TxBox: Building Secure, Efficient Sandboxes with System Transactions

Suman Jana
The University of Texas at Austin
suman@cs.utexas.edu

Donald E. Porter
Stony Brook University
porter@cs.sunysb.edu

Vitaly Shmatikov
The University of Texas at Austin
shmat@cs.utexas.edu

***Abstract*—TxBox is a new system for sandboxing untrusted applications. It speculatively executes the application in a system transaction, allowing security checks to be parallelized and yielding significant performance gains for techniques such as on-access anti-virus scanning. TxBox is not vulnerable to TOCTTOU attacks and incorrect mirroring of kernel state. Furthermore, TxBox supports automatic recovery: if a violation is detected, the sandboxed program is terminated and all of its effects on the host are rolled back. This enables effective enforcement of security policies that span multiple system calls.**

I. Introduction

Secure execution and confinement of untrusted applications is a long-standing problem in system security [35]. We present a new approach to constructing secure, efficient sandboxes based on *system transactions*.[1] In general, system transactions guarantee that a sequence of updates to system resources satisfies atomicity, consistency, isolation, and durability (ACID). Transactions are increasingly popular for managing concurrency in modern operating systems. Our prototype system, TxBox, uses transactions for (1) speculative execution of untrusted applications, (2) uncircumventable enforcement of system-call policies, and (3) automatic recovery from the effects of malicious execution.

TxBox consists of a relatively simple, policy-agnostic security monitor running in the OS kernel and a user-level policy manager. The separation of the security policy and the enforcement mechanism facilitates management of policies. Before the transaction associated with a sandboxed program commits, the monitor inspects its effects on the system (conveniently assembled in the transaction's *workset* and its system-call log) and checks if they satisfy the policy. If so, the transaction is committed and updates become visible to the rest of the system. Otherwise, the transaction is aborted and the system is restored to a good state. TxBox is suitable for sandboxing "one-shot" execution of unknown, untrusted programs, as well as for model-based enforcement of system-call behavior of known benign programs.

Uncircumventability. TxBox cannot be circumvented by a sandboxed process. Its kernel-based enforcement mechanism prevents exploitation of incorrect mirroring of the kernel state, TOCTTOU races, and/or other semantic gaps between the security monitor and the OS [21, 58]. Unlike any monitor that infers effects on the OS from the periphery of the kernel, the effects analyzed by the TxBox monitor when making security decisions are *exactly* the effects that would take place if execution is permitted.

Recoverability. Existing system-call monitors must allow or deny every system call made by the untrusted program before it executes. Once the call is permitted to execute, there is no way to recover. Therefore, they must be able to detect the very first sign of misbehavior since the effects of a malicious execution cannot be "undone." By contrast, TxBox executes untrusted programs speculatively, inside a transaction. If the monitor determines later that the program has violated a security policy, it aborts the transaction and the system is automatically rolled back to a benign state. All changes made by the violating program to the file system effectively disappear, child processes are stopped, and buffered local inter-process communication is canceled, leaving concurrent updates made by other programs undisturbed.

To illustrate the benefits of recoverability, Section VI shows how TxBox can restore the original state of local files if an untrusted program (*e.g.*, a multimedia converter) attempts to violate the sandboxing policy.

Performance. On realistic workloads, the performance overhead of TxBox is less than 20% including the cost of supporting transactions and less than 5% over untrusted execution in a transactional OS. Note that there is a compelling secular (*i.e.*, unrelated to security) reason for supporting system transactions, namely, managing concurrency.

TxBox can take advantage of multi-core processors. In Section VI, we show how to use TxBox to execute an anti-virus scan in parallel with the speculative execution of an untrusted program. This makes on-access scanning practical for routine use in production systems.

Expressive policies. TxBox can enforce a rich class of practical security policies. This includes all policies supported by system-call interposition tools such as Systrace [49], system-call policies for malware detection [7, 33, 36], model-based system-call automata [26, 52, 56], data-flow policies on system-call arguments [5], and, in general, any policy expressible as a security automaton [16] over system calls and system resources. For system-call policies,

[1] System transactions are not transactional memory. System transactions deal with accesses by a user process to system resources such as files and pipes, not with memory accesses.

TxBox provides uncircumventability, recoverability, and parallelization of policy checking.

Unlike system-call monitors, TxBox also supports security policies on lower-level objects such as files and sockets. This includes access-control policies on system resources, *e.g.*, blacklists and whitelists of files and directories. Because system transactions span fork and exec, TxBox enables a simple form of information-flow control by tracking not just the resources accessed by the monitored process, but also those accessed by its children. For these policies, TxBox provides uncircumventability and recoverability.

Semantic fidelity. By analogy with hardware transactional memory, a very different mechanism which implements a similar abstraction, TxBox can be "deconstructed" into *grouping*, *rollback*, *access summary*, and *access check* components [30]. Grouping enables the security monitor to observe the cumulative effects of an entire sequence of system calls before making the security decision. Rollback enables recovery from the effects of malicious execution. Access summary assembles all of the program's effects at the ideal layer of abstraction for making security decisions: changes to kernel data structures, updates to file contents, system calls and their arguments, *etc.* Access checks prevent malicious programs from affecting the rest of the system.

Implementation. Our prototype implementation is based on TxOS [48], a version of commodity Linux with support for system transactions. The main differences are as follows. TxBox is a sandbox architecture based on system transactions; TxOS provides a concrete implementation of system transactions. The focus of TxOS is on managing concurrency. Trusted applications can take advantage of system transactions to prevent TOCTTOU conditions such as access/open file-system races, but TxOS *per se* does not deal with sandboxing untrusted code or preventing attacks on the host machine by malicious applications.

Organization of the paper. In Section II, we describe the challenges of building a robust sandbox and how they are addressed by TxBox. Related work is surveyed in Section III. System transactions are introduced in Section IV. Design and implementation of TxBox are described in Section V and evaluated in Section VI. We analyze the limitations of our approach in Section VII and conclude in Section VIII.

II. Building a better sandbox

A. Understanding behavior of the sandboxed program

A malicious program can attack its host machine in a variety of ways. To block malicious behavior, the sandbox must be observing the program at the right level of abstraction. For example, assembly-level inline reference monitors can enforce properties such as control-flow integrity [1], but preventing a program from opening a network connection after reading private files requires visibility into OS abstractions. Reconstructing OS-level behavior from a hardware-level view of the system requires non-trivial effort [25].

Even if the security monitor is observing the application's behavior in the OS, many straightforward observation points are prone to subtle errors that can compromise security. Consider a naïve user visiting a malicious website. The site appears to host a video file, but asks the user to install a codec to play it. The "codec" contains malicious code which steals the user's private data and sends it over the network. This particular attack can be prevented if the codec is executed in a sandbox which enforces a simple policy: "an untrusted program should not read the user's private files." At first glance, this policy can be enforced by intercepting all open system calls and checking whether the argument is one of the forbidden files. Unfortunately, the system-call API in modern OSes such as Linux is extremely rich in functionality, giving the attacker many ways to achieve his goal. For example, the malicious codec can create a soft link pointing to a public file and then change it so that it points to a private file. Therefore, the system-call monitor must also check link system calls, and so on. To enforce even simple policies, a system-call monitor must know all possible system-call sequences that can lead to a violation and the monitor must check every call which might belong to such a sequence. This is not only difficult to implement correctly but can also degrade performance.

Even if the system-call monitor correctly tracks all relevant system calls, it must completely understand the effects of the sandboxed program's behavior on the host system: which files have been accessed, what is the cumulative effect of several system calls, *etc.* This is a notoriously difficult problem. System-call interposition tools have been plagued by TOCTTOU (time-of-check-to-time-of-use) vulnerabilities [21, 58] which enable malicious programs to exploit incorrect mirroring of kernel state inside the security monitor and discrepancies between system calls as observed by the interposition tool and as executed by the kernel.

By design, TxBox is immune to TOCTTOU attacks. Sandboxed processes run inside separate transactions, so changes made by one of them to their shared state will not be visible to the other until the transaction commits. If a sandboxed process spawns a child, both run inside the same transaction and their updates to system state are reflected in the same transactional workset. Policy enforcement in TxBox is performed by inspecting objects in the workset and thus cannot be evaded by splitting updates between the parent and the child.

Rather than construct an error-prone mapping of system-call arguments or hardware-level events to OS state changes, TxBox directly inspects pending changes to kernel state made by the transaction wrapping the sandboxed process and its children. This makes it easy to enforce whitelist and blacklist access-control policies. Files and directories are represented by their inodes (in the case of multiple

file systems, also by superblock identifiers). The sandboxed process cannot evade enforcement by switching mappings between file names and inodes.

B. Recovering when a violation is detected

When a conventional sandbox detects that the confined program is trying to escape the sandbox (*e.g.*, make a forbidden system call or access a forbidden resource), it can block that particular action but cannot undo the program's previous effects on the host. If the policy is incomplete or if a violation can be detected only after multiple system calls, the damage may already have been done.

For example, consider a codec downloaded by the user from an untrusted website. A reasonable sandboxing policy may ensure that the codec can only write to files in the user's video directory. Suppose the codec is malicious and, after damaging or infecting the user's videos, attempts to escape the sandbox by connecting to the network. The sandbox detects this, denies the call, and terminates the codec, but the user's video files have already been corrupted.

When TxBox detects a policy violation, the transaction is aborted and the system automatically reverts to a good local state (except for the effects of previously allowed external I/O in certain enforcement regimes—see Section V-D). The program's effects on the host are undone, while concurrent updates performed by other processes are left in place. If the policy is incomplete, as long as the sandboxed program attempts to perform at least one of the forbidden actions, TxBox will roll back the effects of *all* of its actions.

C. Taking advantage of parallelism

It is difficult for a conventional sandbox based on system-call monitoring to take advantage of parallelism in modern multi-core processors. Whenever the sandboxed program makes a system call, its execution must be paused in order for the monitor to decide whether to allow or deny the call. Because of this, the sandboxed program and the monitor cannot be executed concurrently. This is also true for other security checks such as anti-virus scanning. An untrusted program must be scanned *before* it is executed because there is no way to undo its effects on the host if the scanner detects an infection after the program has been permitted to execute.

One way to balance security and performance is to first execute a copy of the untrusted code in a monitored sandbox and, if no problems are detected, execute it "natively" in the future. Unfortunately, it is difficult to make the monitoring transparent [22], and this approach is thus vulnerable to "split-personality" malware which behaves benignly if it is being observed and maliciously otherwise (see Section III).

TxBox can take advantage of parallelism because system transactions are a form of *speculative execution*. TxBox lets the sandboxed program run with close-to-native performance while performing security checks such as anti-virus scanning in parallel. If a violation is detected, all changes made by the program are discarded. Because only a single copy of the untrusted code is executed, split-personality malware may refuse to execute in TxBox, but the effects of malicious behavior do not enter into the system.

III. Related work

Speck. Nightingale et al. proposed a system called Speck [40], which uses a multi-core processor to speculatively execute the untrusted program while concurrently performing security checks on an instrumented copy on another core. To synchronize the copies, Speck records all non-deterministic system calls (*e.g.*, read) made by the instrumented copy and replays their outcome to the uninstrumented process.

Because security checks are not applied to the uninstrumented copy, Speck may be circumvented by "split-personality" malware which behaves differently in monitored and unmonitored environments. In general, any approach that involves running an instrumented and uninstrumented copies of the same code requires instrumentation to be completely transparent. Speck uses Pin, which is designed to be transparent to a well-behaved program. A malicious program, however, can detect Pin by checking if certain dynamically loaded libraries are present in its address space. Building an instrumentation system which is truly transparent against an actively malicious program is difficult.

Because instrumentation used by Speck is not transparent, a malicious program can take different paths depending on whether it has been instrumented or not. If the paths differ only on system calls Speck considers deterministic and the instrumented copy passes security checks, then the uninstrumented copy may behave maliciously without being detected. This is a TOCTTOU vulnerability. The problem is not the lack of transparency *per se* (we do not claim that TxBox is transparent), but the lack of transparency combined with concurrent execution of two copies of untrusted code enables "split-personality" malware to evade detection.

Presumably, Speck could be modified to log and replay *all* system calls, reducing the opportunities for the copies to deviate. Such frequent synchronization would dramatically reduce the exploitable parallelism and defeat the primary purpose of parallel security checking.

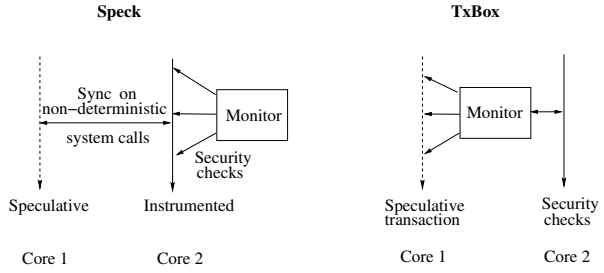

Figure 1. Comparison of TxBox and Speck architectures.

TxBox cannot be circumvented by "split-personality" malware. It speculatively executes a single copy of the untrusted program inside a transaction and all security checks are performed on that copy. Figure 1 shows the difference between TxBox and Speck. Unlike Speck, the TxBox security monitor has access to the transactional workset, which gathers all accesses to system objects by the untrusted program. Depending on the nature of the check, the monitor can request the kernel to store the necessary state information and access it in parallel with the sandboxed process (see Section VI). Unlike Speck, TxBox cannot enforce security policies which require information not readily available to the kernel (*e.g.*, data flows in application's memory).

Using transactions for security. Birgisson et al. present a reference monitor architecture which uses *Software Transactional Memory* (STM) to enforce security policies for multithreaded applications [6]. These policies deal with internal objects residing in application's memory. Harris and Peyton-Jones propose a framework for programmer-provided data invariants in the Haskell STM [29]. Chung et al. employ transactional memory in a thread-safe binary translation mechanism which they use to implement information-flow tracking in application's memory [8].

The key difference is that transactional memory enables transactional semantics for accesses to application's memory, while system transactions enable transactional semantics for accesses to system resources by a user process. STM-based systems are designed to enforce *application-specific* security policies, which are orthogonal to system-level policies and cannot be enforced by our kernel-level security monitor. For system-level policies, transactional memory is at the wrong level of abstraction. STM-based enforcement cannot span system calls and thus cannot protect access to system resources such as files and network sockets.

Sidiroglou and Keromytis instrument application code dealing with memory buffers to provide transaction-like semantics for detecting and recovering from buffer overflow attacks [54]. Locasto et al. use transactional execution for individual functions [38]. Unlike transactions internal to an application, system transactions in TxBox provide a general mechanism for inspecting and, if necessary, rolling back changes made by an application to OS state.

Clark et al. define a model for commercial security policies and suggest the use of well-formed transactions to preserve data integrity [10]. In TxBox, such restrictions can be enforced via a policy that controls access to sensitive data.

Vino OS [53] allows applications to load extensions into the kernel and uses transactions and software fault isolation to protect the kernel from buggy and malicious extensions. The goal is to restrict untrusted extensions to the same range of behavior as user-level programs and properly enforce standard access control on them. Transactions are used to recover shared kernel state (*e.g.*, release locks and free memory) after a misbehaving extension is removed and generally applied at the granularity of a single call into the extension module. By contrast, TxBox leverages the transaction's workset to inspect *cumulative* behavior of an untrusted application, assure semantic fidelity of policy checks, and enforce a wider range of security policies.

Sandboxes and system-call monitors. Prior sandbox architectures include kernel-based systems [3, 4, 11, 19] and system-call interposition tools [2, 27, 32, 49]. Interposition is typically implemented using kernel-mode system-call wrappers. In Section V-B, we show how their functionality can be easily emulated in TxBox.

Malicious programs can bypass wrapper-based enforcement by exploiting race conditions and incorrect replication of the OS state inside the security monitor (see [21, 58] and Section II-A). Ostia [24] and Plash [51] solve these issues by using delegation-based architectures. Rather than attempt to enforce access control on semantically murky system-call arguments, they restrict what file handles can be created in a sandboxed process and only allow operations via approved handles. Delegation-based architectures generally require little or no changes to the OS kernel, instead modifying `libc` to emulate forbidden API functions using approved functions. Because the file-handle-based API (*e.g.*, openat) in most Unix systems is incomplete and because emulating the Unix kernel API is inherently difficult, delegation-based systems are prone to subtle security bugs [45, 46].

TxBox occupies a different point in the design space of sandbox architectures. It requires more kernel changes than delegation-based sandboxes but does not need to solve the problem of accurately emulating the OS API.

Capsicum is a capability-based sandboxing system which adds new primitives to the UNIX API to support compartmentalization of applications [59]. The goals of Capsicum are orthogonal to TxBox. Its new API helps benign applications increase their trustworthiness, while TxBox sandboxes untrusted applications that use the standard Unix API.

Several sandboxes have been proposed for application plugins, especially for Web browsers. Vx32 employs binary translation and x86 segmentation [18], while Native Client requires the code to be recompiled to a restricted subset of the x86 ISA and also confines it using segmentation [62]. Xax places untrusted code in its own address space and restricts it to a small subset of system calls, enforced by system-call interposition [13]. The problem of protecting trusted code from untrusted code in the same address space is orthogonal to system-level sandboxing. System-level policies enforced by plugin sandboxes are typically simple and disallow access to nearly all system resources.

Sun et al. combine system-call interposition with a security monitor between the virtual file system (VFS) and the lower-level file system. This layer implements SEE, a simple, transactional file system enforcing one-way isolation

from an untrusted process [55]. SEE provides a speculative execution environment similar to TxBox but limited to the file system and network, while TxBox can also isolate system calls such as fork and signal.

Placing sandboxing hooks at the VFS interface is somewhat similar to placing them in the system call table. Both are appealing because they minimize OS changes and have a tractable surface area. Just as system-call interposition monitors ultimately struggle with problems such as TOCTTOU races, a monitor and transactional file system implemented below the VFS layer face challenges with insufficient hooks into higher-level functionality. For instance, the isolation mechanism of SEE is based on recording the timestamp of the first read; in the case of file-system metadata in an unmodified kernel, a low-level hook is typically called only if the data is not in a VFS cache. In the common case where the directory structure is cached, this timestamp-based conflict detection can have false negatives that violate isolation. The kernel-based isolation mechanism of TxBox helps avoid engineering and fidelity problems that are common in security monitors at the periphery of the OS.

System-call monitoring for intrusion detection in benign programs has been enhanced by using sequence relationships [31, 60], call-site [52] and call-stack information [17, 20], and system-call arguments [5]. Static analysis can be used to automatically extract system-call models from the program's code [26, 56]. This work can be viewed as a source of system-call policies for TxBox.

Virtual-machine-based monitors. Virtual machines (VM) enable external inspection of both the OS and applications. There is a large body of literature on using virtual machines for intrusion detection, honeypots, *etc.* Unlike TxBox, VM-based methods can potentially protect even against kernel-based malware [23], although there are many challenges: granularity of checking, reconstruction of system-level behavior from hardware-level events, merging committed state back into the system, and how to achieve close-to-native performance while performing frequent security checks.

While in theory all policies described in Section V-B can be enforced using a VM-based monitor, we are not aware of any existing monitor that can (1) isolate execution of a single untrusted user process in the guest OS, (2) detect when it is about to violate a security policy, (3) terminate the process and roll back its effects while leaving concurrent updates performed by other processes in place, and (4) impose minimal performance penalty on benign processes. For example, ReVirt, a system that can record and replay the VM state for analyzing intrusions [14], records events at the granularity of the entire virtual machine. This is too coarse for rolling back the effects of a single process.

Information-flow control. Kernel-based mandatory access control (MAC) systems such as SELinux [42] and AppArmor [41] can restrict an untrusted program to a subset of system resources, but the administrator must identify in advance which resources will be needed. This is non-trivial even for relatively simple applications, thus these systems work only for well-vetted applications and policies. Blacklist policies such as "untrusted programs can access any directory in the file system except /usr/private" can be cumbersome to formulate using either a file-system confinement mechanism such as chroot and jail, or a MAC system that requires explicit assignment of access-control labels. By contrast, TxBox enables easy configuration and enforcement of blacklist policies specifying only the resources that may *not* be accessed by an application.

Operating systems with decentralized information-flow control can enforce end-to-end access-control policies [15, 34, 63]. TxBox does not propagate access-control labels and thus is not able to enforce all policies supported by these systems, but is simpler to deploy.

IV. System transactions

System transactions are a programming abstraction that provides atomicity, consistency, isolation, and durability (ACID) properties for sequences of updates to system resources such as files, pipes, and signals. Informally, from the viewpoint of the system, either all updates are performed as an atomic sequence, or none are. The system always remains in a consistent state. If the transaction is *aborted* before it finishes, all intermediate updates are rolled back as if none of the actions inside the transaction had been executed.

When a process performs a "normal" call to the OS, the effects of this call—for an example, an update to the state of some system resource—become visible to other processes as soon as the OS kernel releases its lock on the resource. System transactions, on the other hand, enclose a code region into a logical unit called a *transaction*, which may span multiple system calls. All accesses to the system within a transaction are kept isolated and invisible to the rest of the system until the transaction *commits*. At the commit time, all actions performed by the process within the transaction are published *atomically* and become visible to other processes.

System transactions should not be confused with transactional memory [37]. The primary purpose of system transactions is to enable applications to express to the OS their consistency requirements for concurrent operations [48]; in this sense, they are akin to database transactions rather than transactional memory. In this paper, however, we use them for a very different purpose, to confine untrusted applications in an uncircumventable sandbox.

System transactions require OS support. Our prototype of TxBox is based on TxOS, an experimental modification of commodity Linux [48]. In TxOS, system transactions are part of the OS interface. An application starts a transaction by making a sys_xbegin system call and ends it by calling sys_xend; all calls between are performed as part of a

single transaction. TxOS allows both transactional and non-transactional system calls to access the same, shared system resources. The OS ensures that these accesses are correctly serialized and contention is arbitrated fairly.

To keep track of the system objects accessed by a transaction, TxOS maintains a transactional *workset*. It stores references to all kernel objects (inodes, *etc.*) for which the transaction has private, "shadow" copies. For fast commit, the workset is sorted by the kernel locking discipline. Each entry in the workset contains a pointer to the stable object, a pointer to the shadow copy, information about whether the object is read-only or read-write, and a set of type-specific methods (commit, abort, lock, unlock, release).

TxOS uses *eager* conflict detection. As soon as two isolated processes attempt a conflicting access to a resource, the OS rolls one back and the transaction is retried. Transactions are serializable: TxOS does not allow a process to make a shadow copy of potentially inconsistent state, and all conflicts are detected before a transaction is allowed to commit [47]. Under the default contention management policy, the losing process is suspended until the transaction that it lost to commits.

In TxBox, transactional worksets provide a convenient vantage point for the security monitor to inspect all potential effects of the sandboxed process on the system state: all files it accessed, all updates it intends to performs, *etc.* The monitor decides whether these updates satisfy the security policy and can be made visible to the rest of the system.

Support for system transactions comes at a modest performance cost. The average overhead in TxOS, at the scale of a single system call, is around 29% [48]. In Section VI, we show the performance overhead of TxBox for several individual system calls and application workloads.

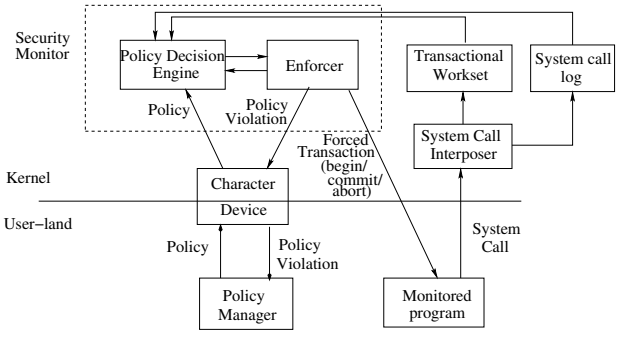

Figure 2. Overview of TxBox architecture

V. Design and implementation of TxBox

A. Architecture of TxBox

The architecture of TxBox is shown in Fig. 2. The two main parts are the *security monitor*, which runs in the OS kernel, and the *policy manager*, which runs in user space.

The system administrator uses the policy manager to define the policy as a set of regular expressions over system call names and arguments and system objects such as inodes and socket descriptors. Each regular expression is marked as either a whitelist or a blacklist, which specifies, respectively, the required or forbidden behavior of the sandboxed program. The policy may also specify system calls that should be denied while permitting the program to continue running.

The policy may specify *critical system calls*, which cause a trap into the security monitor when invoked by the sandboxed process. By default, critical calls always include all calls related to program termination (*e.g.*, exit and exit_group) and external I/O (*e.g.*, network and interprocess communication).

The policy manager compiles the policy and installs it in the security monitor. The monitor forces each untrusted user process to run inside a transaction and applies installed policies as described in Section V-C. TxBox depends on the OS to support transactional system calls (see Section IV); our prototype uses TxOS [48]. Because TxOS does not allow one process to put another process in a transactional mode, we modified the TxOS kernel so that the security monitor can force user processes to run inside transactions.

While the sandboxed process is running inside a transaction, TxBox maintains its *trace* consisting of (1) all system calls the process made, (2) their arguments, and (3) workset of the transaction, which contains references to all system objects affected by the process (see Section IV). Whenever the sandboxed process makes a critical system call—for example, terminates or attempts to perform external I/O—control is switched to the security monitor.

The monitor checks whether the trace of the process violates the installed policy or not. If the policy has been violated, the monitor can either roll back the transaction and kill the violating process, or else pause the process, generate a POLICY_VIOLATION event, and call the user-space policy manager for further investigation. The choice is a policy configuration parameter. The manager can perform additional checking (*e.g.*, use ptrace to attach and examine the memory of the sandboxed process) or request input from the human operator, then inform the monitor whether the sandboxed process should be killed and transaction rolled back. Transactions aborted due to a policy violation are prevented from automatically re-trying.

If the policy is not violated and the critical event that caused invocation of the monitor is process termination, the transaction is committed and all effects of the process's execution become visible to the rest of the system. The only other default critical events are external I/O. If a non-default call is listed as critical in the policy, the monitor executes the call and continues the current transaction after checking that the trace does not violate policy.

The trusted computing base of TxBox consists of the OS kernel and the user-space policy manager. In modern OSes,

the administrator (root) can insert and remove kernel modules. Security of TxBox only holds under the assumption that the adversary does not have root access to the host.

B. Security policies

Security policies are defined by the system administrator who uses the policy manager to install them in the security monitor, as described in Section V-C. Different policies may be associated with different user processes. The administrator can specify either the sandboxed process when installing a policy, or a path and a list of events. In the latter case, the policy manager will automatically associate the policy with any program residing on that path.

A TxBox policy is an ordered list with any number of primitives of one of three types: BLACKLIST, WHITELIST, DENY. BLACKLIST primitives specify forbidden behavior. WHITELIST primitives specify required behavior. DENY primitives are lists of system calls which should be denied if the sandboxed process tries to make them.

TxBox matches primitives to the program's trace in the order they are written. Once a violation is detected (the trace matches a BLACKLIST primitive or does *not* match a WHITELIST primitive), TxBox terminates the sandboxed program and rolls back all of its local changes without checking subsequent policy primitives.

BLACKLIST and WHITELIST primitives. These primitives are regular expressions over system-call names and arguments, or over system objects. TxBox supports two types of system objects in policies: inodes (of directories or files) and sockets. In policies, these are prefixed by 'I' and 'S', respectively. An inode object has two attributes: number and mode (*e.g.*, read or write). If multiple file systems are mounted at the same time, superblock identifiers can be used to distinguish files from different systems that happen to have the same inode number. The set of supported modes is determined by the underlying transactional OS which provides the worksets to TxBox. A socket object has two attributes: type (*e.g.*, INET or UNIX datagram) and either destination IP address (for INET sockets), or name (for UNIX datagram sockets). Using low-level kernel objects such as inodes rather than file names helps make security decisions faster because the TxBox security monitor can match policies against transactional worksets without extra lookups. In our current implementation, policies on objects (marked as WREGEX) use inode numbers, while system-call policies (marked as SREGEX) use file names.

If the policy involves a file name, the policy manager retrieves the corresponding inode number and substitutes it for the name before installing the policy. It also stores the original file name. If the name's inode mapping has changed before an existing policy is automatically installed for a new process, the policy is updated with the new inode number. Policies involving file names should block creation of soft and hard links, as shown below.

The TxBox policy syntax allows complex policies to be expressed in a modular manner. For example, suppose that a known execution profile of some program says that it should open a file in the '/tmp' directory other than 'secret' for reading. This can be enforced by the following policy, where '$(e)*$' means "match expression e any number of times."

```
BLACKLIST SREGEX *open:/tmp/secret:r*
BLACKLIST SREGEX *rename:/tmp/secret*
BLACKLIST SREGEX *symlink:/tmp/secret*
BLACKLIST SREGEX *link:/tmp/secret*
WHITELIST SREGEX (open:/tmp/*:r)*
```

Whitelist policies represent required behavior and can be used to sandbox programs for which a system-level model of correct behavior is available. Such models can be derived by profiling the program's execution or computed from the program's source or binary code using static analysis [26, 56]. Static models are conservative, thus any deviation from a model-based whitelist policy means that the sandboxed program is no longer executing the original code (typically, because of a code-injection attack) and should be terminated.

Table I shows some example BLACKLIST and WHITELIST policies. These policies are simple but can effectively sandbox untrusted, potentially malware-infected programs. For example, they can confine an untrusted file-format converter downloaded from the Internet to reading and writing files in a particular directory. In Section VI, we report the experimental results of sandboxing the FFmpeg multimedia converter with TxBox.

In general, TxBox can enforce any policy expressed as a (possibly non-deterministic) security automaton [16]. This includes policies designed to recognize malware by tracking sequences and graphs of dependent system calls [7, 33, 36]. Multiple-call policies present a challenge to conventional sandboxes because by the time the sequence has been matched, the infected program has already performed system calls whose effects will remain in the system. By contrast, TxBox can roll back all effects of a malicious execution.

Policies based on sequences or graphs of system calls are not evasion-proof. If the malware writer is aware of the policy, he may able to modify the behavior of malware so that its system calls don't match any of the signatures (this may require changing the semantics of his malware). Nevertheless, these policies are useful insofar as they accurately describe the system-call behavior of existing malware.

DENY primitives. A DENY primitive consists of a single system call and a regular expression over its arguments. If a BLACKLIST or WHITELIST primitive is violated, the sandboxed process is terminated and its transaction is rolled back. By contrast, DENY primitives instruct the monitor to block specified calls while permitting the process to continue. DENY primitives can be used to emulate a conventional sandbox which simply denies certain calls,

Table I
SAMPLE TXBOX POLICIES.

Policy objectives for sandboxed process	TXBOX **policy**
Cannot access both 'AddressBook' (inode 100) and 'EmbarrassingSecrets' (inode 200)	BLACKLIST WREGEX (* I:100 *) AND (* I:200 *)
May not perform network I/O after accessing any file in directory 'secret' (inode 200)	BLACKLIST WREGEX * I:200 * S:1 *
May only write to the file '/home/user1/outout' (inode 150)	BLACKLIST WREGEX NOT(* I:150:w *) AND (* I:*:w *)
Must perform network I/O with IP address x.y.z.w	WHITELIST WREGEX (S:1:x.y.z.w)*
Don't allow any network I/O but continue execution	DENY connect* DENY sendto* DENY recvfrom*

possibly depending on the call's arguments.

DENY primitives also enable the administrator to run a sandboxed program without letting it perform certain operations. For example, suppose the administrator wants to run an untrusted codec in a sandbox but does not want it to talk to the network. This can be done by installing DENY primitives for all network I/O calls. If the codec tries to contact the network (*e.g.*, looking for updates), the call will fail but the codec may be able to handle this and continue local execution. Note the difference in enforcement semantics: if network I/O calls are installed as BLACKLIST primitives, then an attempt to make the call is a policy violation and the process will be killed.

TXBOX also benefits from semantic fidelity. Consider a conventional system-call monitor trying to block all network communication. When presented with a `write(fd, buf)` call, it must determine whether `fd` is mapped to a socket, which requires tracking the effect of all prior system calls. By contrast, TXBOX can easily determine whether `fd` is a socket by inspecting the transactional workset.

C. Policy enforcement

The TXBOX security monitor is responsible for enforcing security policies. It is implemented as a kernel module and consists of four parts: character device driver, system-call interposer, enforcer, and policy decision engine.

Character device driver. The character device driver provides an interface between the user-level policy manager and the kernel-level security monitor. The policy manager can be invoked automatically when a program residing in a particular directory is executed (the mechanism for this is described below). To sandbox a process, the policy manager sends the policy and the process's pid to the monitor through the interface of the character device driver. It also compiles the policy supplied by the system administrator into a string which can be loaded into the monitor's policy decision engine. The policy includes two lists of system calls: those that appear in BLACKLIST or WHITELIST primitives and those that appear in DENY primitives. Policies may also include critical system calls that will cause the system-call interposer to trap into the security monitor. The set of critical system calls always includes calls involving program termination and external I/O. To prevent malicious user-level processes from impersonating the policy manager, the monitor will communicate only with processes that are running with root privileges.

System-call interposer. The interposer patches the system call table with TXBOX call wrappers. In contrast to conventional wrappers, TXBOX wrappers are very simple (see Algorithm 1). First, if the call and its arguments match one of the DENY primitives in the policy, the call is blocked and an error is returned to the process. TXBOX wrappers can use kernel data structures to map arguments (*e.g.*, file descriptors) directly to kernel objects, eliminating the risk of race conditions. Second, if the call has not been denied, it is logged along with its arguments. Third, if the call is critical, control is passed to the policy decision engine. Otherwise, the call is permitted to execute. Note that the wrapper does not try to determine if the call is malicious or not, since all of its effects on the local system can be rolled back later if a violation is detected.

To handle symbolic links, TXBOX relies on Dazuko's helper function which tracks the link's target file. If the argument file of a system call is a symbolic link, the TXBOX wrapper calls this function to get the target's name and adds it to the trace instead of the name of the link.

Because rename calls can change the inode number assigned to a file, TXBOX's wrapper for rename keeps the mapping from the old number to the new number. Once the policy decision engine has decided to commit the transaction, any installed policy that uses an old inode number is updated with the corresponding new number.

Enforcer. The enforcer provides additional hooks into the kernel transaction mechanism, forcing the sandboxed process to run in transactional mode. When instructed by the policy decision engine, it either commits the transaction, or

Algorithm 1 Algorithm of a TxBox system-call wrapper

```
if should_deny(syscallname,pid) then
   return error
else if should_log(syscallname,pid) then
   Add the syscall and its arguments to process-specific
   syscall log
end if
if is_critical_call(syscallname) then
   Invoke policy decision engine
   if Policy violation detected then
      Roll back the transaction and exit
   end if
end if
Call the original syscall function
```

kills the process and aborts the transaction.

Policy decision engine. The policy decision engine is invoked when the sandboxed process attempts to perform a critical call. It has access to the process-specific system-call log provided by the system-call wrapper and the transactional workset provided by the OS. Together, they constitute the *trace* of the process. The policy decision engine uses a regular expression parser to match installed policies against the trace and detect policy violations using Algorithm 2. Intuitively, a trace violates the policy if it matches any of the BLACKLIST primitives or if it deviates from any of the WHITELIST primitives.

Algorithm 2 Algorithm for determining if process trace pt violates policy p

```
for each BLACKLIST/WHITELIST primitive prim in policy p in specified order do
   if (prim.regextype==WREGEX) then
      if (!match(prim.regex,pt.txworkset) and (prim.type==
      WHITELIST)) then
         return violation
      else if (match(prim.regex,pt.txworkset) and
      (prim.type==BLACKLIST)) then
         return violation
      end if
   else
      if (!match(prim.regex,pt.syscalltrace) and (prim.type==
      WHITELIST)) then
         return violation
      else if (match(prim.regex,pt.sycalltrace) and
      (prim.type==BLACKLIST)) then
         return violation
      end if
   end if
end for
return ok
```

If a violation is detected, the policy decision engine can either instruct the enforcer to kill the sandboxed process and abort the transaction, or pass control to the policy manager for memory checks, *etc.*, and wait for its decision.

If no violation is detected and the critical call that caused the invocation of the policy decision engine is a termination call such as exit or exit_group, the engine instructs the enforcer to commit the transaction. If the critical call is a user-specified call other than external I/O, it is allowed to execute inside the current transaction. Critical calls involving I/O are handled as described in Section V-D.

Implementation. Our implementation of the TxBox security monitor is based on Dazuko [12], an open-source Linux kernel module. It provides a character device interface and supports system-call interposition in the kernel, which in TxBox is based on hooking the system call table but can also be based on the Linux Security Module framework.

Implementing the TxBox security monitor required several substantial changes to Dazuko and TxOS. Dazuko was modified to (i) compile and run on TxOS—this included changing Dazuko to use TxOS kernel data structures (*e.g.*, inodes) which are different from the standard kernel data structures; (ii) implement TxBox system-call wrappers (described above); (iii) support communication between the user-mode policy manager and the kernel-mode security monitor through the character device interface for installing policies and registering handlers for ON_POLICY_VIOLATION events; and (iv) trap exec calls which execute programs from specified directories and switch control to the policy manager so that the program can be sandboxed. Changes to TxOS included (i) allowing system calls responsible for external I/O to execute non-transactionally without affecting the current transaction and (ii) enabling the TxBox security monitor to force another process to execute in a transaction.

The latter task presented an interesting challenge. In TxOS, an application starts a transaction by calling xbegin, which causes the common system-call handler in 'entry.S' to invoke the $beginTransaction$ kernel function, followed by do_sys_xbegin. Obviously, there is no xbegin in the sandboxed process. To force it into a transaction (possibly in the middle of a system-call execution), TxBox cannot call $beginTransaction$ directly because $beginTransaction$ checks if the current call is xbegin and, if so, stores the context of the user process so that it can be restored if the transaction is aborted. We added a $forced\_transaction$ flag (set by the TxBox enforcer) to the process-specific task structure and modified the handler. If the sandboxed process makes a system call when the flag is set, the handler saves the `eax` register which contains the number of the actual call, replaces it with 342, the number of xbegin, then calls $beginTransaction$ and do_sys_xbegin. Once the forced transaction starts, `eax` is restored to the

number of the actual call.

To implement the TxBox policy decision engine, we ported a regular expression library to execute in the kernel. We use hash tables to store and quickly look up policy-related information. The hash-table key for every policy object is the pid of the process on which that policy is being enforced. A policy object contains the list of all WHITELIST and BLACKLIST regular expressions which are part of that policy. It also contains three hash tables for looking up, respectively, if a system call should be logged, if it should be denied, and if it is critical.

D. Handling external I/O by sandboxed process

A typical application mostly performs two types of I/O: disk and network. System transactions buffer all disk I/O until the transaction commits or aborts. Certain operations—in particular, those requiring bidirectional communication to an external entity (this includes network I/O) and writing to external devices—cannot be buffered until the end of a transaction and thus have to be executed non-transactionally.

How to handle external I/O whose effects cannot be undone is a generally unsolvable problem faced by any sandbox. If the sandboxed code attempts to make a remote network call, the sandbox must allow or deny the call—even if the information about the code's execution so far is insufficient to determine whether the code is malicious or benign. Distributed transactions are not feasible in most sandboxing scenarios because the destination of the call made by an untrusted application may be malicious and not conform to transactional semantics (*e.g.*, it may refuse to roll back when instructed by the security monitor).

Conventional system-call monitors make the allow/deny decision on a call-by-call basis. The TxBox solution is superior. When the sandboxed process attempts to perform an external I/O call, TxBox first checks if the currently enforced policy has any DENY primitives that match this system call and its arguments. If such a DENY primitive exists, TxBox returns an error without performing the external I/O operation and continues executing the program as part of the current transaction, giving it an opportunity to handle the failed call. Denying an I/O call is always a safe decision because it guarantees full recoverability, regardless of what the sandboxed process does to the local system.

If there are no DENY primitives matching the call, control is switched to the policy decision engine. The engine inspects the trace of the process, which includes all of its prior system calls, their arguments, and all system objects affected by the process, and matches the trace against the policy as in normal enforcement (see Section V-C).

If the process has already violated the policy, TxBox terminates it and aborts the current transaction, rolling back all of its effects. If the process has not yet violated the policy, TxBox executes the I/O operation outside of the current transaction but continues running the process in the current transaction. If the process is later found to have violated the policy, TxBox cannot roll back the I/O calls, but local recoverability is always preserved.

In summary, TxBox gives two enforcement options. The first option is to deny external I/O (possibly depending on the arguments).[2] This preserves full recoverability if a violation is detected, but may cripple functionality of the untrusted program. The second option is to allow external I/O, but if the program violates the policy after I/O has been executed, recover *locally* by undoing all of its effects on the host. We argue that this is the best any sandbox can hope to achieve.

VI. Evaluation

In this section, we benchmark the performance of TxBox and evaluate its ability to sandbox substantial applications and roll back the effects of malicious execution. In performance tests, we compare TxBox with the standard Linux kernel (version 2.6.22.6) as well as the Linux kernel with the Dazuko module (version 2.3.4) installed. Because TxBox uses Dazuko's system-call hooking mechanism and character device interface, Linux kernel with Dazuko installed is an appropriate baseline for measuring the overhead of transactional execution and security checks on transactional worksets. All experiments were performed on a server with one quad-core Intel X5355 processor running at 2.66 GHz with 4GB of memory, unless otherwise mentioned. We omit the statistical variance, which is low in all cases.

For installing test policies automatically, we put all test programs in a dedicated directory and register the policy manager for the ON_EXEC event so that its gets control whenever a program from this directory is executed. The policy manager installs the policy and instructs the enforcer to put the sandboxed process into a forced transaction.

A. Performance

Micro-benchmarks. Table II shows the overhead of TxBox for individual system calls—including read, write, and fork/exec—compared to the base Linux kernel with and without Dazuko. The policy for all tests is BLACKLIST WREGEX *I:1234* unless otherwise specified.

In most cases, the cost of transactional execution and security checks—represented by the performance penalty of TxBox viz. standard Linux with Dazuko installed—is negligible. The single exception is open. Note that open is by far the worst possible system call for TxBox, because the TxOS kernel needs to create a shadow copy of the object and add it to the transactional workset. The overhead of a single open call is broken down in Table III; security enforcement is responsible for less than 5%. In practical applications, the cost of open will be amortized over many system calls.

[2]It may also be possible to make decisions specific to a network protocol such as DNS or HTTP. This requires the monitor to accurately mirror protocol state, which is hard in general and prone to the same semantic gaps that allow malicious processes to exploit incorrect mirroring of kernel state in system-call monitors.

Table II
SYSTEM-CALL MICRO-BENCHMARKS. TIMES SHOWN FOR THE FIRST FOUR ROWS ARE AVERAGES OF WALL-CLOCK TIMES OVER 100,000 RUNS FOR 10 DIFFERENT SETS. TIMES SHOWN FOR FORK AND FORK+EXEC ARE AVERAGES OF WALL-CLOCK TIMES OVER 10,000 RUNS FOR 10 DIFFERENT SETS.

Syscall	Kernel		
	Linux	Linux+Dazuko	TXBOX
getuid	0.08 μs	0.08 μs 1.00×	0.08 μs 1.00×
open	1.53 μs	1.62 μs 1.06×	4.72 μs 3.09×
read	0.27 μs	0.27 μs 1.00×	0.27 μs 1.00×
write	0.27 μs	0.27 μs 1.00×	0.32 μs 1.18×
fork	82.7 μs	82.8 μs 1.00×	83.4 μs 1.01×
fork+ exec	136.7 μs	136.7 μs 1.00×	138.9 μs 1.01×

Table III
BREAKDOWN OF PERFORMANCE OVERHEAD FOR OPEN.

Cause	Time
open	1.53 μs
System-call interposition (Dazuko)	0.09 μs
Transactional overhead	2.98 μs
Policy overhead	0.12 μs
Total	4.72 μs

Application benchmarks. We evaluated TXBOX on gzip, make, and PostMark. PostMark (version 1.51) is a file-system benchmark which simulates the behavior of an email, network news, and e-commerce client. Evaluation on larger applications can be found in Section VI-B.

Table IV shows the slowdowns for gzip and make. For gzip, which does not involve many file-system operations, the overhead of TXBOX is negligible (1.007×). For make, which involves more file-system operations than gzip, the overhead is 1.18×. On the other hand, PostMark benchmark involves a large number of file-system operations and represents the worst-case scenario for TXBOX because it requires a large number of shadow objects to be created. Furthermore, because the transaction can only be committed once the PostMark benchmark calls exit, TxOS kernel needs to keep track of all shadow objects until the end of the program.

Table IV
TIME TAKEN BY GZIP TO COMPRESS A 4 MB FILE AND BY MAKE TO COMPILE TWO SOURCE FILES WITH POLICY BLACKLIST WREGEX *I:1234* ON TXBOX AND LINUX. TIMES SHOWN ARE AVERAGES OF WALL-CLOCK TIMES OVER 100 RUNS.

	gzip	make
Linux	0.0401 sec	0.145 sec
Linux+ Dazuko	0.0403 sec 1.004×	0.145 sec 1.00×
TXBOX	0.0404 sec 1.007×	0.177 sec 1.18×

Table V
POSTMARK BENCHMARK RESULTS IN FILE-SYSTEM TRANSACTIONS PER SECOND WITH POLICY BLACKLIST WREGEX *I:1234*. THE NUMBER OF FS-TRANSACTIONS IS SET TO 100,000. POSTMARK IS CONFIGURED TO USE NON-BUFFERED I/O FOR ALL THE TESTS.

Linux	8411 FS-transactions/sec
Linux+Dazuko	7692 FS-transactions/sec 1.09×
TXBOX	16666 FS-transactions/sec 0.50×

Performance results for the PostMark benchmark are shown in Table V. They are presented in terms of file-system transactions per second (we refer to them as FS-transactions to avoid confusion with system transactions). TXBOX results in a factor-of-2 speed-up (represented in the table as 0.5× slowdown) due to the fact that the transaction commit groups all writes and presents them to the I/O scheduler all at once, thus improving disk arm scheduling.

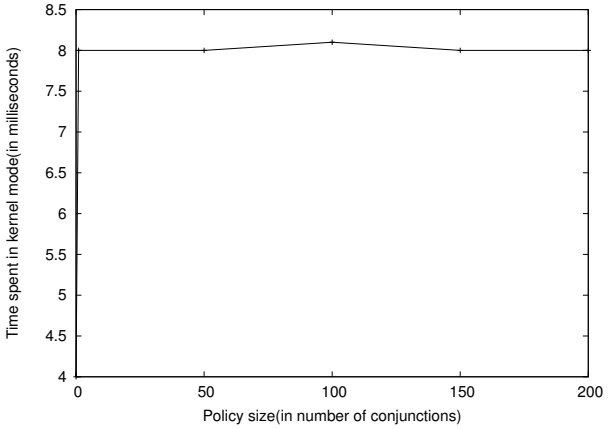

Figure 3. Time spent in kernel mode (as reported by "time") for a simple program which opens 100 existing files, as a function of the policy size. Times shown are averages over 10 runs.

Scalability. To evaluate the scalability of TXBOX, we built a simple application which opens 100 existing files and measured how its runtime varies with the increase in the size of the policy (the number of inodes included in the policy). In this test, we enforce a policy which is a conjunction of N statements, each of which is a blacklist on a single inode. We run this test on a laptop with an Intel Core Duo 2.00 GHz CPU and 2 GB RAM. The results are shown in Figure 3.

I/O-intensive applications. Because transactional semantics cannot be preserved over external I/O, the security monitor must be invoked before allowing any system call which performs external (*e.g.*, network) I/O. The call is then executed outside the current transaction. For example, consider a process making the following sequence of calls:

```
fd = open("foo",..)
read(fd, ..)
sockfd = socket(..)
sendto(sockfd,..)
close(sockfd)
```

```
close(fd)
..
```

Because sendto performs network I/O, TxBox checks if it matches any DENY primitive in the current policy. If yes, the call is denied. If not, the monitor is invoked twice to check if the process violates any BLACKLIST or WHITELIST:

```
TX BEGIN
fd = open("foo",..)
read(fd, ..)
sockfd = socket(..)
CHECK TX POLICY VIOLATION
NONTRANSACTIONAL sendto(sockfd,..)
close(sockfd)
close(fd)
..
CHECK TX POLICY VIOLATION
TX END
```

To measure the performance implications of multiple invocations of the policy decision engine on network-I/O-intensive workloads, we sandbox *wget* in TxBox and use it to download different large files from the Internet. TxBox is configured to execute the following system calls non-transactionally: sendto, recvfrom, connect, send, recv, ioctl, read (from a socket) and write (to a socket). Even though ioctl does not perform any external I/O, it cannot be executed inside a transaction with the version of TxOS we are using in our prototype implementation of TxBox. This is not a fundamental limitation; in the latest version of TxOS, ioctl codes are whitelisted on a case-by-case basis.

After opening a socket, *wget* reads a chunk of data from the socket and writes it to the target file. This is the worst-case setting for TxBox. The results are in Figure 4. In all tests, *wget* runs on a laptop with an Intel Core Duo 2.00 GHz CPU and 2 GB RAM. System time (*i.e.*, time in the kernel) for *wget* increases by 30-40% when running inside TxBox and the overall download time increases by 1-40%.

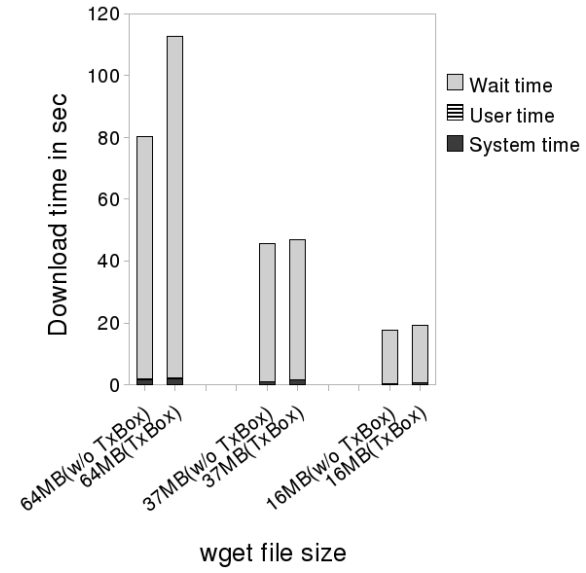

Figure 4. Overhead of TxBox for different sizes of files retrieved by *wget* (time spent in user-land is minuscule and not visible in the graph).

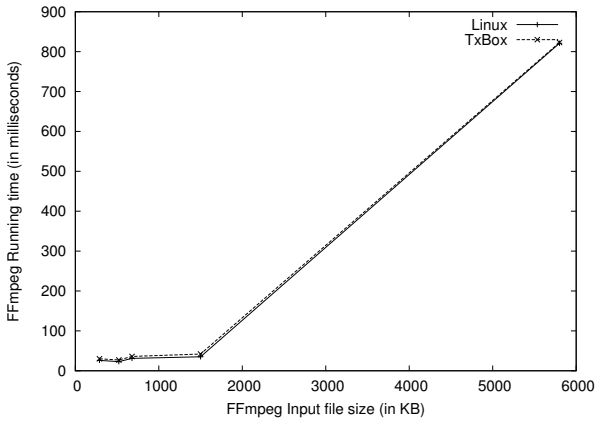

Figure 5. FFmpeg's running time in TxBox with policy BLACKLIST WREGEX *I:1234* and Linux (lines overlap).

Overhead for trusted applications. Because TxBox isolates transactional applications from non-transactional applications, the latter incur a performance overhead to check for conflicts with transactions. When system transactions are used for sandboxing, the overhead for trusted, non-sandboxed applications is modest. The average non-transactional overhead is 44% at the scale of a single system call on TxBox, using the same microbenchmark described in [48]. In the context of a larger application, however, these overheads are amortized across other work. For instance, compiling a non-transactional Linux kernel in TxBox incurs an overhead of less than 2% compared to unmodified Linux.

B. Functionality

Creating files in a protected directory. We downloaded the source code of the *vim* editor and configured it to install in `/usr/local` and compile using `make`. Next, we ran "make install" in a sandbox with the BLACKLIST WREGEX *I:164564* policy where 164564 is the inode number of the directory `/usr/local/bin`. "make install" actually copies files to multiple directories, such as `/usr/local/share`, `/usr/local/bin`, *etc.* Our policy only designated `/usr/local/bin` as the protected directory, so we were testing the ability of TxBox to correctly roll back all of the sandboxed process's effects on the system, not just those on the forbidden directory. Execution of "make install" resulted in a violation and TxBox correctly rolled back its effects on all directories, restoring the entire `usr/local` to its original state.

Malicious MIME handler in a browser. We created a 'tarhandler' which reads the `sample` file from the pro-

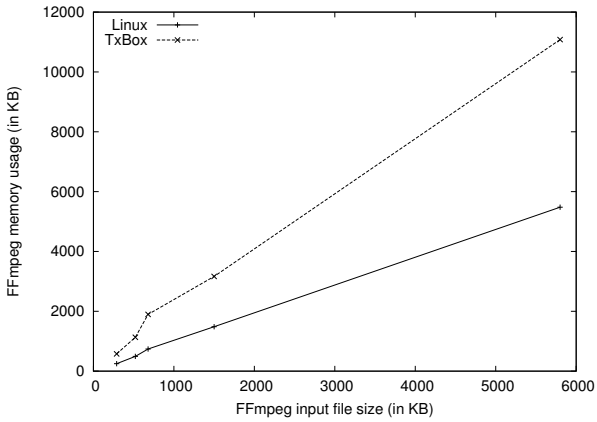

Figure 6. FFmpeg's memory usage in TxBox with policy BLACKLIST WREGEX *I:1234* and Linux. The increased memory usage is due to maintaining the transactional workset.

tected directory `/home/secret` and writes out its contents into `/tmp/foo`. We registered 'tarhandler' with the *lynx* browser for MIME type 'application/x-tar' and installed the BLACKLIST WREGEX *I:183145* policy, where 183145 is the inode number of `/home/secret`. The browser correctly executed in the sandbox with full functionality, but after 'tarhandler' read from the forbidden directory, all changes to `/tmp/foo` were rolled back.

Multimedia converter. To simulate the effect of a malicious multimedia converter trying to write to unrelated files in a user's home directory, we configured *ffmpeg*, a popular open-source codec, to create output files in the `/home/user1/` directory. We created a sandbox with the BLACKLIST WREGEX *I:181064* policy, where 181064 is the inode number of `/home/user1`. When *ffmpeg* tried to write to a file named 'output.avi' in `/home/user1`, TxBox detected a policy violation and reverted all changes made by *ffmpeg* to `output.avi`. Performance overhead of sandboxing FFmpeg is shown in Figs. 5 and 6, where memory overhead is computed as the difference between the total cached memory before each execution and the total cached memory before committing the transaction.

JavaScript engine. To evaluate TxBox on a complex application, we use the Google V8 benchmark (version 2) [28] on the SpiderMonkey JavaScript engine (version 1.8.0) running inside TxBox. JavaScript engines are designed to ensure that an untrusted script has access only to limited system resources (files, system calls) needed for its correct operation. They are, however, fairly complex programs and can suffer from vulnerabilities (*e.g.*, buffer overflows) which may be exploited for arbitrary code execution [39, 61]. Executing the engine inside a transaction can help ensure that all system accesses by untrusted JavaScript are confined even if the engine is buggy. Furthermore, additional policies can be enforced. For example, to prevent any script from contacting known malicious domains or to enforce the same-origin policy from outside the potentially buggy JavaScript engine, the administrator can install a blacklist or whitelist policy on the destinations of network connections.

Table VI
GOOGLE V8 JAVASCRIPT BENCHMARK SCORES FOR THE SPIDERMONKEY ENGINE. THE POLICY IS WHITELIST WREGEX (S:1:X.Y.Z.W)*. HIGHER SCORES MEAN BETTER PERFORMANCE, *i.e.*, LESS EXECUTION TIME. AVERAGES ARE CALCULATED USING GEOMETRIC MEAN AS SUGGESTED IN THE BENCHMARK.

Test	w/o TxBox	with TxBox
Richards	25.6	25.2
Deltablue	30.2	29.9
Raytrace	53.2	51.9
EarlyBoyer	83.4	83.1
Avg. score (GM)	44	43.2

The results of executing SpiderMonkey inside TxBox are in Table VI. These tests were performed on a laptop with an Intel Core Duo 2.00 GHz CPU and 2 GB RAM. In all tests, the overhead of TxBox is negligible (less than 5%).

On-access anti-virus scanning and parallelization of security checks. Anti-virus scanners are among the most common tools used to prevent spreading of malicious programs. They primarily use signature matching to detect viruses. The scanner maintains a database of signatures for known viruses and searches for matching patterns in programs and files.

An anti-virus scanner can be activated manually by the user who requests to scan a specific file or directory. The alternative is transparent, *on-access* activation when a program is executed or a document is opened. This often imposes a significant performance penalty because the program cannot start executing until the scan is finished.

TxBox makes computationally intensive on-access anti-virus scanning practical by speculatively executing untrusted applications and loading suspicious documents inside a sandbox, while performing a concurrent scan. Many virus detection methods, including string matching, filtering, and algorithmic scanning, can be executed in a parallel thread (more sophisticated methods may require access to transactional state—see Section VII). If the suspicious application's interactions with the system cause no conflicts with other processes and the scanner thread does not find any problems, then the transaction is committed and the application can continue from the point where scanning finished.

We carried out several experiments to demonstrate how parallelizing security checks improves performance (our approach is substantially different from Nightingale et al. [40]—see Section III). For gzip and PostMark, control is passed to the policy manager whenever a file is opened. The manager then runs the ClamAV anti-virus scanner on this file. In standard Linux with Dazuko, the manager blocks the program until the scan is finished. In TxBox, the manager

Table VII
PERFORMANCE OF GZIP WITH CLAMAV ANTI-VIRUS ON-ACCESS SCANNING OF FILES WHENEVER THEY ARE OPENED. TIMES SHOWN ARE AVERAGES OF WALL-CLOCK TIMES OVER 100 RUNS.

	Input file size	
	4MB	400MB
Dazuko	0.14s	3.720s
TXBOX	0.203s 1.45×	3.718s 0.99×

lets the program continue inside the sandbox while the anti-virus scan is being performed.

Performance for gzip is shown in Table VII. When the amount of data handled by gzip is small (4 MB), the overhead of the transactional mechanism dominates. As the amount of data increases, the transactional overhead is compensated by performance gains from parallelization.

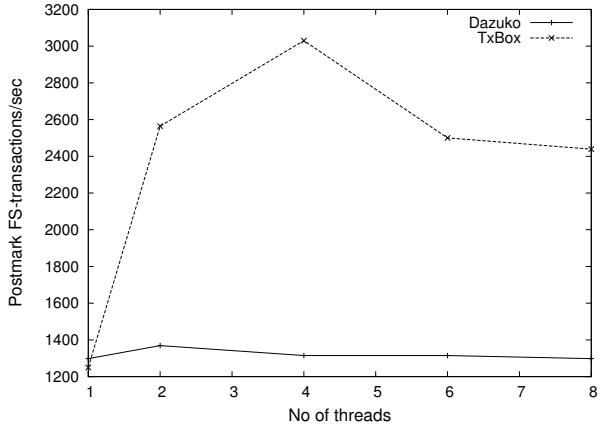

Figure 7. Performance of PostMark with ClamAV anti-virus on-access scanning of files whenever they are opened. We use non-buffered I/O and set the number of file-system transactions to 100,000.

For PostMark tests, we use a multi-threaded policy manager (Fig. 7). As we increase the number of threads, performance of TXBOX increases at a fast rate while performance of Linux+Dazuko remains roughly the same. With 4 threads TXBOX is 2.3 times better than Linux+Dazuko.

VII. LIMITATIONS

Kernel-based security monitor is vulnerable to kernel attacks. Like any OS-based security enforcement mechanism, including existing system-call monitors, TXBOX is intended to provide security against malicious user-level code. If the OS kernel is compromised, malicious code can potentially disable TXBOX or interfere with its operation. Defenses against kernel-based malware are outside the scope of this paper, but, in general, maintaining integrity of security enforcement in the face of kernel attacks requires a source of trust outside the OS, such as provided by trusted hardware (*e.g.*, [43, 50]) or a virtual machine (*e.g.*, [44, 57]).

Transactional semantics may change the behavior of sandboxed processes. One possible side effect of running every untrusted process inside a TXBOX sandbox is that an access to a shared resource by a benign sandboxed process may fail due to a transactional conflict with another sandboxed process (see Section IV). This usually indicates a race condition, which may very well be a problem by itself. Furthermore, we expect that in a normal execution, only untrusted applications are sandboxed and thus the number of sandboxed processes on the host machine is fairly small.

TxOS has an auto-retry mechanism which, if set, attempts to re-execute the failed transaction transparently to the process. The number of retries is a configurable parameter. If the transaction is aborted due to a security violation, it is not re-started automatically (see Section V-A).

A secondary concern with automatically wrapping applications in system transactions arises when the application itself uses system transactions for internal synchronization. Currently, TXBOX only provides flat nesting; a nested transaction uses the same working set as its parent. TXBOX could isolate nested sibling transactions from each other with fairly straightforward extensions to the system transaction mechanism, which we leave for future work.

Transactional state is not shared. Our parallelization experiments use ClamAV [9], a relatively simple scanner which looks for bit patterns associated with known viruses. More sophisticated anti-virus tools may need to observe the execution of the program in order to decide whether it is malicious or not. To run such tools in parallel with the sandboxed process, TXBOX must share the transactional state of the process with the tool. This is not supported in our current prototype but presents no conceptual difficulties.

Colluding malware may evade security policies. Any non-trivial policy that involves more than one system call may be violated by two or more colluding malicious programs which execute on the same host independently. Consider a very simple policy: "a program is malicious if it makes system call A followed by system call B." The first malicious program makes call A and saves its internal state in a local file. The second program reads in the state of the first program and makes call B, achieving the same effect as a single violating program. Obviously, more complex policies can be bypassed by a similar attack. No sandboxing mechanism can reliably prevent this.

Processes that generate very large worksets are killed. The sandboxed process may try to bloat its transactional workset by performing irrelevant operations. If TXBOX runs out of memory to store the workset of any process, the process is killed and the transaction is rolled back. TXBOX does not currently support swapping out a process's workset as this may open up opportunities for denial of service.

This approach also prevents sandboxing very long-lived applications because TXBOX cannot tell the difference be-

tween a program that has been running for a long time and legitimately accumulated a large workset and a malicious program which is deliberately bloating its workset.

An alternative approach is to perform intermediate commits. When the workset of the sandboxed process gets too big, TxBox checks if the process has already violated the sandboxing policy. If so, the process is killed and the transaction is rolled back. If the policy is not (yet) violated, the transaction is committed and a new one started, but TxBox keeps trace information from the old transaction and merges it into the trace of the new transaction.

This approach preserves the ability of TxBox to detect violations that span the commit point, but sacrifices full recoverability when a violation is detected because the process can only be rolled back to the last commit point. This is a strict generalization of standard system-call monitoring, since the latter commits on every system call.

VIII. Conclusions

Increasing popularity of multi-core architectures is driving the development of new mechanisms for managing concurrency in software applications. One such mechanism is system transactions, which allow a sequence of updates to the system state made by one process to be performed atomically, in isolation from other processes. We demonstrate that system transactions provide a powerful primitive for implementing secure, efficient sandboxes for untrusted code.

TxBox, our prototype sandboxing system based on a modified Linux, enables speculative execution of untrusted programs and automatic recovery from their harmful effects. By inspecting system calls made by the program and its accesses to system resources, the TxBox security monitor can determine whether the programs satisfies the desired security policy. Supported policies include system-call automata, as well as data-flow and access-control policies spanning multiple system calls. If a security violation is detected, TxBox aborts the transaction wrapping the malicious program, and the system is restored as if the process never executed.

Unlike many system-call interposition tools and monitors based on speculative execution, TxBox cannot be circumvented by TOCTTOU (time-of-check-to-time-of-use) and other concurrency attacks, nor by attacks that exploit incorrect mirroring of the kernel or file-system state, nor by split-personality malware whose behavior changes depending on whether it is instrumented with security checks or not.

TxBox combines kernel-based enforcement with user-level policies. This yields low performance overheads, enabling the use of TxBox in production systems, especially ones that already need system transactions to manage concurrent access to system resources. TxBox also improves the performance of on-access anti-virus scanning on multi-core processors by executing the untrusted application on one core and performing the scan in parallel on another core.

As system transactions increase in popularity and support becomes available in commodity operating systems, security enforcement mechanisms should take advantage of them. We view TxBox as a step in this direction.

Acknowledgements. We are grateful to our shepherd David Wagner for many helpful comments and to Emmett Witchel for his insightful advice and for guiding the development of TxOS. The research described in this paper was partially supported by the NSF grants CNS-0746888 and CNS-0905602, Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *USENIX Security*, 2000.

[3] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *USENIX Winter*, 1995.

[4] M. Bernaschi, E. Gabrielli, and L. Mancini. REMUS: A security-enhanced operating system. *TISSEC*, 5(1), 2002.

[5] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *S&P*, 2006.

[6] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *CCS*, 2008.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE*, 2007.

[8] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCA*, 2008.

[9] Clam AntiVirus. http://www.clamav.net/lang/en/.

[10] D. Clark and D. Wilson. A comparison of commercial and military computer security policies. In *S&P*, 1987.

[11] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *LISA*, 2000.

[12] Dazuko. Dateizugriffskontrolle (file access control). http://www.dazuko.org.

[13] J. Douceur, J. Elson, J. Howell, and J. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[14] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, 2005.

[16] U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *NSPW*, 1999.

[17] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *S&P*, 2003.

[18] B. Ford and R. Cox. Vx32: Lightweight userlevel sandboxing on the x86. In *USENIX ATC*, 2008.

[19] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *S&P*, 1999.

[20] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *CCS*, 2004.

[21] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[22] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS*, 2007.

[23] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[24] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[25] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[26] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

[27] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *USENIX Security*, 1996.

[28] Google. V8 benchmark suite. http://v8.googlecode.com/svn/data/benchmarks/v2/.

[29] T. Harris and S. Peyton-Jones. Transactional memory with data invariants. In *TRANSACT*, 2006.

[30] M. Hill, D. Hower, K. Moore, M. Swift, H. Volos, and D. Wood. A case for deconstructing hardware transactional memory. Technical Report CS-TR-2007-1594, Dept. of Computer Sciences, University of Wisconsin-Madison, 2007.

[31] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3), 1998.

[32] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*, 2000.

[33] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security*, 2009.

[34] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[35] B. Lampson. A note on the confinement problem. *CACM*, 16(10), 1973.

[36] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. AccessMiner: Using system-centric models for malware protection. In *CCS*, 2010.

[37] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[38] M. Locasto, A. Stavrou, G. Cretu, and A. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX ATC*, 2007.

[39] Mozilla Firefox JavaScript engine vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3982.

[40] E. Nightingale, D. Peek, P. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[41] Novell. AppArmor application security for Linux. http://www.novell.com/linux/security/apparmor/.

[42] NSA. Security-enhanced Linux. http://www.nsa.gov/research/selinux/.

[43] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security*, 2004.

[44] N. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, 2007.

[45] Hard link vulnerability. http://plash.beasts.org/wiki/PlashIssues/HardLinkVulnerability?highlight=%28PlashIssues/%29|%28CategoryPostponed%29.

[46] open() with O_CLOEXEC returns an error. http://plash.beasts.org/wiki/PlashIssues/CloexecOpenFails?highlight=%28PlashIssues/%29|%28CategoryPostponed%29.

[47] D. Porter. *Operating system transactions*. PhD thesis, The University of Texas at Austin, 2010.

[48] D. Porter, O. Hofmann, C. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[49] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.

[51] M. Seaborn. Plash. http://plash.beasts.org/wiki/.

[52] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *S&P*, 2001.

[53] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *OSDI*, 1996.

[54] S. Sidiroglou and A. Keromytis. Execution transactions for defending against software failures: use and evaluation. *Int. J. Inf. Secur.*, 5(2), 2006.

[55] W. Sun, Z. Liang, R. Sekar, and V. Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *NDSS*, 2005.

[56] D. Wagner and D. Dean. Intrusion detection via static analysis. In *S&P*, 2001.

[57] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *CCS*, 2009.

[58] R. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT*, 2007.

[59] R. Watson, J. Anderson, K. Kennaway, and B. Laurie. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, 2010.

[60] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *RAID*, 2000.

[61] Microsoft Windows JavaScript engine arbitrary code execution vulnerability. http://tools.cisco.com/security/center/viewAlert.x?alertId=18969.

[62] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*, 2009.

[63] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.