" is actually a pointer to the head of a singly linked list of taint tags. A taint tag is a 32-bit integer, and so taint tags can track up to 4GB of sensitive data. Snowman uses a linear array to store the register "shadow bytes" for each thread. The data structure for the memory "shadow bytes" is a combination of a hash table and a linear array. Specifically, "shadow bytes" of a memory page are stored in a 4096-entry array. The location of this page array is saved in a hash table using the page address as the hash key. This hybrid design strikes a good balance between using a pure hash table and a pure linear array, where the former might trigger too many hash collisions while the latter consumes too much memory.

Taint tag allocation is a time-consuming operation and costs kernel memory. We implemented a *copy-on-write* taint propagation scheme to avoid unnecessary tag allocation. For the movement instructions, we only copy the pointer of the taint list from the source "shadow byte" to the destination "shadow byte" and increase the reference count of that pointer by one. For the arithmetic and logical instructions, where the source taint tags will be merged into the destination taint tags, we make a new copy of the taint list from the destination "shadow byte" if that list is also referenced elsewhere and then merge the source taint tags.

We also implemented a *garbage collection* scheme to free memory used to track already-leaked sensitive bytes. In our system, if a sensitive byte is leaked, the leakage count is increased by one. Future leakages of that same byte don't leak any more information and so the system doesn't need to keep tracking it. Therefore, Snowman maintains a list of already leaked taint tags and periodically invokes garbage collection to remove those tags from the shadow memory and shadow registers.

Cache settings. As described in Sec. 3.3.3, we implemented a block cache and an instruction cache. The block cache is a direct-mapped cache that has only one element in each cache entry. If a new basic block is mapped to the same entry as an old one, the old cache entry will be replaced. Since a program usually executes a relatively small number of basic blocks, this direct-mapped cache worked well in practice. Cache collisions are more frequent in the instruction cache, and so we implemented it as a two-way set associative cache (two elements in one entry). If a cache collision happens, the least recently used cache block will be replaced. To ensure the correctness of the cached data, we don't cache instructions and basic blocks if they are in a writable and executable page.

Control transfers. The analysis engine needs to take control from the replica at the right times to do taint analysis (see Fig. 2). To make the replica run in single-step mode, we set the TF bit in `eflags` register. We use the x86 debug register to set breakpoints at basic block boundaries. We add hooks in the debug trap handler (`do_debug()`) and the page fault handler (`do_page_fault()`) to transfer control to the analysis engine. Since the replication engine also sets single-step mode for some of its replay operations, we maintain an internal state to indicate to the analysis engine to transfer control to the replication engine instead of directly to the replica.

4 EVALUATION

In this section, we focus on evaluating Snowman's performance by measuring the reaction time of various GUI programs to user actions. We also evaluate Snowman's capability of differentiating the data-leakage patterns of malicious insiders from those of normal users.

GUI programs. We selected three typical and widely used GUI programs—a word processor, a spreadsheet, and a code editor—for our evaluations. The code editor is Gedit[2] (v3.18.3), which is pre-installed in many Linux distributions and has common features like syntax highlighting and word completion. The word processor and spreadsheet program are from LibreOffice[3] (v5.4), which is an open-sourced office suite (comparable to Microsoft office) and has a large user base. In particular, LibreOffice is a fairly complicated multi-threaded program that has 9 million lines of code (including C++, Java, and Python components)[4]. We believe testing Snowman with LibreOffice would make a comprehensive validation of our design and implementation.

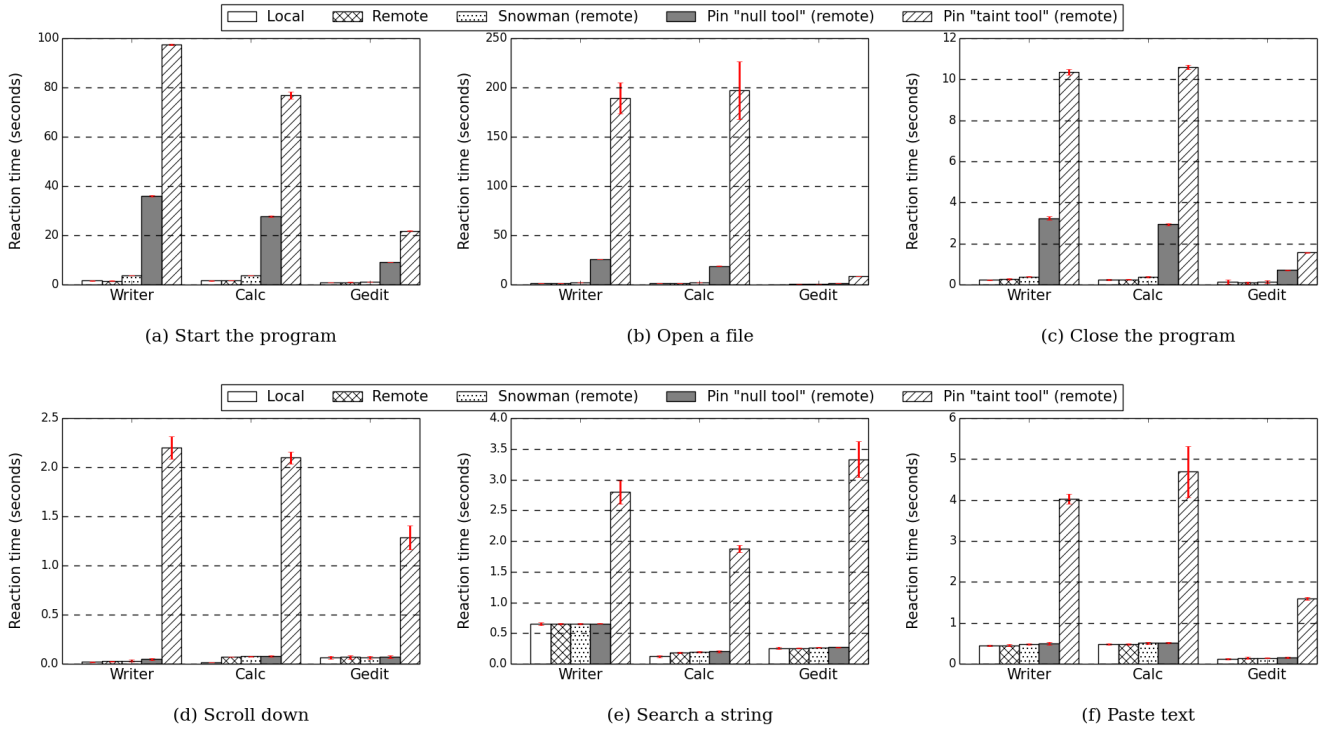
Environment. In the remote-access scenarios, the GUI programs ran in a Linux server that installed Snowman with the customized v4.11.12 kernel. We interacted with the GUI programs using a client machine running Ubuntu 16.04 with the v4.15.0 kernel. The client took mouse and keyboard inputs and sent the inputs to the server. The server did computation, generated graphical outputs, and sent the outputs to the client. The inputs and outputs were exchanged through the X11 protocol[5], which is the basic component of the Linux GUI framework, via TCP connections. The client machine was equipped with a 2-core 3GHz CPU and 4GiB of memory. The server machine was equipped with a 4-core 3.5GHz CPU and 8GiB of memory. The client and server were connected by 1Gbps Ethernet links in a local area network.

4.1 Performance for benign users

A core indicator of the GUI program's performance is its reaction time to user actions. To accurately measure the reaction time, we used Wireshark[6] to monitor the X11 packets passed through the TCP socket in the client machine. The reaction time was calculated as the difference between the departure time of the first user input packet and the arrival time of the last graphical output packet triggered by the user action.

We measured the reaction time of various actions performed on LibreOffice Writer (the word processor), LibreOffice Calc (the

[2] https://wiki.gnome.org/Apps/Gedit
[3] https://www.libreoffice.org/
[4] https://www.openhub.net/p/libreoffice/analyses/latest/languages_summary
[5] https://www.x.org/
[6] https://www.wireshark.org/



(a) Start the program (b) Open a file (c) Close the program

(d) Scroll down (e) Search a string (f) Paste text

Figure 4: Reaction time for common user actions when running the programs locally 1) in the client machine, remotely 2) in the server machine without instrumentation, 3) monitored by Snowman, 4) by the Pin "null tool", or 5) by the Pin "taint tool"

spreadsheet), and Gedit (the code editor). Each measurement was repeated 10 times. The actions we tested are the following.

Start the program. Each program was started by entering a command in the terminal. The reaction time was measured as the duration between the command key press and the display of the first window of the program on the screen.

Open a file. We opened a 46KiB text file in Writer, a 25KiB spreadsheet in Calc, and a 88KiB source code file in Gedit. The reaction time was measured as the duration between the open button click and the rendering of the full text on the screen.

Close the program. We exited each program by closing its first window. The reaction time was measured as the duration between the close button click and the release of all graphical resources by the program.

Scroll down. For each program, we clicked the scroll bar once to scroll down the window by one page. The reaction time was measured as the duration between the scroll bar click and the rendering of the new text on the screen.

Search a string. We searched a string in each program. The reaction time was measured as the duration between the search button click and the string being located on the screen.

Paste text. We pasted a 3984-character sentence in Writer, a 47-by-14 spreadsheet table in Calc, and a 13-line source code snippet in Gedit. The reaction time was measured as the duration between the paste button click and the rendering of the pasted text on the screen.

These actions were tested in five settings. In the first setting, the GUI program ran locally in the client machine, which is the normal setting without data protection from remote-only access. To assist reaction-time measurement, the X11 packets were transmitted through local TCP sockets. In the other four settings, the GUI program ran on the server machine, and the user interacted with the program through the client machine. Among these four settings, the first one ran the program natively without instrumentation; the second one ran the program under the protection of Snowman; the third one ran the program under the "null tool" of Pin [32] (v3.6); and the last one ran the program under the "taint tool" of Pin (v3.6) with the bytes read from the opened file marked as tainted. The Pin "null tool" does the minimal amount of instrumentation to maintain supervised execution of the program. The Pin "taint tool" conducts multi-label taint tracking and employs the same set of taint analysis rules as Snowman. It was implemented by us and was used to debug and validate the implementation of Snowman's taint analysis engine.

Fig. 4 shows the average reaction time of various actions in different settings, where the error bar represents the standard deviation. Overall, the reaction time of the actions in Snowman was 0.92× to 2.41× the reaction time of those actions in the remote-only setting without any instrumentation. Among these actions, opening a file, starting and closing the program have relatively large

overhead (1.16× to 2.41×) because Snowman needs to record the non-deterministic inputs and take extra steps to set up and tear down the environment for recording. However, we don't expect this would have significant impact on user experience since these actions are not frequently triggered by the user in typical workloads. The reaction time of other actions in Snowman are comparable to those in the remote-only setting (0.92× to 1.19×).

Additionally, Snowman performs better than the Pin "null tool" and "taint tool" in all tests. Compared with the remote-only setting without any instrumentation, the reaction time overhead is 1.05× to 23.07× in the Pin "null tool" and 4.3× to 133.1× in the Pin "taint tool". The actions triggering taint propagation (opening a file, scrolling down, searching a string, and pasting text) have huge overhead in the Pin "taint tool". We don't claim this multi-label taint tracking tool implemented by us is the best possible implementation. But we expect other similar tools would have similarly considerable overhead because the taint analysis routines are inlined with the normal program code by the Pin instrumentation. Besides, considering that the "null tool" doesn't implement any instrumentation for taint analysis, which represents a lower bound for taint analysis approaches implemented with Pin, Snowman should perform better than any Pin-based taint analysis tools.

4.2 Data exfiltration detection

Snowman aims to detect data exfiltration by monitoring the amount of sensitive data leaked to the user. Here we describe our evaluation of its efficacy in this regard, using the same applications as used for the performance evaluation for benign users in Sec. 4.1. We chose a primitive demonstration of the detection capabilities of Snowman: we simulated a "typical" normal user session and a malicious user session for each GUI program, and then showed that the leakage profiles of the two sessions as observed by Snowman could be statistically differentiated with overwhelming ease (and the speed with which this differentiation could occur, etc.). We designed the "normal" sessions based primarily on their representation in publicly available resources (see below), so that readers can easily assess the nature of activities in each, should they so choose. Moreover, these sessions were performed without undue delay or extra "thinking time," so as to simulate a more rapid leakage of data—and so, presumably, yielding a reasonably conservative evaluation. We discuss the settings and results of this evaluation in this section.

4.2.1 Settings. In our evaluations, the GUI program ran in the remote server and we interacted with the program in the client machine. Snowman maintained a replica of the program and conducted taint analysis on the replica to measure leakage.

LibreOffice Writer sessions. In the normal session, we formatted an ebook document by following the instructions from the Kindle ebook formatting guide[7]. The document we used was the first three chapters of the Python tutorial[8] with all formatting removed. We started the session by opening the document. Following the guide, we restored the format of the original tutorial by changing fonts of the section titles, inserting hyperlinks, adding footnotes and page numbers, and creating a table of contents. In the malicious session, we opened the same document, quickly scrolled down the document, and physically took pictures of all the pages.

LibreOffice Calc sessions. In the normal session, we did calculations on a spreadsheet containing employment and salary information. The spreadsheet was created with an online template[9] and filled with synthetic data (100 rows and 36 columns). Throughout the session, we calculated the number of employees taking more than two days off, the average medical expenses, and the total basic salary by using the built-in functions from Calc. In the malicious session, we quickly scanned the whole spreadsheet and took pictures of all the rows and columns.

Gedit sessions. In the normal session, we edited a C file to finish an assignment from the MIT Operating System Engineering class. We implemented the `env_init()` and `env_setup_vm()` functions by editing the `env.c` file, as required by exercise 2 of lab 3.[10] We also opened the `env.h` file to reference the related data structures. To best enable repeatability, we simply followed the solution from a github repository.[11] In the malicious session, we opened the `env.c` and `init.c` files, and took pictures of all the file contents.

4.2.2 Leakage detection. Snowman records various events of the monitored GUI program as described in Sec. 3.2, as well as the timestamp of each event. So, when Snowman detects new leakage via its taint-tracking in the replica, it can report the time of that leakage event from the original execution.

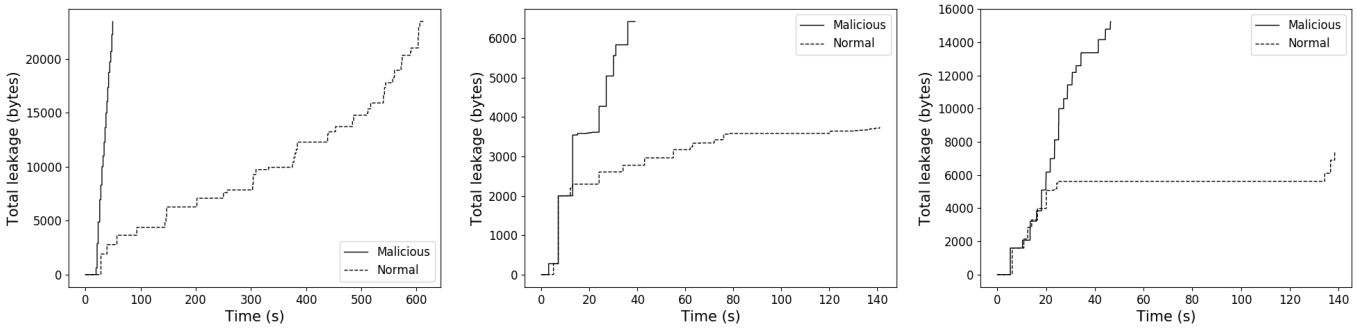
Fig. 5 reports the total leakage from the sensitive files over time, as detected by Snowman, in the normal and malicious session of each program. The x-axis is the time of the original user session. The y-axis is the total leakage. As can be seen there, in all three malicious sessions, the sensitive bytes were leaked out within one minute. In the normal sessions, the leakage occurred at a slower speed. Additionally, the normal sessions of the Calc and Gedit programs leaked only a part of the file contents.

We applied a statistical analysis to test whether the malicious session and the normal session can be easily differentiated. We used the logrank test [10], which is usually applied to compare the survival experience of two groups of patients. We treated the new leakage of a sensitive byte as analogous to the death event of a patient in this analysis. With the collected data, we calculated the time intervals between leakages and so the frequencies of data leakages. We subjected the time intervals from the malicious session and the normal session of each program to the logrank test. The p-value calculated from the Writer, Calc, and Gedit sessions is 0, 9.925×10^{-262}, and 9.157×10^{-168}, respectively. Thus, for each program, we can safely reject the null hypothesis that the leakage of the malicious session and the normal session are the same.

4.2.3 Detection delay. Although the taint analysis conducted by Snowman doesn't affect the program execution with which the user interacts, it slows down the execution of the replica and adds delay to the detection of the leakage event.

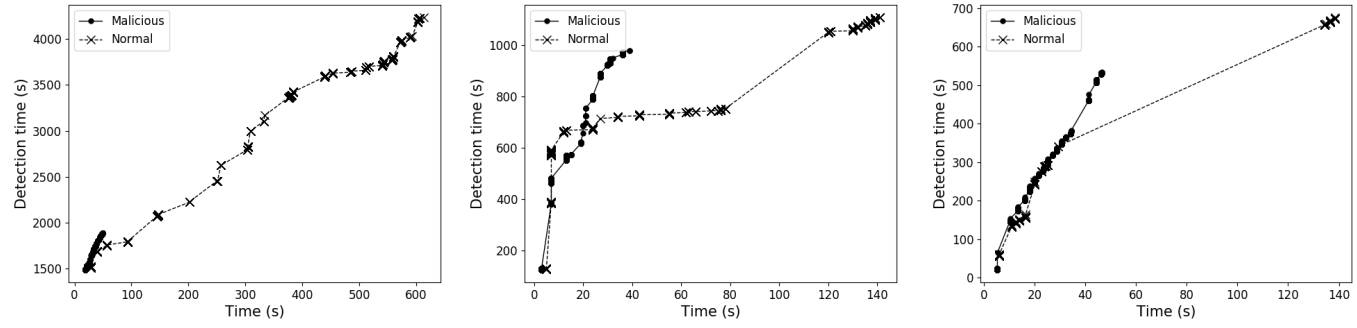
Fig. 6 plots the time at which the replica detected each new leakage, as a function of the time that the leakage actually occurred. In

[7] https://kdp.amazon.com/en_US/help/topic/G200645680 (This guide is based on Microsoft Word but we can find the same functionalities in LibreOffice Writer.)
[8] https://docs.python.org/3/tutorial/
[9] https://exceldatapro.com/download-salary-sheet-template/
[10] https://pdos.csail.mit.edu/6.828/2018/labs/lab3/
[11] https://github.com/Babtsov/jos/tree/lab3



(a) Writer (b) Calc (c) Gedit

Figure 5: The amount of leakage in normal and malicious sessions, as a function of time



(a) Writer (b) Calc (c) Gedit

Figure 6: The time at which the Snowman replica detected each new leakage, as a function of the time (in the user-facing execution) that the leakage originally occurred

the normal sessions, the Snowman replica detected the last leakage event with a lag of 6.9×, 7.8×, and 4.9× for Writer, Calc, and Gedit, respectively, behind when that leakage event occurred. In the malicious sessions, detection of the last leakage event lagged by 38.2×, 25.1×, and 11.5× for Writer, Calc, and Gedit, respectively. Gedit has the smallest lag, presumably because it has simpler code logic and takes less CPU cycles to process user requests. For all three programs, the malicious sessions lag more than normal ones, since the malicious user sent requests to the program at a higher frequency, leaving the replica fewer idle cycles to catch up.

We also want to figure out how quickly Snowman can differentiate the malicious session from the normal one. We adopted the following procedure to answer that question. In the timeline of the replica's execution, the leakage events from the malicious session, as detected by Snowman, were added to one dataset, and the leakage events from the normal session were added to another. These two datasets were updated at the end of each second *of the replica's execution time*. We started to apply the logrank test (the same as described in Sec. 4.2.2) on the updated datasets after both datasets had more than 1000 leakage events. We stopped the procedure when the calculated p-value dropped below 0.05. The time at which we stopped is reported as the earliest time Snowman could detect the malicious session (using the normal session as a "typical" baseline). The earliest detection times were 1522s, 574s, and 276s for Writer, Calc, and Gedit, respectively.

5 DISCUSSION

Taint analysis accuracy. Currently, we have implemented taint analysis rules for only a subset of x86-64 instructions in Snowman. This subset includes every instruction that explicitly propagates taint tags during the execution of the three tested programs (LibreOffice Writer, LibreOffice Calc, and Gedit). To capture this subset of instructions, we added an assertion in the taint analysis engine, which gives an alert if an executed instruction contains tainted operands but no taint analysis rule was implemented for that instruction. For other programs, we can take the same approach to select instructions for taint analysis.

Taint propagations caused by implicit data flows are mostly ignored by Snowman. Implicit data flows can exist in branch instructions where the branch condition variable decides which value will be saved to a given variable. There are also implicit data flows in pointer dereferences. In some cases, the data value in a memory location is decided by the value of the memory address, such as table lookups. Ignoring these implicit data flows can cause false

negatives in the leakage detection. However, enabling taint analysis for implicit data flows leads to false positives and taint explosion [45], and for this reason has been excluded in many prior tools (e.g., [29, 37, 43]).

To reduce false negatives caused by table lookups, we manually identified functions that might propagate taint tags through table lookups and implemented taint analysis rules based on these functions' semantics (see Sec. 3.3.2). Although we only hooked and analyzed the `pango_shape_full()` function in our current implementation, other functions can be analyzed in the same way as soon as they are identified.

Replication delay. The replica has to trap into the kernel to be analyzed by the taint analysis engine. This trap happens at least once per basic block and can increase to once per instruction if the basic block contains tainted operands. The frequent context switches between the user and kernel modes add significant overhead. We chose to implement the taint analysis engine in the kernel to avoid adding extra instructions into the replica. The benefit of doing so is that the replication engine can then determine when to inject the asynchronous signals and scheduling events by using the CPU retired conditional branch counter to measure the progress of the replica. In theory it should be possible to implement taint analysis in user space by instrumenting the binary and adjusting the measurements of the replica's execution accordingly.

Due to the amplified execution time of the replica, the malicious user might intentionally trigger expensive operations to increase the delay by which replica processing lags behind the original, in order to "buy time" for copying data. To detect such attacks, we could build a model of normal lag, i.e., a profile based on the lag during normal user sessions. If the lag during a user session deviates substantially from that profile, Snowman can raise an alert of potential malicious behavior, or alternatively slow down the user-facing execution. It is also possible to run the replica in a machine with higher CPU clock rate than the one where the original execution runs, so the replica can better keep up with the original execution.

Quick leakage estimation. Snowman's accurate leakage tracking could be augmented with a much quicker method of estimating the leakage based on examining the GUI traffic between the user-facing execution and the thin client. Ideally this estimator would quickly approximate the leakage with good recall and reasonable precision, to be corroborated (or corrected) by the replica when its analysis is complete. In doing so, Snowman could be made even more responsive to data exfiltration attempts. However, to tolerate false alarms by the estimator, a response to an estimator-based alarm might be to only slow down the user-facing execution, for example, until the taint-tracking replica catches up.

From differentiation to anomaly detection. We showed in Sec. 4.2 that it was trivial to statistically distinguish our own sessions of normal activity from ones in which we simply paged through files and photographed them, based on the leakage patterns determined by Snowman. While these tests provide strong evidence that detecting theft (as long as it is sufficiently aggressive) on the basis of a leakage profile is possible, the dearth of datasets characterizing the leakage patterns of either normal or theft-oriented usage of the applications tested there renders it impossible to properly evaluate an anomaly detection methodology based on Snowman. Moreover, since a data-leakage profile during normal use might be highly dependent on the expertise of the user and the type of file being accessed, there may not be a one-size-fits-all detector; rather, leakage models created per user, per file, or at least per organization might be more appropriate. Of course, a purely volume-based detector could presumably fail to detect data exfiltration performed very slowly, at a speed similar to normal usage. For such cases, analyzing the exact *order* in which bytes are leaked—which Snowman also provides—might be necessary. Still, we believe that even our primitive studies already provide a strong basis to motivate the further study of GUI leakage measurement as enabled by Snowman.

6 CONCLUSION

In this paper we presented Snowman, which aims to deter data theft by malicious insiders through strong data isolation and fine-grained data monitoring. In Snowman, a user is restricted to accessing sensitive data only remotely on a trusted server, via the GUI presented by the application to a thin client. In the server, Snowman detects data theft by monitoring the number of sensitive bytes leaked to the user. Maintaining good performance for normal usage of the monitored program while accurately monitoring for data theft is challenging. Snowman addresses this problem by replicating the execution of the program alongside its original execution and conducting multi-label taint analysis on the replicated execution. Our implementation of Snowman works on unmodified Linux binaries and off-the-shelf hardware without assuming that the replicated application is race-free (unlike some previous replication solutions). Our evaluations show that Snowman adds only moderate overhead for common user actions and, thanks to several novel optimizations, is far more efficient than, e.g., Pin-based taint analysis solutions. We also demonstrated that the data-leakage patterns of sufficiently aggressive malicious insiders can be leveraged to easily distinguish them from normal ones.

ACKNOWLEDGMENTS

Research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-17-1-0370. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Citrix virtual apps and desktops. https://www.citrix.com/products/citrix-virtual-apps-and-desktops/. Accessed: 13 May 2019.
[2] National insider threat task force. https://www.dni.gov/index.php/ncsc-how-we-work/ncsc-nittf. Accessed: 12 May 2019.
[3] Osaka Gas provides secure application availability with XenDesktop. https://www.citrix.com/customers/osaka_gas_en.html. Accessed: 13 May 2019.
[4] R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: A virtual display architecture for thin-client computing. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, page 277–290, 2005.
[5] C. Basile, Z. Kalbarczyk, and R. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *2003 International Conference on Dependable Systems and Networks*, pages 149–158, 2003.

[6] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 448–465, 2006.

[7] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, 2010.

[8] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In *9th USENIX Conference on Operating Systems Design and Implementation*, pages 177–191, 2010.

[9] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *2nd International Conference on Virtual Execution Environments*, pages 154–163, 2006.

[10] J. M. Bland and D. G. Altman. The logrank test. *BMJ*, 328:1073, Apr. 2004.

[11] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *14th International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2011.

[12] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr. Abstractions for practical virtual machine replay. In *12th ACM International Conference on Virtual Execution Environments*, pages 93–106, 2016.

[13] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, pages 15:1–15:58, July 2009.

[14] D. Chen, J.-M. Odobez, and H. Bourlard. Text detection and recognition in images and video frames. *Pattern Recognition*, 37(3):595 – 608, 2004.

[15] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, pages 22–22, 2004.

[16] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[17] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232, Dec 2004.

[18] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 525–540, 2014.

[19] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared-memory multiprocessing. *IEEE Micro*, pages 40–49, 2010.

[20] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering with panda. In *5th Program Protection and Reverse Engineering Workshop*, pages 4:1–4:11, 2015.

[21] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th USENIX Symposium on Operating Systems Design and implementation*, pages 211–224, 2002.

[22] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *4th ACM International Conference on Virtual Execution Environments*, pages 121–130, 2008.

[23] D. Gugelmann, D. Sommer, V. Lenders, M. Happe, and L. Vanbever. Screen watermarking for data theft investigation and attribution. In *10th International Conference on Cyber Conflict*, May 2018.

[24] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *1st ACM European Conference on Computer Systems*, pages 29–41, 2006.

[25] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *2008 International Symposium on Computer Architecture*, pages 265–276, June 2008.

[26] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 235–246, 2013.

[27] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee. RAIN: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM Conference on Computer and Communications Security*, pages 377–390, 2017.

[28] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 105–114, June 2009.

[29] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *8th ACM International Conference on Virtual Execution Environments*, Mar. 2012.

[30] J. Laurikkala, M. Juhola, and E. Kentala. Informal identification of outliers in medical data. *5th International Workshop on Intelligent Data Analysis in Medicine and Pharmacology*, July 2000.

[31] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *2001 IEEE Symposium on Security and Privacy*, pages 130–143, May 2001.

[32] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[33] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 308–319, 2016.

[34] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 65–80, 2015.

[35] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[36] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture*, pages 284–295, 2005.

[37] J. Newsome and D. Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *ISOC Network and Distributed System Security Symposium*, Feb. 2005.

[38] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *9th ACM International Conference on Knowledge Discovery and Data Mining*, pages 631–636, 2003.

[39] B. Obama. Presidential memorandum: National insider threat policy and minimum standards for executive branch insider threat programs. https://www.dni.gov/index.php/ic-legal-reference-book/presidential-memorandum-nitp-minimum-standards-for-insider-threat-program, 21 Nov. 2012.

[40] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. In *USENIX Annual Technical Conference*, pages 377–389, July 2017.

[41] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *8th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 2–11, 2010.

[42] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *ACM European Conference on Computer Systems*, pages 15–27, 2006.

[43] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Dec. 2006.

[44] S. Shi and C.-H. Hsu. A survey of interactive remote rendering systems. *ACM Comput. Surv.*, 47(4), May 2015.

[45] A. Slowinska and H. Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 61–74, 2009.

[46] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Symposium on Fault Tolerant Computing*, pages 250–259, 1996.

[47] E. Snell. 58% of healthcare PHI data breaches caused by insiders. https://healthitsecurity.com/news/58-of-healthcare-phi-data-breaches-caused-by-insiders, 5 Mar. 2018.

[48] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[49] P. Szoldra. This is everything Edward Snowden revealed in one year of unprecedented top-secret leaks. https://www.businessinsider.com/snowden-leaks-timeline-2016-9, 16 Sept. 2016.

[50] J. D. Valois. *Lock-free Data Structures*. PhD thesis, 1996. Rensselaer Polytechnic Institute.

[51] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *14th IEEE International Symposium on High Performance Computer Architecture*, pages 173–184, 2008.

[52] P. Walker. Bradley Manning trial: what we know from the leaked WikiLeaks documents. https://www.theguardian.com/world/2013/jul/30/bradley-manning-wikileaks-revelations, 30 July 2013.

[53] G. Williams, R. Baxter, H. He, S. Hawkins, and L. Gu. A comparative study of RNN for outlier detection in data mining. In *2002 IEEE International Conference on Data Mining*, pages 709–712, Dec. 2002.

[54] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari. The performance of remote display mechanisms for thin-client computing. In *USENIX Annual Technical Conference*, pages 131–146, 2002.