

Fine-Grained User-Space Security Through Virtualization

Mathias Payer

mathias.payer@inf.ethz.ch
ETH Zurich, Switzerland

Thomas R. Gross

trg@inf.ethz.ch
ETH Zurich, Switzerland

Abstract

This paper presents an approach to the safe execution of applications based on *software-based fault isolation* and *policy-based system call authorization*. A running application is encapsulated in an additional layer of protection using dynamic binary translation in user-space. This virtualization layer dynamically recompiles the machine code and adds multiple dynamic security guards that verify the running code to protect and contain the application.

The binary translation system redirects all system calls to a policy-based system call authorization framework. This interposition framework validates every system call based on the given arguments and the location of the system call. Depending on the user-loadable policy and an extensible handler mechanism the framework decides whether a system call is allowed, rejected, or redirect to a specific user-space handler in the virtualization layer.

This paper offers an in-depth analysis of the different security guarantees and a performance analysis of libdetox, a prototype of the full protection platform. The combination of *software-based fault isolation* and *policy-based system call authorization* imposes only low overhead and is therefore an attractive option to encapsulate and sandbox applications to improve host security.

Categories and Subject Descriptors D.4.6 [Operating Systems]: Security and Protection; D.3.4 [Programming Languages]: Processors — Run-time environments

General Terms Security, Performance

Keywords Security, Virtualization, Dynamic binary translation, Dynamic instrumentation, User-space software virtualization, Process sandboxing, Policy-based system call authorization, Optimization.

1. Introduction

The secure execution of unknown binary code is an important problem. As the complexity and diversity of the installed software base increases, more techniques are needed to guarantee the security of a

system. Software patches in response to identified exploits or malware discovery tools are useful, yet both approaches are reactive (and solutions or workarounds for vulnerabilities may take time to develop) and are therefore of limited utility. A better solution is a step towards proactive security and fault detection by strictly limiting the potential damage that can be done.

To contain security problems in a practical system, it is important to encapsulate applications and to limit the data they can access. An application running in a user-space sandbox can access its data but cannot break out of the virtualization layer and access any other system data or escalate privileges.

User-space sandboxing builds an additional fine-grained layer of protection around an application. Binary translation enables fine-grained control of the executed instructions and enables additional security guards that can be added dynamically into the compiled code. These guards control all executed instructions inside the sandbox, and check all system calls that interact with the kernel inside the process itself. Instructions that change the control flow of the application are wrapped so that they comply with a tight security model and instructions that branch into the kernel are redirected to a policy-based interposition system. The system calls are checked depending on the name, supplied parameters, and call location. A per-application policy describes the set of system calls that a program can execute and specifies which parameter combinations are allowed for each system call.

The proposed sandbox is completely invisible to the running application. Applications see no functional difference to an untranslated run, so programs cannot detect or circumvent the sandbox.

This paper presents an approach to a fast, secure user-space sandbox that enforces security. The security concept is built on the following two principles:

1. *software-based fault isolation*: the binary translator uses special guards to ensure that only application and library code is translated, that code cannot escape the binary translator, that no injected code on the heap and on the stack is executed, and that all system calls are redirected to the interposition framework.
2. *policy-based system call interposition*: the system call interposition framework ensures that all system calls are checked and validated and that only authorized system calls are executed. A policy controls which arguments and which program locations are allowed for each individual system call.

The sandbox loads a policy file before an application is executed and enforces this policy at runtime. If the user-program executes an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'11, March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03...\$10.00

illegal system call, illegal code, or tries to execute an unchecked control transfer then the process is terminated by the sandbox.

The policies can contain both white-listing and black-listing of system calls based on system call numbers, locations and arguments. Wildcards can be used to specify groups of arguments. An additional extension are the *redirected system calls*. If a system call is blocked then a fake value can be returned to the user-space program. The user-space program is unable to detect if a real system call or a redirected system call was executed. This feature can be used to analyze untrusted software or to re-implement system calls in user-space. See Figure 1 for an overview of the fault isolation layer and the implementation of fake system calls in user-space.

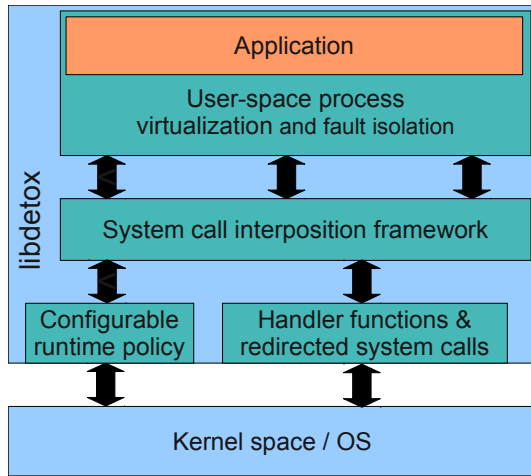


Figure 1. Sandbox overview with user-space fault isolation and delegation to redirected system calls.

The contributions of this paper are the combination of:

1. fine-grained *software-based fault isolation* using special guards that are implemented through binary translation to protect against code injections and the execution of unchecked code,
2. the additional guards that guarantee the fault isolation properties, and
3. flexible per-process user-defined *policy-based system call interposition* in user-space without the need for context switches to validate specific calls and without additional privileged code in the kernel. This paper also presents sample policies to guard the SPEC CPU2006 benchmarks and the nmap network security tool. A case study uses the Apache web-server to secure and benchmark a daemon process.

The presented approach is not limited to Linux or x86. libdetox, our implementation prototype for x86, supports the complete IA-32 ISA. libdetox is able to sandbox unmodified Linux binaries, dynamically add additional security guards, and redirect all system calls to the policy-based system call interposition framework.

Section 2 presents background information on a basic dynamic binary translator. Section 3 discusses security implications and covers process virtualization details as well as the specific guards and explains policy-based system call interposition. Important implementation details are highlighted in Section 4. The system is evalu-

ated in Section 5, followed by a discussion of related work and our conclusion.

2. Dynamic binary translation

This section describes the design of a basic dynamic binary translator that implements software-based fault isolation. The dynamic binary translator processes basic blocks of the original program and places the translated blocks in a code cache. The mapping between untranslated and translated blocks is recorded in a mapping table. Figure 2 illustrates the runtime layout of a basic table-based binary translator.

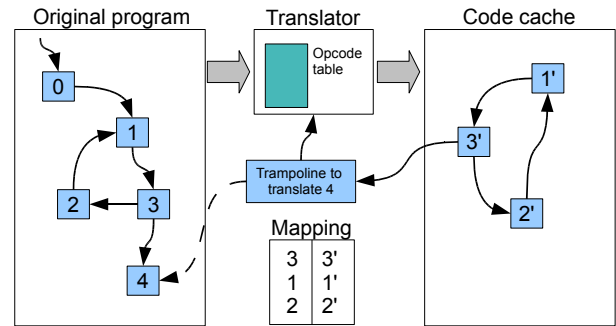


Figure 2. Runtime layout of a binary translator.

At the end of a basic block the translator checks if the outgoing edges are already translated and redirects control transfers to the translated counterparts in the code cache. If a target is not translated then the translator (i) constructs a trampoline that triggers the translation of the corresponding basic block and (ii) redirects control flow to that trampoline.

The translator processes one instruction at a time. Control instructions are translated using special handler functions, all other valid instructions are copied verbatim to the code cache. If the translator encounters an illegal instruction it aborts the program.

Control instructions are translated so that the execution of the translated program never leaves the code cache. For conditional control transfers, jump instructions, and call instructions the translator either constructs a trampoline if the target is not yet translated, or encodes a control transfer to the translated target in the code cache.

Indirect control transfers like indirect jumps, indirect calls, and function returns depend on runtime data and can change upon every execution. The targets of indirect control transfers point to untranslated code. The translator must replace these control transfers with a runtime lookup in the mapping table and a control transfer to the translated counterpart in the code cache. The runtime lookup for each indirect control transfer ensures that control flow is only transferred to well-known and translated locations.

These runtime lookups are responsible for most of the runtime overhead of dynamic binary translators, and most optimizations present in fast binary translators try to reduce this overhead.

3. Security guidelines

Software-based fault isolation is secure if programs are unable to escape the fault isolation sandbox and no code is executed that does not conform to a strict security policy. All executed code and all executed system calls must be checked and verified. If the executed code does not follow the security policy (e.g., due to programming bugs, potential exploits, or backdoors) then the sandbox detects these offenses and terminates the program.

The sandbox uses a layered security concept that builds on two principles, *software-based fault isolation*, and *policy-based system call interposition*. Each principle protects from specific attacks. Process sandboxing through dynamic binary translation is the first line of defense and ensures that (i) no injected code is translated or executed, (ii) application or library code cannot be overwritten, (iii) program code cannot escape or terminate the sandbox, and (iv) program code cannot overwrite any data structures owned by the sandbox. The binary translation framework translates all code before it is executed and uses special guards to ensure that the code conforms to the security specification and that, e.g., control transfers always target valid code. The first principle protects against all code based exploits (e.g., overflows and return to libc attacks).

Process sandboxing builds the foundation for the second security principle. The binary translation framework redirects all forms of system calls to the policy-based system call interposition framework. This framework checks every system call, the arguments of the system call, and the location of the system call. Only system calls that conform to the policy for the current application are allowed. Depending on the policy decision each system call can be:

1. rejected and the program is terminated,
2. rejected but a fake return value is returned,
3. redirected to an internal implementation (for special handling or additional checks),
4. allowed and the program continues.

Process sandboxing ensures that the translated code cannot escape the binary translator and the policy-based interposition framework limits the system calls that can be executed. The second principle protects against data based exploits (e.g., integer overflows and type errors) and builds a second line of defense at the coarse-grained system call level. This policy must be tight to avoid privilege escalation.

For performance reasons, target addresses of individual translated instructions are not checked. User-space software-based fault isolation and the system call interposition framework ensure that a program is unable to execute uncontrolled code and unspecified system calls and lays the power to control system calls and their allowed arguments into the hands of the policy writer.

3.1 Software-based fault isolation

Binary translation (BT) is a key component for user-space software-based fault isolation. A dynamic translation system translates and checks every machine code instruction *before* it is executed. Every direct control transfer is translated, and every indirect control transfer is intercepted and only translated branch targets are reached. The translator can change, adapt, or remove any invalid

instruction and is able to intercept system calls *before* they are executed. During the translation process code can be instrumented and augmented with additional security features. Security features like non-executable stack, stack guards, control flow evaluation, or argument checking for specific functions are added without the need to recompile the application. Even patches can be applied at runtime to fix bugs in running applications.

Fault isolation offers a very fine-grained control of security as all executed code must comply to a defined security policy, not just the executed system calls. Security frameworks that validate only bare system calls miss exploits that target the data integrity of the executed program because they only detect an intrusion if an invalid system call is executed but not when malicious code is executed in user-space; e.g., data can be written to open files or a memory mapped file can be changed without the need to execute a system call. Fault isolation detects code injections and terminates the program before the data structures are corrupted. Data-based exploits on the other hand are not detectable by fault isolation and are caught using policy-based system call interposition.

The binary translator is the foundation of the security guarantees of the presented sandbox. The binary translator should be modular and small to keep the trusted computing base small. This section presents design criteria for a modular and flexible binary translation fault isolation layer that is needed to implement the additional security guards and *system call authorization*. An important feature of the binary translator is that return addresses on the stack remain unchanged. This adds additional complexity when handling return instructions as they are translated to a lookup and an indirect control transfer. On the other hand an unchanged stack ensures that the original program can use the return instruction pointer on the stack for (i) exception management, (ii) debugging, and (iii) return trampolines. Additionally the user program does not know that it runs in a virtualized environment, and the address of the code cache is only known by the binary translator.

3.1.1 Translated code

Only translated application and library code is executed. This principle is enforced by the binary translator. By rewriting indirect control transfers into a runtime lookup and dispatch and adding trampolines to translate untranslated code on the fly, the translator ensures that execution of machine code is unable to escape the isolation layer. All outgoing edges of translated basic blocks either point to (i) trampolines that translate new code, (ii) translated code in the BT's code cache, or (iii) translated indirect control transfers that map untranslated targets to translated code and transfer execution flow accordingly. Only code in valid locations (e.g., imported library functions and application code) is translated. This principle ensures that no code injections on the heap or on the stack are possible.

3.1.2 Binary translation: static versus dynamic

The most important property of a binary translator is to ensure that all instructions are checked and translated prior to execution. Static binary translation is not able to cover all code. Hidden code in data sections could be reached through indirect jumps or a jump could target into an instruction. Such control transfers are hard to analyze statically, especially if malicious code targets a specific binary

translator, but are handled naturally in dynamic binary translators that translate code on a basic block level before the basic block is executed the first time.

Dynamic binary translators are therefore well suited to implement user-space software-based fault isolation.

3.1.3 Additional security guards

Binary translation guarantees that only translated instructions will be executed but does not prevent individual instructions from overwriting memory regions or executing specific system calls. Dynamic binary translation enables the implementation of additional security guards by rewriting and encapsulating specific instructions.

An important feature of the binary translator is that no pointers to internal data structures are left on the stack. The binary translator uses the same stack as the translated user-program to dynamically translate new basic blocks and dispatch indirect control transfers. A custom tailored exploit could target the binary translator itself. If the program were able to locate the internal data structures of the binary translator (e.g., the code cache), it could modify the executed code by directly changing instructions in the code cache and so break out of the isolation layer. Therefore the stack is pruned of pointers to internal data structures before the execution returns to the translated user-program. Additionally the translator guarantees that application code that tries to access internal data structures is not translated by virtualizing, e.g., addresses or registers.

The basic binary translator is extended by the following security guards that harden the user-space isolation sandbox and to ensure that application code cannot escape out of the sandbox:

Executable space protection: implements a form of executable space protection for x86 on a section basis in user-space. This protection holds for regions defined in ELF headers of the programs and loaded libraries, even if they are smaller than a page. The guard checks if the target area is defined in the program or an imported library and if it actually contains code. If there is a violation then the program is terminated. This guard protects against the execution of code injected through stack-based and heap-based buffer overflows.

Executable bit removal: this guard marks the code of the untranslated program as *non-executable* (by using `mprotect` calls). The application does not know the location of the internal code cache and is therefore not able to overwrite parts of the code cache with injected code. This guard ensures that only code from the binary translator and translated code in the code cache can be executed.

Return address verification: This guard checks that the return address is not changed by implementing a shadow stack that is only accessible from internal code where the return address is verified for each return instruction. The shadow stack contains pairs of addresses, the original location and the translated counterpart. If the address on the original stack does not correspond to the address on the shadow stack then the program is terminated. This guard protects against stack based overflows and return to libc attacks and is orthogonal to solutions like StackGuard [14], Propolice [24], libverify [3], and FormatGuard [12].

Signal handling: the binary translator keeps a mask of installed signals on a per-thread basis. Whenever a new handler for a specific signal is installed the code of the handler is wrapped into a trampoline that guarantees the secure execution of the signal and ensures that the signal processing code cannot escape the binary translator. This guard protects against errors in the trap handling and enables the sandbox to catch memory accesses to unmapped memory regions (e.g., probing for the location of the code cache).

Secure context transfers: the control transfer from the binary translator to translated code uses an indirect control flow transfer to ensure that no pointers to any internal data structures of the binary translator are exposed on the stack. This guard hides the internal data structures of the sandbox from the application.

Randomized addresses: the binary translator allocates all internal data structures on random addresses using an internal `mmap` implementation. On IA-32 the instruction pointer cannot be read directly (e.g., through a register) and indirect control transfers (e.g., `call` instructions) are replaced by a secure sequence of virtualized instructions during the translation process. All translated indirect control flow transfers point into the original code region and are replaced with a lookup in the mapping table. The translated code is therefore unable to recover a pointer into the code cache or any other internal data structure of the binary translator. The internal `mmap` implementation uses the address space layout randomization feature that is available in the Linux kernel [7, 8, 30]. Address space layout randomization is exploitable if used in isolation [36] but all the exploits rely on some form of indirect control flow transfers and return to libc attacks. This guard relies on other guards to be secure. But the guard is nevertheless effective in raising the complexity for potential exploits.

The following security guards can be enabled on demand and are not part of the default configuration.

Section guard: only (direct and indirect) function calls and function returns are allowed to transfer control to a function in a different code region. All other control transfers (like jumps or indirect jumps) are verified to target the code of the same section. This guard prohibits unintended control flow transfers.

Call guard: call instructions are verified to transfer control to an existing function by checking the exported symbols of loaded objects. If the call does not target a symbol defined in any of the loaded libraries or the program itself then the call is not allowed. This guard prohibits arc attacks and the redirection of function pointers to unintended code.

Protection of internal data structures: adds an additional heavyweight guard that uses `mprotect` to disable write access to all internal data structures (e.g., mapping table, code cache, internal translator data) of the binary translator whenever translated code is executed. This way translated code is unable to change translated code or any other internal data structures of the binary translator. Even if this guard is not active all pointers to the internal data structures are pruned from the stack. An exploit is unable to detect the internal data structures due to the virtualization guarantees of the translator.

The current guard configuration does not allow applications with self-modifying code. An application with, e.g., a JIT compiler could be handled by specifying at exactly what regions the compiler will be emitting code and using an additional guard for these regions.

3.2 Policy-based system call interposition

All the potentially dangerous functionality of a program is performed by system calls (e.g., I/O, network sockets, privilege escalation). A mechanism that restricts a program's use of system calls is a useful and important extension to fault isolation. Code based exploits are handled by the software-based fault isolation. Data driven attacks where no malicious code is executed (e.g., integer overflows and type errors) are caught whenever a system call is executed that does not conform to the application's policy.

Policy-based system call interposition relies on *software-based fault isolation* and the rewriting and replacement of system calls. All system calls through both *sysenter*, and *int 80* instructions (Linux uses and supports both systems [21]) are rewritten by the binary translator to execute a validation function before they are allowed. The sandbox offers an extensible *system call interposition framework* that makes it possible to allow or disallow system calls based on the call stack, the system call number, and the parameters.

The sandbox validates system calls through handler functions and by a policy that is loadable at runtime. A policy has the advantage that combinations of allowed and disallowed parameters can be specified in a simple way. Handler functions on the other hand enable in-depth verification of arguments and can use state (e.g., a list of previous *mmap* calls, arguments, and call locations) to track application behavior throughout the execution of different system calls. The combination of a policy to handle simple and static combinations of system calls and handler functions for complex system calls enables an even tighter and more dynamic security model than policies alone.

3.2.1 Special handler functions

The privileges of a program can be managed by specific handler functions on a per system call basis. Every system call can use a different handler function that analyzes *call stack* and *arguments*. The handler functions are a part of the binary translator and have full control over the application. Handler functions may *allow* the system call, *abort* the program, or redirect the system call and return a *fake value*.

These redirected system calls can be used to implement different functionality in user-space. If a system call is redirected then a user-space function is executed whenever the system call is called. This function runs in the context of the binary translator and can execute arbitrary other system calls (redirected system calls can, e.g., emulate or isolate vulnerable system calls). More generally redirected system calls add additional validation of arguments that are passed to the kernel.

The sandbox uses additional handler functions to check all *mmap*, *mprotect*, *open*, and *openat* system calls. For *mmap* and *mprotect* the sandbox checks if the arguments overlap or touch any internal data structures that the binary translator uses. If there is a conflict then the application is terminated. For the *open* and *openat* system calls the sandbox uses *stat* to check

```
mode:whitelist /* deny unlisted syscalls */
open("/dev/arandom", 0_RDONLY):allow
open("/dev/urandom", 0_RDONLY):allow
time(null):allow
getuid32():return(0) /* return static uid=root */
close(*):allow /* close open files */
write(1,*,*):allow /* stdout */
access("/etc/*,*"):allow
// implicit: access("*,*"):deny
```

Figure 3. Example excerpt from a policy file that uses white listing as default policy. It allows two specific files to be opened, execution of the *time()*, *close()*, and *write()* system calls. *getuid()* returns 0 (root), and *access()* is restricted to */etc/** only.

if the file is in the black list or tries to access protected files like */prof/self/maps* that would leak information about the sandbox.

The handler functions are used as an extension of the policy system. Handler functions enable additional control logic to guard and tighten the allowed actions of the application.

3.2.2 Policy-based system call authorization

The *system call authorization framework* for policy-based system call authorization builds on *process sandboxing* and extends the system call interposition framework. The sandbox loads a user-defined policy at startup to decide for each individual system call if it is allowed or not.

If a system call and its arguments do not match the policy (e.g., the configuration is not present in the policy for white-listing, or is present for black-listing), the isolation system assumes that there is an error, bug, or security problem and terminates the user process. Additionally it can signal the system operator that an authorization fault occurred.

The policy file contains a list of system calls and parameter-sets that are allowed or denied. This allows a combination of white-listing and black-listing of different argument combinations per system call. Arguments are encoded as integers, pointers, strings, null, or asterisk for an unspecified value that matches any input. Partial strings can be matched with an appended asterisk (e.g., */etc/apache2/**). Effective path arguments can additionally be evaluated using *stat* system calls. Possible actions for each combination are to allow the system call, to abort the program, or to return a predefined integer value. See Figure 3 for an excerpt from a policy file and Figure 4 for an overview of runtime data structures needed for the policy-based authorization.

The redirected system calls can be used to change and test the behavior of a program if certain system calls return special values (e.g., many programs behave differently if they are run as root, so returning a fake value for *getuid* is useful in some cases). The current policy is limited to returning a static fake value. But the system call interposition framework can be used to call any user supplied function to handle a specific system call. This functionality enables, e.g., additional stateful security checks, the emulation of (obsolete or unsafe) system calls, virtualization or reimplementations of specific system calls in user space, or can be used for whole kernel emulation in user-space.

Depending on the first line of the policy file either white-listing or black-listing is used. Black-listing can be used, e.g., for testing or implementation of new features. For security policies we assume that a white-listing approach is taken. An unmatched combination of parameters for a specific system call either aborts the program if white-listing is used, or is allowed if black-listing is used. White-listing specifically allows system calls and implicitly denies all other system calls. Black-listing denies or redirects specific system calls and implicitly allows all unspecified system calls.

3.2.3 TOCTTOU attacks

Time Of Check To Time Of Use (TOCTTOU) [40] attacks rely on the fact that a second thread can replace the arguments on the stack of another thread after they have been checked by the interposition framework but before the system call is executed.

Each thread of an application is guarded by a sandbox. The security guards inside each sandbox prohibit the execution of malicious code. A thread that tries to execute an illegal control flow transfer or illegal code terminates the application immediately. This setup prevents a second thread from using injected code to rewrite the system call arguments after they are checked by the policy-based system call interposition.

A hypothetical exploit could use a data-based exploit in one thread to corrupt a data-structure that is used in both threads after the system call interposition framework validated the arguments. To guard against such an exploit, we must restrict access to a thread's stack by other threads. The randomness introduced by the thread scheduler and random stack locations suffices in many scenarios to limit the risk of such an exploit.

The combination of the additional fine-grained checks of the additional security guards in the isolation layer and the policy-based system call interposition make TOCTTOU attacks based on code injection impossible.

4. Implementation

libdetox uses fastBT [31, 32], a generator for binary translators to generate a lean and efficient table-based user-space binary translator that follows the description in Section 2. The generated binary translator is extended by the security guards to enforce fine-grained

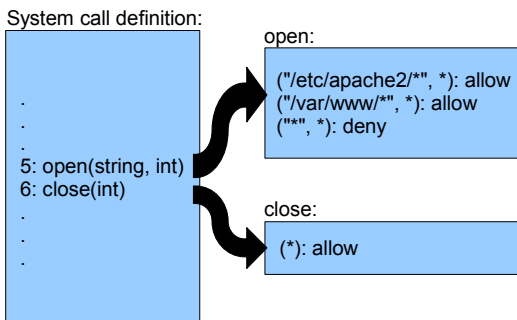


Figure 4. Runtime data structures for a given policy with examples for the `open()` and `close()` system calls.

user-space fault isolation and to implement the system call interposition framework.

This section discusses the implementation of the software-based fault isolation mechanism and highlights the changes needed to implement a secure binary translator. The policy-based authorization framework is then implemented on top of the software-based fault isolation.

4.1 Software-based fault isolation

The four binary translators that we evaluated for user-space fault isolation and the system call interposition framework are PIN [28], HDTrans [37, 38], DynamoRIO [9, 10, 23], and fastBT [31, 32]. These binary translators use different approaches to implement user-space binary instrumentation and isolation. HDTrans and fastBT use a table-based approach that translates instructions based on translation tables. DynamoRIO and PIN translate the machine code into a high level intermediate representation (IR) and compile the translated IR back to machine code.

Important features are that the binary translator must cover the complete IA-32 instruction set and that no pointers to thread local data structures are left on the stack if the binary translator uses the same stack as the user-program. To offer an attractive alternative to full system translation the overhead for the binary translation must be low, both for the translation of new instructions and for the execution of translated code.

libdetox uses fastBT because (i) the complete IA-32 instruction set is supported, (ii) the binary translator is very modular and extendable by new handler functions that control the low-level translation of control instructions, and (iii) the trusted computing base is small. libdetox consists of only a couple of thousand lines of code.

The binary translator framework translates all control flow instructions and ensures that the execution stays inside the code cache.

All internal data is allocated on a thread-local basis. Each thread has a private code cache and runs its own translator. This removes the need for inter-thread synchronization.

libdetox uses fastBT as a tool for software-based fault isolation. The security concepts presented in this paper are not specific to fastBT, but can be implemented in any table-based binary translator. Details of the implementation of the binary translator are available in [32]. The additional security guards that are discussed in 3.1.3 are an extension implemented on top of the binary translator and crucial to guarantee isolation.

The combination of fastBT with the additional security guards ensures the first security principle defined in Section 3. Illegal instructions and instructions that redirect control flow to malicious code lead to a direct abort of the program through the binary translator.

4.1.1 Thread and process handling

All system calls that create threads or new processes are handled in a special way. If the arguments are allowed according to the policy then the system call is instrumented such that the new thread or process is started in a new libdetox instance.

If a new thread is started then a trampoline is executed first that starts a new instance of the binary translator that controls the execution of the new thread. For each new process that is started the binary translator is injected with a `LD_PRELOAD` directive (if the started program supports dynamically shared libraries, otherwise the process creation is aborted).

4.1.2 Additional guards

libdetox reads all symbol and section information when the program or shared libraries are loaded. This information is imported into data structures of libdetox that are used at runtime to implement the additional security guards.

The guards from Section 3.1.3 are executed either when new code is translated or whenever the translated instruction is executed in the application. Executable space protection, executable bit removal, and signal handling are implemented using static implementations during the translation process. The section guard, call guard, and return address verification need both a check during the translation and an additional dynamic check. The check during the translation process is used for fixed or precomputed targets and the dynamic check is patched into the translated code for all dynamic control flow transfers.

Secure context transfers are implemented through changes in the binary translator. The optimizations for indirect control flow transfers are modified so that no pointers to the code cache are left on the stack of the user-program. For example a return instruction is translated into code that (i) executes a lookup in the mapping table, (ii) stores the translated target address in a local data structure, and (iii) uses an indirect jump through that data structure to redirect the control flow to the translated target. Using such trampolines guarantees that pointers to the code cache are never left on the application stack and there is no need to overwrite return addresses of the original application which would leak information about the sandbox.

4.2 Policy-based system call authorization

All system calls through interrupts and the `sysenter` instruction are rewritten by the binary translator. The system call interposition framework is implemented on top of the binary translator to wrap all system calls into individual evaluation functions. The system call interposition framework then checks the system call and its arguments against the loaded policy. libdetox loads the policy and parses it into an array of parameter lists. Per system call a parameter list is generated with combinations of valid parameters. If the user program wants to execute a system call then the list is checked. If a parameter-set matches then the system call is either executed or a fake value is returned. The process is terminated if no parameter-set matches. This enforces the second security principle. The combination of these two principles makes user-space isolation and encapsulation possible and secure.

5. Evaluation

This section evaluates the libdetox user-space virtual machine. Low overheads for isolation and sandboxing features show that the libdetox approach is highly attractive. The discussion about system

call coverage shows that most programs execute a low number of specific system calls with a limited set of arguments.

The Apache case study in Section 5.3 shows that it is possible to isolate Apache in a user-space sandbox that implements a hard security policy with low overhead.

The benchmarks are run under Ubuntu 9.04 on an E6850 Intel Core2Duo CPU running at 3.00GHz, 2GB RAM, and GCC version 4.3.3. Averages are calculated by comparing overall execution time for all programs of untranslated runs against translated runs. The SPEC CPU2006 benchmarks are presented as a way to compare performance with other systems.

5.1 Isolation and sandboxing overhead

This section provides an analysis of the runtime overhead introduced through libdetox. The overhead is separated into (i) BT overhead alone, (ii) overhead for system call validation and executable space protection, and (iii) full protection using `mprotect` to guard the internal datastructures from attacks against the sandbox.

Table 1 displays overheads for all SPEC CPU2006 benchmarks compared to an untranslated run. The different configurations are:

BT: A configuration without additional security features, showing the overhead of the isolation and binary modification toolkit.

libdetox: This configuration shows libdetox's overhead with the default guards enabled.

libdetox+mprot: The last configuration shows full encapsulation including protection of internal data structures using explicit memory protection through `mprotect`.

Table 1 uses the standard SPEC CPU2006 benchmarks and shows the overhead for long running programs. The average slowdown for binary translation (and no other transformation) for the full SPEC CPU2006 benchmarks is 6.0%. The libdetox security extensions increase the overhead to 6.4%. The full protection mechanism results in an overhead of 8.2%. The overhead for binary translation and basic libdetox protection for most programs is between -3.5% and 4.0%; some benchmarks like 400.perlbench, 433.gcc, 453.sjeng, 483.xalanabr, 447.dealII, and 453.povray result in a higher overhead of 23% to 60% due to many indirect control flow transfers that cannot be optimized. The speedup of some programs is achieved by a better code layout through the translation process. libdetox adds static overhead per translated block and per system call. The SPEC CPU2006 benchmarks have a low number of system calls and high code reuse, which is typical for server applications. Therefore the libdetox extensions add no measurable overhead to these programs.

libdetox with full protection leads to more overhead (8.21% on average) because the number of system calls increases. But the overall overhead is low for these benchmarks, although the translation overhead is higher. The translation overhead is still small compared to the runtime of the translated program. As soon as all active code is translated, no further memory protection calls are necessary.

5.2 System call coverage

Figure 5 shows a policy that covers all SPEC CPU2006 benchmarks. This policy is not secure and only used to evaluate the over-

Benchmark	BT	libdetox	+mprot
400.perlbench	55.97%	59.88%	74.69%
401.bz2	3.89%	5.39%	5.54%
403.gcc	20.86%	22.68%	55.56%
429.mcf	-0.49%	0.49%	0.25%
445.gobmk	18.17%	14.57%	16.69%
456.hmmer	4.64%	4.75%	5.72%
458.sjeng	24.62%	27.65%	31.22%
462.libquantum	0.98%	0.98%	0.98%
464.h264ref	6.17%	9.20%	9.20%
471.omnetpp	13.91%	14.11%	15.12%
473.astar	3.66%	3.83%	4.33%
483.xalancbmk	23.72%	27.22%	31.27%
410.bwaves	2.12%	2.68%	3.91%
416.gamess	-3.50%	-4.20%	-0.70%
433.milc	0.97%	2.18%	3.26%
434.zeusmp	-0.13%	-0.25%	0.13%
435.gromacs	0.00%	0.00%	0.00%
436.cactusADM	0.00%	-0.66%	0.00%
437.leslie3d	0.00%	0.00%	0.86%
444.namd	0.65%	0.65%	0.65%
447.dealII	44.20%	41.12%	43.66%
450.soplex	7.25%	5.02%	7.25%
453.povray	22.10%	25.14%	26.52%
454.calculix	-1.68%	-0.56%	-1.12%
459.GemsFDTD	1.79%	1.79%	2.68%
465.tonto	9.19%	10.27%	12.43%
470.lbm	0.00%	0.00%	-0.11%
482.sphinx3	2.36%	2.25%	1.89%
Average	6.00%	6.39%	8.21%

Table 1. Overhead for different configurations executing the SPEC CPU2006 benchmarks (relative to an untranslated run). +mprot: libdetox with full memory protection.

head of policy-based user-space software-based fault isolation. The policy is a summary of all individual policies for each SPEC benchmarks so that the overhead for all benchmarks can be evaluated in a single run of the SPEC benchmark script. Differences to real policies include the over-generalization of attributes and the lax handling of the open, unlink, mmap2, unlink, and stat64 system calls. A production policy would tighten the policy for a single program and explicitly list all needed files and directories or restrict these system calls to specific directories. These system calls are used to access many data files in each individual benchmark and for each data size. The long list of explicit configurations was abbreviated through over-approximation to give a clearer picture. A safe policy for a single specific SPEC benchmark does not result in any measurable additional overhead.

Figure 5 shows that all SPEC CPU2006 benchmarks need no more than 38 different system calls with a few more individual parameter configurations.

The second case study shows *nmap*, which is a network exploration and security tool that checks and fingerprints running services of servers over the Internet. libdetox virtualizes and encapsulate version 4.53 of *nmap* into a secure sandbox. The policy shown in Figure 6 shows a set of rules that restricts *nmap* to a few different

```

mode:whitelist /* not listed: abort program */
brk(*):allow /* memory management */
mmap2(null,*, PROT_READ | PROT_WRITE, \
MAP_ANONYMOUS | MAP_PRIVATE, -1,*):allow
mremap(*,*,*, MREMAP_MAYMOVE):allow
munmap(*,*):allow
execve("/bin/echo",*,*):allow /* allowed prog.s */
execve("/opt/cpu2006/bin/echo",*,*):allow
execve("/sbin/echo",*,*):allow
execve("/usr/bin/echo",*,*):allow
execve("/usr/local/bin/echo",*,*):allow
execve("/usr/local/sbin/echo",*,*):allow
execve("/usr/sbin/echo",*,*):allow
clone(*,null,0,null):allow /* allowed file I/O */
close(*):allow
dup(*):allow
fcntl64(*,*):allow
fstat64(*,*):allow
ftruncate64(*,*):allow
getcwd("*,*"):allow
ioctl(*,*):allow
llseek(*,*,*,*, SEEK_SET):allow
llseek(*,*,*,*, SEEK_CUR):allow
lseek(*,*, SEEK_SET):allow
lseek(*,*, SEEK_CUR):allow
lstat64("/opt/cpu2006/benchspec/CPU2006/*", \
*):allow
open("*,*"):allow /* relaxed for spec */
pipe(*):allow
read(*,*,*):allow
stat64("*,*"):allow
rmdir("foo"):allow /* remove foo directories */
unlink("*,*"):allow /* unlink relaxed for spec */
write(*,*,*):allow
writev(*,*,*):allow
futexp(*, FUTEX_PRIVATE | FUTEX_WAKE, 0x7FFFFFFF, \
null,*,*):allow /* process mgmt */
waitpid(*,*,0):allow
rt_sigprocmask(SIG_BLOCK,*,*):allow /* signals */
rt_sigprocmask(SIG_SETMASK,*,null):allow
getegid32():allow /* information retrieval*/
geteuid32():allow
getgid32():allow
getrusage(RUSAGE_SELF,*):allow
gettimeofday(*,null):allow
getuid32():allow
setrlimit(RLIMIT_DATA,*):allow
ugetrlimit(RLIMIT_DATA,*):allow
nanosleep(*,*):allow /* sleep and time */
time(null):allow
times(*):allow

```

Figure 5. A relaxed policy to measure the sandboxing overhead for the SPEC CPU2006 benchmarks. Some rules are relaxed to facilitate the run of the spec benchmark scripts.

system calls, e.g., opening any network connection. The *nmap* program uses 23 different system calls, individual parameters are used to open 15 different files, use *stat64* on 6 files, and use *access* for two files. This policy sandboxes the network scanner, and an attacker cannot escalate privileges if one of the many *nmap* detection modules contains exploitable code.


```

mode:whitelist /* not listed: abort program */
brk(*):allow /* memory management */
/* due to shared libraries all mmap calls must be
   additionally checked in a handler function for
   a set exec bit. */
mmap2(*,*,*,*,*,*):allow
munmap(*,*):allow
futex(*,*,*,*,*,*):allow /* thread futexes */
access("/etc/ld.so.nohwcap",*):allow /* limit I/O */
access("/usr/share/nmap/nmap-services",*):allow
close(*):allow
fcntl64(*, F_GETFL):allow
fcntl64(*, F_GETFD):allow
fcntl64(*, F_SETFL, O_RDWR | O_NONBLOCK):allow
fstat64(*,*):allow
ioctl(*, TIOCGPRG, *):allow
llseek(*,*,*,*,*):allow
newselect(*,*,*,*,*):allow
open("/dev/arandom",*):allow
open("/dev/tty",*):allow
open("/dev/urandom",*):allow
open("/etc/host.conf",*):allow
open("/etc/hosts",*):allow
open("/etc/ld.so.cache",*):allow
open("/etc/localtime",*):allow
open("/etc/nsswitch.conf",*):allow
open("/etc/passwd",*):allow
open("/etc/resolv.conf",*):allow
open("/lib/i686/cmov/libnsl.so.1",*):allow
open("/lib/i686/cmov/libnss_compat.so.2",*):allow
open("/lib/i686/cmov/libnss_files.so.2",*):allow
open("/lib/i686/cmov/libnss_nis.so.2",*):allow
open("/usr/share/nmap/nmap-services",*):allow
read(*,*,*):allow
stat64("/etc/localtime",*):allow
stat64("/etc/resolv.conf",*):allow
stat64("/home/test/.nmap/nmap-services",*):allow
stat64("./nmap-services",*):allow
stat64("/usr/lib/nmap/nmap-services",*):allow
stat64("/usr/share/nmap/nmap-services",*):allow
write(*,*,*):allow
socketcall(PF_NETLINK, SOCK_RAW, 0):allow /* net */
socketcall(PF_INET, SOCK_STREAM, IPPROTO_TCP):allow
socketcall(PF_FILE, SOCK_STREAM | \
    SOCK_CLOEXEC | SOCK_NONBLOCK, 0):allow
geteuid32():allow /* system information */
gettimeofday(*,*):allow
getuid32():allow
time(*):allow
uname(*):allow
ugetrlimit(*,*):allow
setrlimit(RLIMIT_NOFILE, *):allow

```

Figure 6. Full policy covering and encapsulating the nmap network scanner and all additional detection modules. Network access is allowed as well as access to libraries and configuration files.

5.3 Apache isolation

The Apache 2.2.11 HTTP server is used to benchmark a daemon that needs both access to local files and is accessible over the network. libdetox encapsulates the Apache processes and threads and only allows few system calls with restrictive parameter con-

figurations. Like the nmap policy in Figure 6 Apache is only allowed to open specific files and access files in two directories (/etc/apache2, and /var/www) and is not allowed to execute other processes. On the other hand the daemon process is free to open connections over the network.

Benchmark	native	BT	libdetox
test.html	84.83s	97.47s	101.34s
	22.48Mb/s	19.57Mb/s	18.82Mb/s
phpinfo.php	84.40s	98.63s	101.34s
	3.28Mb/s	2.8Mb/s	2.73Mb/s
picture.png	249.87s	261.92s	266.98s
	945.18Mb/s	901.67Mb/s	884.6Mb/s
Avg. overhead	-	9.29%	12.06%

Table 2. The ab benchmark is used to compare a native run without isolation to fast binary translation only, and libdetox with user-space fault isolation and policy-based system call authorization.

The overhead for the Apache daemon was measured using the ab Apache benchmark which used 10 concurrent instances to receive each file 1'000'000 times. Table 2 shows the overheads using different configurations. The test uses the following files: (i) test.html, a static html file with 1.7kB, (ii) phpinfo.php, a small php file that issues the phpinfo call, and (iii) picture.png, a 242kB file.

The overhead to download a small file is 14.9% for binary translation because of the high number of system calls needed to open, read, and send the file. The overhead for libdetox is 19.5%. For larger files, as seen by the numbers for picture.png, the overhead of binary translation is 4.83% and 6.85% for libdetox.

An interesting feature is the throughput difference between small and large files. Throughput is increased from 22.48Mb/s to 945.18Mb/s for native runs and even more for libdetox, namely from 19.57Mb/s to 901.67Mb/s, which is more than 46 times faster compared to the small file.

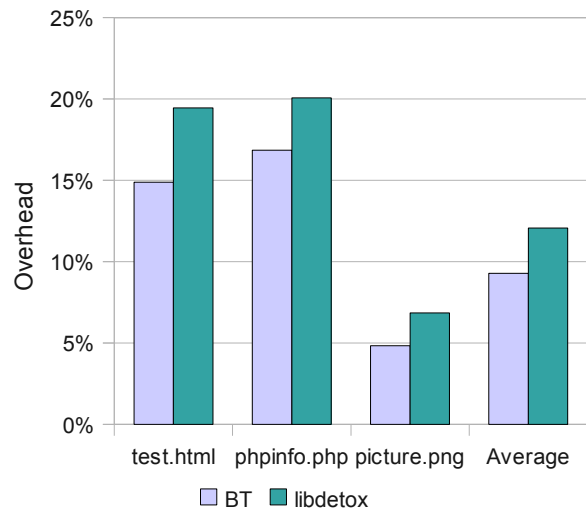


Figure 7. Overhead introduced through isolation, sandboxing, and policy-based authorization for the Apache benchmarks

Figure 7 shows the overhead introduced through isolation and sandboxing for the different Apache benchmarks. The numbers show that libdetox introduces a moderate overhead of about 20% for small static and small dynamic files. For larger files the overhead drops to below 7%. The SPEC CPU2006 benchmarks in Section 5.1 showed that the overhead for compute intensive programs is even lower. The average overhead of libdetox for the Apache web-server results is 12.06%, which makes user-space fault isolation attractive.

6. Related work

Related work to the presented approach combines ideas from different fields of research. An important area are systems that enforce some kind of security policy by either limiting the instruction set or relying on some kernel infrastructure. Security can be enforced on many levels. Some of them are limiting the system calls a program can use, others limit the instruction set inside the application, or use hardware extensions to limit the program.

An important characteristic of security systems is the granularity they work on. Some systems use full system translation to encapsulate and virtualize a complete running system. This offers strong security guarantees as hardware virtualization extensions can be used to separate different virtualized instances. A drawback of these solutions is data exchange between programs running on different virtualized instances and administrative overhead that is needed to configure and secure every single instance. Process encapsulation on the other hand limits the code and system calls a running process can execute.

This section discusses different alternate approaches to guarantee security. Full system translation offers the possibility to encapsulate complete systems and to guarantee security on a coarse-grained system level. System call interposition evaluates an application's system calls and offers coarse-grained interposition on the granularity of system calls. User-space isolation through dynamic binary translation offers a fine-grained level of control about all executed control flow transfers. Static binary translation limits the instruction set and uses a static checker that validates the application before it is run.

6.1 Full system translation

Full system translation virtualizes a complete system, including the operating system and all hardware. Complete system virtualization by QEMU [6] offers full encapsulation, but comes with high overhead. Other full system virtualization tools like VMware [11, 15] and Xen [4] rely on kernel or hardware support. Livewire [20] extends VMware to build an intrusion detection system around the virtual machine. Ho et al. [25] present a protection system that uses a Xen based virtualization approach that uses QEMU to emulate individual instructions based on taint information.

A disadvantage of full system virtualization is that every virtual machine is an independent system with its own configuration and support needs. Additionally only very coarse-grained events can be observed from such a high level of abstraction. From a security and safety perspective the encapsulation of such an approach is needed but without the complexity of administrating individual systems. Our sandbox offers user-space process isolation, combining encapsulation with fine-grained security on a single system.

6.2 System call interposition

System call interposition uses either a kernel module or ptrace support to implement a control mechanism on the level of system calls. These systems share several drawbacks. For one the protection mechanism is very coarse-grained and they do not detect the execution of malicious code until a system call that is not part of the policy is executed. These systems miss exploits that target opened files or use the policy's allowed system calls and are often prone to TOCTTOU attacks. These file changes evade detection of the security systems that validate purely on system calls. A second concern is the overhead introduced due to context switching. An expensive context switch has to be performed whenever a policy rule is checked. A third drawback is that these systems rely on trusted code in the kernel to stop the monitored program which poses an additional security risk.

Janus [22] is a system call interposition framework that uses the Solaris process tracing facility (ptrace) to allow one user mode process to filter the system calls of a second process. This framework builds on kernel support and has two drawbacks: (i) the traced application is already in the kernel when it is stopped, a situation that poses a potential security problem, and (ii) the overhead of switching between an inspecting process and the corresponding application is high. MAPbox [1], AppArmor [5], SubDomain [13], and Consh [2] extend the idea of a ptrace interposition framework by implementing policy-based authorization. Tal Garfinkel analyzed practical problems of system call interposition in [18].

The Linux kernel offers an API for security modules [41] that can be used to implement many kernel-based coarse-grained security extensions. Systrace [33] uses a kernel module to implement a system call policy. Some global system calls are validated in the kernel with low overhead, for all other system calls the program is stopped and a user-space daemon decides based on the parameters if the system call is allowed or not. Switchblade [16] enforces a system call policy using an in-kernel system call model and dynamic taint analysis. Ostia [19] prevents TOCTTOU attacks by using a proxy and a delegation model to delegate system calls to different processes or threads. Alcatraz [27] is a hybrid isolation approach that offers unrestricted read access to a sandboxed application but redirects all writes to a buffer which can be examined before it is committed. Unfortunately this approach does not protect against data leaked over the network or processes that use local root exploits to gain privileges.

Our sandbox uses a fine-grained level of control that checks individual instructions and makes it impossible to execute injected code. Each thread of an application runs in its own sandbox. Each sandbox is secured against the execution of malicious code, therefore a second thread cannot execute code that races between the system call argument check and the execution of the system call thereby removing the threat of TOCTTOU attacks.

6.3 User-space isolation through dynamic binary translation

Vx32 [17] implements a user-space sandbox built on BT that uses segmentation to hide the internal data structures. Due to the use of segmentation the Vx32 system is limited to 32-bit code. Interrupts, system calls, and illegal instructions are translated to traps that call special handler functions. The proposed BT results in a high overhead as there are no optimizations for indirect control transfers.

The traps for system calls offered by Vx32 are targeted towards a reimplementing of the system calls and are not intended for a policy-based system call authorization framework.

Strata [34, 35] is a safe virtual execution environment using software dynamic translation. It uses dynamic binary translation to isolate user-space programs and implements a basic system call interposition API. This API is used to instrument individual system calls. The translation framework is neither limited to a single system nor to a single architecture. Strata uses binary translation to enforce a non-executable stack but there are no additional security guards that limit return to libc attacks or heap based overflows.

Program shepherding [26] uses the DynamoRIO [9, 10, 23] framework to safeguard running applications. A single policy is hardcoded and enforced using binary translation. The binary translator adds additional checks to restrict code origins and to control the targets of indirect control transfers.

Our approach implements a user-space sandbox and extends this sandbox with additional security guards that check the execution of application code at runtime. System calls in our approach are not replaced by software traps but are validated through a policy-based system call authorization framework. Additionally libdetox does not depend on specific hardware features like segmentation, a feature that is not present on AMD64 and hinders portability. The policies used in the sandbox can be refined and changed without the need to recompile the safe execution platform.

6.4 Security through static binary translation

The Google Native Client [42] executes x86-code in a sandbox. The native client uses the same instruction padding techniques as presented in the software-based fault isolation [39] system by Wahbe et al.. The instruction set is limited to a safe subset of the IA-32 ISA, making illegal operations impossible. A verifier checks if the program is valid before the program is executed without any additional isolation. Such a system limits the possible range of used instructions, the programs must be linked statically, and no dynamic libraries can be used. Programs must be compiled with a custom-tailored compiler and special libraries.

PittSField [29] implements a static binary translation and checking tool used for software-based fault isolation. The static rewriting algorithm (i) aligns targets for control transfers on 16 byte boundaries, (ii) changes control transfer instructions so that targets are always 16 byte aligned, and (iii) separates data and code regions by adding additional instructions to force pointers to point to data or code. This static translation results in an overhead of 13% for instruction alignment and 21% for data verification as reported in [29].

The PittSField approach only verifies that a program executes valid instructions and writes to the correct data regions. Our sandbox offers the executable space protection mechanism which writes protects application and library code and implements full protection of the internal data structures using kernel memory protection. Additionally our sandbox approach offers a system call interposition framework which validates every single system call and its arguments.

The PittSField approach validates that a return address on the stack is in the code region and not in the data region. But a carefully designed return to libc attack is possible. Our approach disables

return to libc attacks by only allowing the execution of a safe subset of system calls that are defined by a custom-tailored policy, checking return addresses, and verifying all control transfers.

7. Conclusion

We present an approach to low overhead software-based fault isolation that implements fine-grained security in user-space. This approach limits programs in their use of system calls and execution of privileged instructions through flexible per-process policies and a configurable system call interposition framework.

An implementation prototype of our approach called libdetox uses dynamic binary translation to support the full IA-32 ISA without kernel support. Full binary translation adds security guards, detects code injections, guards dangerous instructions at runtime and interposes system calls with an authorization framework. System calls are validated based on individual handler functions for special system calls and a policy that allows to control the allowed parameters on a per system call basis.

The approach presented here is attractive for many scenarios that look for a low-cost and widely useable way to secure a system. Applications are isolated and encapsulated while using a shared system image with a single system configuration, and there is no need to virtualize a complete system. Our approach is the first virtual execution system that combines a fast and efficient software-based translator with additional guards and a policy-based system call interposition framework. This combination results in a low overhead software-based fault isolation and encapsulation system. Such a system then can guard daemon processes like the Apache web-server to prevent unwanted access to system resources.

As users (and system administrators) look for ways to deal with the wide range of security problems, libdetox presents a simple yet highly attractive approach to protect a system against a wide range of attacks.

References

- [1] ACHARYA, A., AND RAJE, M. MAPbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium* (2000).
- [2] ALEXANDROV, A., KMIEC, P., AND SCHAUSER, K. Consh: Confined execution environment for internet computations. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.488> (1999).
- [3] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference* (2000).
- [4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03* (New York, NY, USA, 2003), pp. 164–177.
- [5] BAUER, M. Paranoid penguin: an introduction to novell apparmor. *Linux J.* 2006, 148 (2006), 13.
- [6] BELLARD, F. QEMU, a fast and portable dynamic translator. In *ATEC '05* (Berkeley, CA, USA, 2005), pp. 41–41.

The source code of the libdetox framework and additional examples can be downloaded at <http://nebelwelt.net/projects/libdetox>.

- [7] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.
- [8] BHATKAR, S., BHATKAR, E., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium* (2005).
- [9] BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. Design and implementation of a dynamic optimization framework for Windows. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-4)* (2001).
- [10] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (Washington, DC, USA, 2003), pp. 265–275.
- [11] BUGNION, E. Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache. US Patent 6704925, March 2004.
- [12] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium* (2001).
- [13] COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., AND GLIGOR, V. Subdomain: Parsimonious server security. In *LISA '00: Proceedings of the 14th USENIX conference on System administration* (2000).
- [14] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium* (1998).
- [15] DEVINE, S. W., BUGNION, E., AND ROSENBLUM, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6397242.
- [16] FETZER, C., AND SUESSKRAUT, M. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), ACM, pp. 273–286.
- [17] FORD, B., AND COX, R. Vx32: lightweight user-level sandboxing on the x86. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 293–306.
- [18] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 163–176.
- [19] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (February 2004).
- [20] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (February 2003).
- [21] GARG, M. Sysenter based system call mechanism in linux 2.6 (<http://manugarg.googlepages.com/systemcallinlinux2.6.html>).
- [22] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Usenix Security Symposium* (1996).
- [23] HAZELWOOD, K., AND SMITH, M. D. Managing bounded code caches in dynamic binary optimization systems. *TACO '06* 3, 3 (2006), 263–294.
- [24] HIROAKI, E., AND KUNIKAZU, Y. propolice : Improved stack-smashing attack detection. *IPSJ SIG Notes* 2001, 75 (2001-07-25), 181–188.
- [25] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. vol. 40, pp. 29–41.
- [26] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.
- [27] LIANG, Z., SUN, W., VENKATAKRISHNAN, V. N., AND SEKAR, R. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Inf. Syst. Secur.* 12, 3 (2009), 1–37.
- [28] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05* (New York, NY, USA, 2005), pp. 190–200.
- [29] MCCAMANT, S., AND MORRISSETT, G. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium* (Vancouver, BC, Canada, August 2–4, 2006), pp. 209–224.
- [30] PAX-TEAM. PaX ASLR (Address Space Layout Randomization). <http://pax.grsecurity.net/docs/aslr.txt>.
- [31] PAYER, M., AND GROSS, T. R. Requirements for fast binary translation. In *2nd Workshop on Architectural and Microarchitectural Support for Binary Translation* (2009).
- [32] PAYER, M., AND GROSS, T. R. Generating low-overhead dynamic binary translators. In *SYSTOR '10* (2010).
- [33] PROVOS, N. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2003), USENIX Association, pp. 18–18.
- [34] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. Tech. rep., Charlottesville, VA, USA, 2001.
- [35] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. *Computer Security Applications Conference, Annual 0* (2002), 209.
- [36] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS'04* (2004), pp. 298–307.
- [37] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALÉ, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News* 35, 1 (2007), 135–140.
- [38] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALÉ, P. P. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE '06* (New York, NY, USA, 2006), pp. 175–185.
- [39] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP'93* (New York, NY, USA, 1993), ACM, pp. 203–216.
- [40] WATSON, R. N. M. Exploiting concurrency vulnerabilities in system call wrappers. In *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies* (2007).
- [41] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium* (2002).
- [42] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. *IEEE Symposium on Security and Privacy* (2009), 79–93.