2012 IEEE Symposium on Security and Privacy

Safe Loading - A Foundation for Secure Execution of Untrusted Programs

Mathias Payer
ETH Zurich, Switzerland

Tobias Hartmann
ETH Zurich, Switzerland

Thomas R. Gross
ETH Zurich, Switzerland

***Abstract*—The standard loader (`ld.so`) is a common target of attacks. The loader is a trusted component of the application, and faults in the loader are problematic; e.g., they may lead to local privilege escalation for SUID binaries.**

Software-based fault isolation (SFI) provides a framework to execute arbitrary code while protecting the host system. A problem of current approaches to SFI is that fault isolation is decoupled from the dynamic loader, which is treated as a black box. The sandbox has no information about the (expected) execution behavior of the application and the connections between different shared objects. As a consequence, SFI is limited in its ability to identify devious application behavior.

This paper presents a new approach to run untrusted code in a user-space sandbox. The approach replaces the standard loader with a security-aware trusted loader. The secure loader and the sandbox together cooperate to allow controlled execution of untrusted programs. A secure loader makes security a first class concept and ensures that the SFI system does not allow any unchecked code to be executed. The user-space sandbox builds on the secure loader and subsequently dynamically checks for malicious code and ensures that all control flow instructions of the application adhere to an execution model.

The combination of the secure loader and the user-space sandbox enables the safe execution of untrusted code in user-space. Code injection attacks are stopped before any unintended code is executed. Furthermore, additional information provided by the loader can be used to support additional security properties, e.g., inlining of Procedure Linkage Table calls reduces the number of indirect control flow transfers and therefore limits jump-oriented attacks.

This approach implements a secure platform for privileged applications and applications reachable over the network that anticipates and confines security threats from the beginning.

I. Introduction

Secure execution of applications in user-space remains a hard problem. Software-based fault isolation (SFI) has been embraced by many projects [29], [43], [42], [22], [37] to address this problem. SFI aims to provide an execution environment that allows the safe and undisturbed execution of applications. An advantage of SFI is that other techniques to protect the execution of applications (e.g., Address Space Layout Randomization (ASLR) [35], [8], [9], Data Execution Prevention (DEP) [46], stack canaries [27], and policy-based system call authorization) are orthogonal to SFI and can be combined with SFI. The protection techniques can be used to strengthen the implementation of the SFI platform (that forms a crucial element of the trusted computing base) as well as the application code.

A key component of every system is the dynamic loader. The loader takes control of the application before the application is even started. The dynamic loader maps the application into memory and resolves all symbols that are used from different shared libraries. These relocations and symbol lookups enable a program to use libraries and to implement different techniques like position independent code. After the program has been prepared for execution (i.e., after the loader has finished the initial relocation and loading), the initialization code of the application is executed and the application starts.

The loader has access to all symbols and relocated objects at runtime and shares this information with the executing program. The standard Linux dynamic loader is optimized for fast relocation and aims to offer a rich functionality. The combination of the rich functionality (and complexity) and the fact that all applications (e.g., privileged applications, applications reachable over the network, local applications) use the same loader makes it a promising attack vector. Recent attacks [17], [34], [33], [40] illustrate the problem. Our approach replaces the standard dynamic loader by a tool that makes security a *first class concept*.

This paper introduces a Trusted Runtime Environment (TRuE) for the safe execution of untrusted code. TRuE is the combination of a secure loader and a sandbox. These two components of the trusted computing base are small and provide a a safe (the implementation is reviewed and bug-free) and secure (the design does not provide attack vectors in the offered functionality) environment. This combination enables a safe foundation for software-based fault isolation where all application code is executed under the control of the sandbox. The sandbox separates user-space into two privilege domains, the privileged sandbox domain that contains the secure loader and the sandbox, and the application domain that executes application code.

The standard loader focuses on feature support and low loading times, whereas the secure loader focuses on a rigorous security concept. The secure loader implements the safe foundation for the SFI framework, provides information for the sandbox that executes application code, and supports basic loader functionality. The secure loader collects information about all symbol locations and relocated pointers. This information is then used in the sandbox to secure the execution of the untrusted code.

This approach to a user-space sandbox implements a holistic view of the program execution. Inter-module control

DOI 10.1109/SP.2012.11

flow transfers are no longer implemented as jump tables but the loader forwards the information about the target location into the sandbox. The sandbox directly encodes the correct target location at the source control transfer instruction. This dynamic setup removes the additional indirections needed to cross module boundaries while still enabling dynamic loading of shared objects. The presentation in this paper focuses on a Unix-like context, but the ideas apply to other contexts as well.

We position TRuE as a replacement of the standard execution model for applications that run with a higher privilege level than the user that accesses the application. Two examples of such applications are (i) privileged "SUID" applications and (ii) applications that are connected to the network. A third possible use-case for TRuE is the execution of potentially malicious code in combination with specific system call policies. The interactions between the malicious binary (e.g., the program, a library, or a module) and the regular system libraries is controlled and checked using dynamic security guards.

The contributions of this paper are as follows:

1) the design of TRuE, a Trusted Runtime Environment for the creation of secured processes. TRuE builds on two principles, (i) a security-aware secure loader, and (ii) a sandbox that enforces the security policy of the execution model.
2) a report on a prototype implementation of TRuE. The prototype implementation demonstrates that the integration of the secure loader into an existing user-space sandbox is practical.

The rest of the paper is organized as follows. Section II defines the attack model and the execution model. Section III presents background information about the dynamic loading and sandboxing. Section IV describes problems of the standard approach. Section V defines the design and concepts of safe loading and secure execution in a trusted runtime environment. Section VI highlights the implementation of TRuE. Section VII presents the evaluation of the prototype implementation and discusses possible limitations. Section VIII presents related work and Section IX concludes.

II. Attack model and execution model

The Trusted Runtime Environment (TRuE) places the loader in the trusted computing base and runs all application code in a sandbox. The loader informs the sandbox about valid code regions of the application. The sandbox weaves additional checks (security guards) into translated application code. The sandbox does not implement full memory tracing to limit the overhead of SFI. The translated application code can read any readable memory location and can write any writable memory location without any additional penalties.

The attack model defines the constraints for an attack, the properties of a successful attack, and limitations of the trusted runtime environment. The execution model explains which applications can be protected and describes changes to the original memory layout of the application. The flow of execution in an application can be seen as a graph where control flow transfers are the edges and the nodes are individual basic blocks. The execution model describes how a sandbox transforms this graph at runtime to protect the execution flow and to ensure that no alternate "unwanted" locations are reached.

A. Attack model

A potential attacker tries to escalate the available privileges by executing injected or constructed code. A local attacker escalates the available privileges to a higher privileged account, e.g., by triggering an exploit in a "SUID" application to gain super-user access. A remote attacker without shell access, e.g., using a web service, gains user-level access by escalating the available privileges to a local user account, e.g., www-data.

The security guards of the sandbox protect the application from all code injection attacks. Code can be injected (as data) through, e.g., a buffer overflow, but the sandbox never executes the injected code. The sandbox either detects an illegal region that contains code when a "control flow instruction"[1] attempts to transfer control to a data region or the kernel generates a protection fault if the application tries to write a code region. A shadow stack [23], [38] in the sandbox domain ensures that return-oriented programming [44] attacks are not possible. The different privilege level of the sandbox domain naturally protects the shadow stack. The sandbox dynamically removes indirect control flow transfers whenever possible to reduce the opportunities for jump-oriented programming [10]. The sandbox uses per-application system call policies to protect from remaining attack vectors. Data-based attacks and jump-oriented programming are stopped whenever an illegal system call is executed. Attacks against the sandbox are limited by protecting internal data. All data structures of the sandbox domain (including the secure loader) are write-protected during the execution of translated code.

The sandbox kills the application if an attack is detected. Denial of service attacks (e.g., an attacker can repeatedly kill an application by trying to exploit a vulnerability) are outside of the scope of the attack model. Several mitigation techniques exist to restart failed services but they are not the topic of this paper.

B. Execution model

A binary-only application is executed under the control of the dynamic sandbox. The binary itself is untrusted but not malicious (i.e., the binary can contain implementation

[1]Any instruction that changes the control flow of the program, e.g., jump instructions, indirect jump instructions, call instructions, indirect call instructions, or return instructions.

bugs but the binary is not controlled by the attacker). No static modifications or static analysis are needed to execute an application in the sandbox. The sandbox implements an additional protection domain in user-space, splitting the user-space into an application domain and a sandbox domain.

TRuE creates a secure process in three steps: (i) TRuE initializes the secure loader and the sandbox during startup, (ii) the secure loader then loads the application in the sandbox and resolves all dependencies to external libraries, and (iii) the sandbox starts the main application thread in the application domain. The secure loader is a part of the sandbox domain. Accesses of the application into the loader are intercepted and redirected to the protected domain.

The sandbox dynamically translates all application code before it is executed. Original code regions in the application are mapped as read-only. The targets of static control flow instructions (direct jumps, direct calls, and conditional jumps) are verified during the translation. Dynamic control flow instructions incur a runtime check that verifies for each execution that the target is valid.

Self-modifying code in the application is not supported[2], i.e., the application is not allowed to generate new code at runtime. Code can only be added to the runtime image of an application through the secure loader API.

III. Background information

The startup process of an application is as follows: the loader opens the application, analyzes the dependences, loads and initializes the required libraries, and passes control to the application.

TRuE changes this approach. The secure loader first initializes the secure sandbox. The secure loader then opens and analyzes the application and all needed libraries. All application and library code is executed under the control of the sandbox, no unchecked code is executed directly.

A sandbox controls the executed application code. The sandbox combines software-based fault isolation (SFI) concepts and policy-based system call authorization to ensure that neither the application nor any exploits can escape the sandbox.

A. Dynamic loading

Linux uses the Executable and Linkable Format [41], [19] (ELF) to describe the on-disk layout of Dynamically Shared Objects (DSO) and executables. The ELF format defines two views for applications and libraries. The first view contains essential information for the loader about the different segments (areas with same page permissions) in the object. The second view contains the section header table with more fine-grained information like symbol tables.

[2]A sandbox that supports self-modifying code must move the modification engine (i.e., the JIT compiler) into the trusted sandbox domain to ensure that the generated code conforms to the execution model. This approach increases complexity but guarantees that only the trusted modification engine can generate dynamic code for the application.

Libraries contain one or two symbol tables: `.dynsym`, a dynamic symbol table that contains information (size, type, permissions, and others) about all exported symbols; and `.symtab` an optional table that contains information about all symbols in the library. The optional table is available by default and removed if the object is *stripped*.

Libraries are mapped to dynamic (non-constant) addresses in memory, the compiled code must therefore be position independent. Position independent code relies on the Global Offset Table (GOT), which contains information about imported and exported symbols for each DSO. This information is used to access symbols in other DSOs with non-constant addresses. The Procedure Linkage Table (PLT) is used to transfer control to symbols in other DSOs, entries in the PLT correspond to an indirect jump through a GOT slot, see Figure 1 for an example.

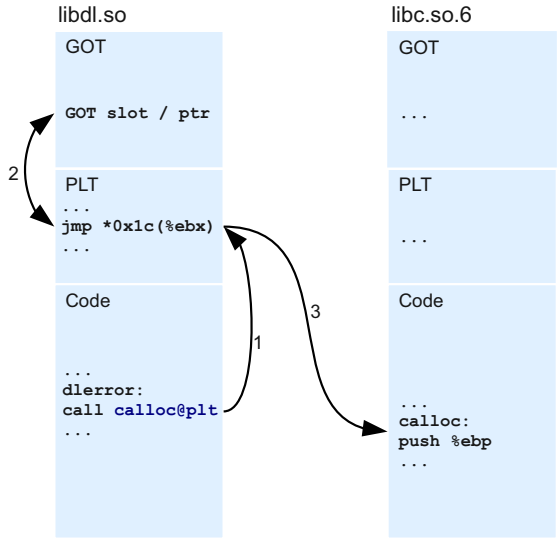

Figure 1. Example of PLT-based position independent code. The function in the code section transfers control to the PLT slot (1), the code in the PLT slot executes a lookup in the GOT section (2), and transfers control to the other shared object (3).

Function references in the GOT are initialized with a pointer to the dynamic loader. The first execution resolves the actual symbol and stores the resolved pointer in the GOT slot. Later calls to the same function result in a direct transfer to the resolved symbol. This feature allows lazy binding.

B. Sandboxing applications

Sandboxing or SFI is a technique that enforces a security policy on executed code. The security policy includes guards and restrictions on memory accesses, restrictions on control flow transfers, and restrictions on instructions and combinations of instructions that can be executed. Many dynamic SFI tools use binary translation to check the executed code.

SFI translates and encapsulates all executed instructions of the application.

Dynamic binary translation, the technique used to implement SFI, translates all executed code on-the-fly. Translated code is placed in a code cache to lower translation costs. The user-space dynamic binary translator (BT) takes control and implements a second privilege domain in user-space. Translated application code is executed with lower privileges than the BT. The BT ensures (during the translation) that the translated application code has no access to the data structures and to the code of the binary translator.

Figure 2 shows the design of a dynamic BT as presented in [37]. Each basic block is translated before execution and placed in the code cache. A mapping table maps basic blocks from the original program to translated basic blocks. Outgoing edges in translated basic blocks point to other translated basic blocks or back to the translator if the basic block is not translated. Untranslated basic blocks are translated on demand. Most instructions are copied verbatim for regular binary translation, all instructions that change control flow need special treatment.

The translator emits extra code that handles direct control flow transfers (e.g., jump instructions, conditional jump instructions, call instructions, interrupts, and system calls). Jump instructions and conditional jump instructions are redirected to the translated basic blocks in the code cache, interrupts and system calls are replaced with a code sequence that traps into the privileged domain of the BT. Indirect control flow transfers (e.g., indirect jumps, indirect calls, or function returns) are replaced with code sequences that execute a lookup in the mapping table and an indirect control flow transfer to the translated basic block. The special treatment of indirect control flow transfers is needed to keep up the illusion that the application is running in its native environment. No pointers and return addresses are changed by the BT, therefore the BT must trap the indirect control flow transfers to keep control of the program.

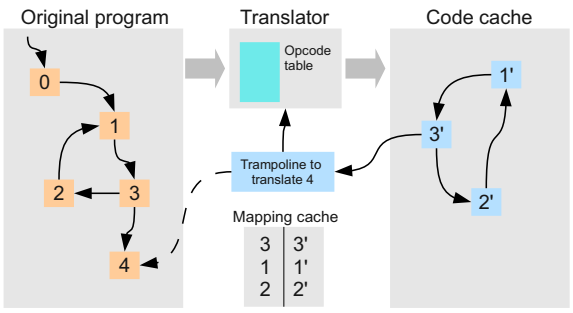

Figure 2. Overview of a dynamic binary translator.

BT enables the checking of machine code instructions before they are executed. SFI extends BT, the translation process (often using a code cache for already translated code) weaves additional guards (e.g., unexecutable stack, shadow stack, and write-protected executable code) into the translated code that ensure that the targets of control flow instructions cannot be changed by code based exploits.

System calls are requested using specific instructions and the sandbox is in the unique position to replace these instructions with special sandbox code. The sandbox checks individual system calls and verifies system call numbers, parameters, and locations according to a given system call policy. This concept is known as policy-based system call authorization.

IV. Problems of the standard loader

The standard loader has several problems if it is used in a security-relevant context. Bugs in the standard loader lead to direct privilege escalation. If the application and the sandbox share the same loader then the sandbox can be attacked through the loader. All dynamically loadable applications rely on features of the loader to dynamically resolve references or to load additional modules. If the loader is translated alongside the application then the application must have the privileges to map code as executable which pose a security risk.

A. Exploiting the standard loader

The standard execution model uses the same loader for all applications, no matter if they are regular user applications, privileged applications, or remotely accessible applications.

The standard loader supports a wide range of dynamic functionality (e.g., debugging, dynamic library replacement, and tracing of method calls) and a huge feature-set (e.g., many different relocation types). Large parts of the functionality are not needed or even harmful for privileged programs. Extra checks ensure that it is not possible to (i) preload alternate libraries or to (ii) replace the standard search path for libraries if an application uses the SUID flag.

Missing or faulty checks for privileged applications or other bugs in the standard loader [17], [34], [33], [40] can therefore be exploited to escalate privileges for SUID applications.

These problems can be mitigated or reduced if the functionality and feature-set of the loader is restricted to the bare minimum of necessary features to execute current applications.

B. The late interception problem

Many dynamic interception tools ([43], [45], [29], [11] use `LD_PRELOAD`[3] to gain control of the application. The application, the standard loader, and the binary translator (BT) share the same memory space. The data structures

[3] `LD_PRELOAD` is an option for the dynamic loader that injects an additional library into the process space of the application; this option executes the initialization code of the injected library prior to the initialization of the application.

of the loader contain pointers to the BT as well as to the application. The loader is in the application domain and the loader functionality can be used to gather information about the BT (i.e., resulting in an information leak) or to break the integrity of the BT. A potential exploit uses the data from the loader (e.g., modifies the GOT section of the BT) to compromise the BT itself and to redirect BT functions to malicious code.

The standard loader treats a BT that uses `LD_PRELOAD` just like any other shared object and enables the application to read information about the shared object (e.g., address space, PLT, and GOT sections). The standard loader does not guarantee that the BT initialization function is the first sequence of instructions that is executed after the loader finished its initialization. For example, the `INITFIRST` flag that can be set by multiple libraries. Preloaded libraries are loaded first but the standard loader executes the initialization code of the last loaded library with this flag first.

Another example is a symbol of the `GNU_IFUNC` relocation type. The standard libc uses `GNU_IFUNC` to select the best possible version of a function for the current hardware at runtime. Such a scenario may trigger the executing of setup code before the `LD_PRELOAD`-based BT is initialized. The BT is deprived of control of its environment if application code is executed before the BT is initialized.

C. The loader black box

In related work [43], [45], [29], [37], [11] the sandbox is either (i) unaware of the standard loader and translates the code of the standard loader as part of the application, or (ii) does not support dynamic loading [47], [31], [1], [20]. Solutions that are unaware of the loading process treat library loading as a black box.

The loader plays a privileged part during the runtime of all application that use shared libraries. The dynamic loader manages information about all loaded shared objects (libraries) and about all exported symbols that can be used in other objects.

The sandbox uses functionality of the loader to discover the loaded shared objects and the exported functions. The sandbox also relies on information about executable regions and data regions that is exported by the standard loader.

The loader is a crucial component of the application as it can load and map new code into the running process. If the loader is translated as a black box then the application must have the privileges to load and map any code (e.g., using the `mmap` system call, or the `mprotect` system call to map memory regions as executable). On the other hand if the sandbox provides a transparent and secure loader API then the privilege to map executable memory regions can be abstracted into the trusted sandbox domain. The sandbox can control the application and limit the loading process to predefined libraries.

An extension of the secure loader can be used to implement a clear separation between the different shared objects. Privileges and permissions (e.g., specific system calls and parameters to the system calls) can be tuned and specified on a per-object basis and are no longer enabled for all parts of an application.

V. Safe loading in a trusted runtime environment

The Trusted Runtime Environment presents an alternate model for process creation (turning an executable and all associated libraries into a running program) and is the first technique that takes complete control over an application in user-space. The standard dynamic loader is replaced by a secure loader that is part of the sandbox domain. As a result the application domain no longer needs the permissions to map executable code. A secure loader must be safe (the implementation is reviewed and bug-free) and secure (the design does not provide attack vectors in the offered functionality). This approach bridges programming languages and operating systems, a language independent loader is used to secure and confine binary-only applications in their execution pattern running on an operating system. Figure 3 provides a comparison between the standard runtime environment and TRuE.

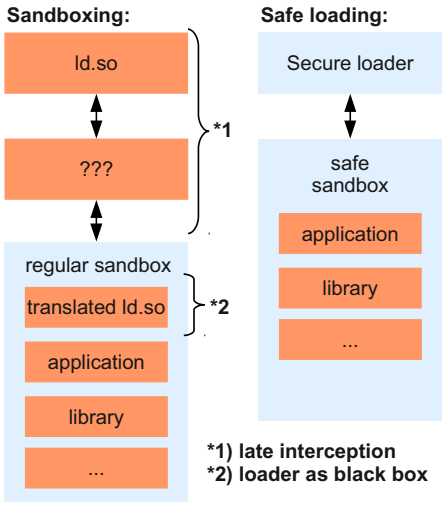

Figure 3. Comparison between a regular sandboxing approach and safe loading as provided by TRuE with a secure loader. The left-hand side shows two problems: 1) the late interception problem and 2) the loader black box problem.

The secure loader runs as part of the privileged sandbox domain. The secure loader is the only entity that is allowed to load new code and the application is only allowed to access loader functions through a well defined API. The sandbox and the loader are tightly coupled and share information about the program. The loader analyzes segment and section information of the application and all dynamically

loaded objects and enables per object privileges. The loader resolves objects, symbols, and relocations for the sandbox that then embeds resolved addresses in the translated code.

The tight coupling of the loader and the sandbox enables module separation. Control transfers between modules are inlined directly into the translated code. The translated source object contains a direct reference (that is unreadable from the application) to the target object and no call through the PLT and GOT is needed. The PLT data structure is only kept for reference reasons.

The secure loader solves problems of the standard loader that are discussed in Section IV. The secure loader ensures that the SFI library is initialized first and treated specially so that symbols are neither added to the global scope nor accessible through any API functions.

A. The sandbox

The sandbox implements two privilege domains in user-space: the sandbox domain (a trusted domain that contains the loader and the sandbox proper) and the application domain, an untrusted domain that contains the application code and all needed libraries.

The sandbox domain ensures that no unchecked code is executed in the application domain. Application code is examined by the sandbox before execution and additional security guards are added to ensure that the executed code cannot escape out of the sandbox.

Binary Translation (BT) is a key component for user-space software-based fault isolation (SFI). A dynamic translation system translates and checks every machine code instruction before it is executed. Translated code is placed in a code cache. Indirect control flow transfers trap into the privileged sandbox domain and are checked upon every execution, ensuring that 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.

An important requirement for the integrity of the sandbox is that return addresses of the translated application remain unchanged on the stack. Translated return addresses on the application stack would leak pointers into the code cache. Unchanged return addresses add additional complexity when handling return instructions as they are translated to a lookup in the mapping table (the mapping table is a sandbox-internal data structure that relates between translated and untranslated code) 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 has no easy way to discover that it runs in a sandboxed environment, and the address of the code cache is only known by the binary translator.

B. Solving the loader's security problems

Combining a secure loader and a safe sandbox to form a trusted execution environment solves the problems defined in Section IV. The loader must be separated from the application and the application may not access the loader's internal data structures directly. The privileged sandbox domain is a perfect location for the secure loader. The secure loader and the sandbox share information about all loaded shared objects and symbols. The shared information enables the sandbox to restrict control flow transfers in the application domain. The loader needs privileges to resolve dependencies and to map executable code, these privileges are best placed in the sandbox domain.

1) Restricting privilege escalation attacks: The secure loader implements a subset of the features of the standard loader. The subset is complete enough to run in practice any programs compiled with a recent version of the compiler toolchain. The supported programs are independent of the source language (C, C++, Fortran, and handwritten assembly programs were tested).

The feature set of the secure loader is limited to relocation types needed on the current platform (the standard loader supports relocation of other platforms as well), no runtime configuration, no debugging features that execute user-specified code, no backwards compatibility to old formats, and no direct access to loader internal data structures from the application are available.

TRuE targets programs running with a higher privilege level than the user interacting with the program. The secure loader does not read any environment variables and has no configuration files that are parsed at runtime. A user is not allowed to change settings for privileged programs. All settings are hardcoded during the compilation. Library paths, debugging features, and loader settings can only be changed before the compilation of the secure loader. The secure loader does not allow changes to any settings at runtime.

The removal of these user-settable features protects from attacks mentioned in Section IV-A. Privileged applications do not need these features, therefore removing the features altogether is more secure than executing additional checks before accessing the features (as done by the standard loader).

2) Protecting all executed application code: The initialization code of the secure loader is the first code that runs when an application is started. This initialization code starts and initializes the sandbox as well.

The secure loader can execute all application code under the control of the sandbox because the loader is part of the privileged sandbox domain. The secure loader tells the sandbox to translate an entry point to application code whenever the standard loader would pass control to application code. The application traps into the sandbox domain when it uses any loader functionality (e.g., resolving symbols, loading

additional modules, or loading PLT entries). The secure loader verifies the correctness of the request, and returns the result to the application domain. The secure loader cleans all references to internal data from any returned structures as an additional level of protection (next to setting internal data structures read-only when executing translated application code). The application uses the loader features through a well-defined API and can no longer read or write internal loader data. Trapping into the sandbox domain switches the stack of the current thread, stores information about the current state of the application thread (i.e., registers and PC), and adds write permission for the internal data structures of the secure loader; these changes are reversed when returning to translated application code.

This procedure ensures the safety of the secure loader, the sandbox, and the internal data structures at all times. Consequently the problems mentioned in Section IV-B do not exist for the secure loader.

3) Opening the loader black box: Placing the loader in the sandbox domain solves the loader black box problem from Section IV-C. The sandbox and the loader are in the same trust domain and together provide the base for trusted execution. Loader and sandbox can share data structures and exchange information about executable code regions, data regions, and symbol locations.

The loader is no longer translated by the sandbox as a part of the application but is an integral part of the sandbox. The application no longer needs privileges to map executable code into the application memory space but uses the loader API provided by the sandbox. All applications remain unchanged but calls to the loader are redirected to the secure loader API in the sandbox domain.

C. PLT inlining

The tight integration of the secure loader into the sandbox enables PLT inlining. The PLT is originally used to enable position independent code. The binary translator in the sandbox can remove the PLT code and inline the resolved target addresses directly into the generated code.

This optimization reduces the amount of indirect control flow transfers (these control flow transfers account for the main overhead in dynamic binary translation) and hides the location of other objects from the application.

The addresses are encoded directly in the code cache and the application has no access to the instructions in the code cache. This feature enables module separation and raises the bar for security exploits because a potential exploit is unable to determine the locations of specific functions in other objects. The total number of indirect control flow transfers is reduced, limiting jump-oriented attacks.

Format string attacks [36] and other data-oriented attacks can be used to circumvent guards in the application domain like ASLR, DEP, and stack canaries. These attacks overwrite pointers in GOT sections of shared objects and use the indirect jump instructions in the corresponding PLT regions to set up arbitrary code execution. Each PLT call in the application code can be used as a gadget for jump-oriented programming. PLT inlining closes this attack vector. Remaining indirect jump instructions in the application code (e.g., C switch statements are compiled to indirect jumps through jump tables) may still be used for jump-oriented programming attacks. The frequency of these remaining indirect jump instructions is low, thereby severely limiting the opportunities for jump-oriented programming.

D. Additional security features

A custom tailored exploit could target the binary translator itself. If the program is 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 break out of the isolation layer. Therefore any pointers to internal data are only allowed in the sandbox domain, and a protection guard in the sandbox must ensure that all data of the sandbox domain is write-protected whenever application code is executed.

The basic binary translator is extended by the following security guards that secure the user-space isolation sandbox and to ensure that application code cannot escape the sandbox: *Non-executable data and code* ensures that neither data nor code of the original application can be executed directly (by setting the non-executable bit on all memory pages of the application). This guard prevents code injection attacks. Only translated code in the code cache and code of the sandbox domain are executable. A *shadow stack* in the sandbox domain protects all return addresses on the application stack. This guard prevents return-oriented programming [44]. The sandbox uses *Address Space Layout Randomization* (ASLR) to allocate internal data structures at random addresses. ASLR is an additional probabilistic protection against bugs in the implementation of the sandbox. The sandbox uses a *dedicated stack* for all privileged code to prevent data leaks to the unprivileged domain. A trampoline switches the context (and stack) whenever privileged code is executed. A *protection guard* ensures that no data from the sandbox domain is writable when code from the application domain is executed. The protection guard uses information from the sandbox internal memory allocator and `mprotect` system calls to write-protect all sandbox internal data structures whenever translated application code is executed.

VI. Implementation

The Trusted Runtime Environment (TRuE) is implemented as a combination of a secure loader and an extended version of the libdetox [37] sandbox. TRuE integrates the information from the loader into the security guards. The secure loader initializes the sandbox before any application

or library code is loaded or executed. All application and library code is then executed in the sandbox[4].

The secure loader uses ELF information and symbol table information [41] and implements all needed functionality to load most programs (e.g., OpenOffice, and the SPEC CPU benchmarks).

The SFI platform is tightly coupled with the secure loader. The loader first maps libdetox into the address space and initializes the SFI platform. This special treatment ensures that the SFI platform is initialized and that the application has no access to or knowledge of the sandbox domain. The next steps are the relocation of the application and all needed shared objects. The loader controls all data that is passed to the application and runs all user code under the control of the SFI platform.

The prototype implementation of TRuE is small. According to `ohcount`[5] the secure loader consists of around 5,400 lines of code (including 2,100 lines of comments) and the sandbox platform consists of around 20,200 lines of code (including 5,000 lines of comments and 4,900 lines for the full IA32 translation tables[6]).

A. Application and library loading

The secure loader implements the most common subset of features from the standard loader. Some features (e.g., overwriting library search paths, runtime debugging, or debugging features that execute user-supplied code for specific events) are removed and not implemented out of security concerns. Unimplemented features result in an error message and graceful termination of the program. The current implementation prototype covers the core functionality needed to execute in practice any ELF programs of Ubuntu 11.04 that originally use the standard loader (i.e., any ELF program that uses shared libraries). Further options (e.g., obscure relocation patterns, additional callbacks from the application into the loader, and access to internal loader data[7]) can be added if needed.

The standard loader has no protection for internal data structures and leaks pointers to the internal data structures to the application. The API of the secure loader that is accessible from the application (e.g., `dlopen`, `dladdr`, and `dlsym`) ensures that no protected internal data is leaked to the application. The sandbox write-protects all internal data whenever (translated) application code is executed by using `mprotect` on all memory regions of the sandbox.

[4]The source code of the prototype implementation of TRuE is available as open-source at http://nebelwelt.net/projects/TRuE.

[5]`ohcount` is a tool to measure different code metrics like lines of code.

[6]The IA32 translation tables contain detailed information about all IA32 instructions. The translation tables enable the BT to decode and to translate individual IA32 instructions.

[7]GDB uses undocumented direct access to the internal data from the loader to get more information about individual symbols. This feature can be implemented as a proxy that projects information out of the secure loader if needed.

The secure loader must handle the startup of new applications. First of all the loader is completely independent from any libraries (even the standard libc) and is just mapped into memory. This loader then examines the ELF headers of the application and maps the runtime sections of the application to a fixed address in memory. Then the list of needed libraries is examined and entries are added to a "to-process-list". The loader dequeues one entry at a time and loads and initializes this library at random addresses. If the library depends on other libraries then they are added at the end of the "to-process-list". This algorithm conforms to a breadth-first traversal of the dependence graph of the application starting with the application as the root node.

References to needed libraries only contain the name of the library but not the path. When the loader locates a new library several paths are examined: first a per-DSO variable that specifies one or more search paths per DSO, then the standard search paths defined in `/etc/ld.so.conf`. The standard libc loader also supports additional search directories using the `LD_LIBRARY_PATH` environment variable and the local cache file `/etc/ld.so.cache`. Out of security reasons the secure loader does not support runtime-configurable paths.

B. Symbol resolving

The loader resolves symbols using the symbol tables in the different shared objects. Every shared object contains the `.dynsym` table with all exported symbols. If the loader needs to resolve an imported symbol then the loader checks different lookup scopes. The loader defines three different lookup scopes that are checked one after the other:

1) *Loader scope:* this scope contains the symbols that are exported by the secure loader. The loader scope is checked first and symbols in this scope cannot be overwritten.
2) *Local scope:* the local scope of a DSO contains its own symbols and the symbols of all libraries that the DSO depends on. This scope is a subset of the global scope.
3) *Global scope:* shared objects that are in the initial set of objects loaded during the startup of the application (e.g., all objects in the dependence graph) or shared objects that are loaded at runtime with the `RTLD_GLOBAL` flag set are in the global scope.

A special feature is symbol versioning where symbols can be defined multiple times with different versions. The correct symbol is then selected based on a matching version.

The secure loader supports the GNU IFUNC relocation format (`STT_GNU_IFUNC`) where a piece of code is executed to determine the correct location of the symbol. This feature is, e.g., used in the libc to select between multiple implementations of a function. The test function checks if a specific CPU feature is available and returns the most optimized version for the current environment. The

Nr	Name	Type	Size	Flags
0		NULL	0	
1	.note.gnu.build-i	NOTE	24	R
2	.note.ABI-tag	NOTE	0x20	R
3	.gnu.hash	GNU_HASH	0x3c38	R
4	.dynsym	DYNSYM	9200	R
5	.dynstr	STRTAB	005acd	R
6	.gnu.version	VERSYM	0x1240	R
7	.gnu.version_d	VERDEF	0x3d8	R
8	.gnu.version_r	VERNEED	0x40	R
9	.rel.dyn	REL	0x2a20	R
10	.rel.plt	REL	0x40	R
11	.plt	PROGBITS	0x90	RX
12	.text	PROGBITS	0x1088d4	RX
13	__libc_freeres_fn	PROGBITS	0xfc8	RX
14	__libc_thread_fre	PROGBITS	0x182	RX
15	.rodata	PROGBITS	0x1b808	R
16	.interp	PROGBITS	0x13be68	R
17	.eh_frame_hdr	PROGBITS	0x333c	R
18	.eh_frame	PROGBITS	0x132b4	R
19	.gcc_except_table	PROGBITS	0x5c1	R
20	.hash	HASH	0x3484	R
21	.tdata	PROGBITS	0x8	RWT
22	.tbss	NOBITS	0x38	RWT
23	.fini_array	FINI_ARRAY	0x4	RW
24	.ctors	PROGBITS	0x14	RW
25	.dtors	PROGBITS	0x8	R
26	__libc_subfreeres	PROGBITS	0x70	RW
27	__libc_atexit	PROGBITS	0x4	RW
28	__libc_thread_sub	PROGBITS	0xc	RW
29	.data.rel.ro	PROGBITS	0x1afc	RW
30	.dynamic	DYNAMIC	0xf0	RW
31	.got	PROGBITS	0x174	RW
32	.got.plt	PROGBITS	0x2c	RW
33	.data	PROGBITS	0x97c	RW
34	.bss	NOBITS	0x3068	RW
	...			
68	not allocated			

Table I
THESE SECTIONS OF THE STANDARD LIBC ARE MAPPED AT RUNTIME USING THE GIVEN FLAGS (X - EXECUTE, W - WRITABLE, R - READABLE, T - THREAD LOCAL STORAGE). COMMAND USED TO GET THIS INFORMATION: `READELF -S /LIB32/LIBC-2.13.SO`.

loader then uses this function pointer and forwards it to the requesting DSO where the function pointer can be embedded in the GOT.

C. Memory protection

One of the advantages of a secure loader is that all loader-related data structures can be write-protected. The secure loader manages two kinds of data structures, internal data structures and application data structures.

Internal data structures contain information about the different relations between shared objects, scope information, and other details about the loaded objects. This information is updated by the secure loader whenever new shared objects (e.g., additional shared libraries) are loaded and initialized. The secure loader maps these data structures read-only whenever application code is executed.

Shared objects contain data structures that are only changed by the loader and are only read by the application. If we take the standard libc 2.13 as an example we see in Table I that there are 34 ELF sections that are mapped to memory. 11 sections are mapped writable (.tdata, .tbss, .fini_array, .ctors, __libc_subfreeres, __libc_atexit, __libc_thread_sub, .data.rel.ro, .dynamic, .got, .got.plt, .data, .bss) and 1 section (.dtors) is marked read-only but on the same memory page as .ctors and is therefore writable as well. Most of these sections are used only during the initialization of the shared object. The sections .data.rel.ro, .dynamic, .got, .got.plt are critical for the loader and can be used in attacks against a classic sandbox that does not integrate the loader into the security concept. The standard loader maps .data.rel.ro as read-only after the initialization but the other sections remain writable. Out of the writable set of sections only .data and .bss are used by the libc code.

The secure loader write-protects all sections except .data and .bss dynamically to protect the application from modification attacks in these sections whenever translated application code is executed. If the secure loader needs to update write-protected structures (e.g., a GOT entry) then the write-permission is set temporarily during the update in the sandbox domain. The write-permission is removed when returning to the application domain.

D. Loader optimizations

The secure loader currently implements two optimizations, lazy binding and PLT inlining.

Lazy binding reduces the amount of relocations that have to be calculated when a library is loaded. Only symbols in the data region are relocated but symbols in the PLT region are only resolved and relocated when the function is executed the first time. This optimization is also implemented in the standard loader.

The implementation of PLT inlining follows the design in Section V-C and uses the close relationship between the secure loader and the sandbox. The sandbox intercepts all call instructions and checks for each instruction if the call is a PLT call. The secure loader then resolves the static target address of the PLT target. The original call and indirect jump of the PLT call are then replaced by a translated call instruction to the resolved target. This removes an indirect jump including an indirect control flow check for every PLT call that is executed.

During the loading process weak symbols of prior DSOs can be overwritten by symbols in the current DSO. Library loading forces all threads to trap into the sandbox domain and to flush their code caches. If the weak symbol points to a function and this function was inlined (through a PLT slot) then the sandbox has an invalid reference in the code cache and must therefore flush the code cache to retranslate the given PLT slot.

E. Handling of the sandbox

The secure loader handles the sandbox in a special way. The loader resolves the additional sandbox code before any

shared library or application code is loaded and initialized. The symbols of the TRuE framework are also resolved in a protected scope that is only accessible by the secure loader and the sandbox.

Any application or library code that is then executed during the initialization phase is executed under the control of the sandbox, enabling security right from the start.

F. Changes to the regular sandbox

The safe sandbox in Figure 3 is based on the libdetox [37] open-source project. Changes to the original implementation are an API for the secure loader, an alternate sandbox stack for functions in the sandbox domain, and a new shadow stack to store information about the application stack in the sandbox domain.

The sandbox uses the secure loader to lookup information on the different sections. The sandbox uses this information to decide if code is in an executable section of a shared object or in some other region. The same information is used to implement PLT inlining as described in Section VI-D.

Our sandbox uses specific entry and exit trampolines to simplify the transition between the application domain and the sandbox domain. The entry trampoline handles the transition from translated application code to privileged sandbox code. The application stack remains unchanged, registers are spilled to a thread-local storage area in the sandbox domain and the stack is swapped to a sandbox stack. Code running in the sandbox domain uses the sandbox stack to store local information. The exit trampoline returns from the sandbox domain to the application domain. The trampoline restores registers, switches back to the application stack, and continues the execution of the translated code.

Events that trigger a switch from the application domain to the sandbox domain are:

Lookup misses in the mapping table: if an indirect control flow transfer cannot be resolved with the inlined assembler code (e.g., a quick lookup in the first entry of the mapping hash table) then the control flow transfer code escalates to the sandbox domain and requests a slow-path lookup.

Untranslated code: if the translated application code branches to untranslated code an exception is triggered and the sandbox either translates the untranslated code and continues execution or faults.

Signals and exceptions: the sandbox installs special handlers to catch all signals and exceptions. These handlers check the signal or exception, resolve the original instruction pointer[8], check if the signal or exception is legit, and pass the information to the application.

System calls: system calls trigger a switch to the sandbox domain. A handler copies the arguments of the system call into the sandbox domain. If an argument is a pointer to a data structure then only the pointer is copied. The handler then checks the combination of system call and parameters using a per-application policy. The system call is evaluated in the sandbox domain to protect from *time of check to time of use attacks* by concurrent threads.

[8]The kernel passes an instruction pointer to the code cache that must be resolved to a pointer in the application domain before the signal or exception is passed to the application.

The shadow stack protects the application from return-oriented programming attacks. The basic concept of the shadow stack keeps information about the application stack frames in the sandbox domain. The shadow stack uses triples of pointers of return instruction pointer, translated return instruction pointer, and stored application stack pointer. The original libdetox implementation uses only pairs of pointers of return instruction pointer and translated return instruction pointer. The advantage of using triples is that the stack can be resynchronized if there is a mismatch. If the last translated function removed multiple stack frames (e.g., through exception handling) then the reauthentication method can pop stack frames on the shadow stack until the application stack pointer matches the stored stack pointer on the shadow stack, resynchronizing the shadow stack with the application stack.

G. Implementation alternatives

We discuss two implementation alternatives that offer a similar security concept to the combination of a secure loader with a sandbox. The first alternative uses static recompilation. All libraries are compiled to a statically linked binary, guards are added during the recompilation, and the loader is no longer needed. This approach has several drawbacks: (i) there is no second protection domain; exploits can get control of the user-space and then execute arbitrary system calls, (ii) static recompilation is limited to statically known targets and code locations (e.g., handling of dynamic jump-tables for switch statements), (iii) a secure static runtime environment must restrict the ISA and the dynamic control flow transfer instructions to limit the dynamic options of the IA32 ISA.

A second alternative implements a sandbox without changing the loader. The sandbox is hidden from the application using loader tricks that alter the data structures of the loader, or the sandbox is added as a binary blob and injected into the process image by an external process. This implementation approach has the disadvantage that it is hard to hide the sandbox from the loader/application and to remove all traces from the sandbox in the loader data structures. A second disadvantage is that loader code is translated as well, especially when new symbols are resolved. This disadvantage leaves the loader black box problem unsolved.

VII. Evaluation and discussion

This section evaluates and discusses the implementation prototype of the secure execution platform to demonstrate its practicability. The evaluation shows a performance evaluation for the SPEC CPU benchmarks and discusses limitations of the current implementation.

A. SPEC CPU benchmarks

We use the SPEC CPU2006 benchmarks version 1.0.1 to evaluate the performance and feasibility of our prototype implementation. The SPEC benchmarks are run on an Intel Xeon E5520 CPU at 2.27GHz on Ubuntu Jaunty with gcc version 4.3.3 on a x64 kernel with 32bit support. We evaluate a subset of the SPEC CPU 2006 benchmarks. The missing C++ benchmarks did not compile with the gcc 4.3.3 due to changes of the C++ header files; the missing fortran benchmarks did not link due to library problems under 64bit. These missing SPEC CPU 2006 benchmarks run if the source files (for C++ based benchmarks) or the Makefile (for fortran based benchmarks) are patched. This section only reports on the unmodified SPEC CPU2006 benchmarks.

Table II displays the number of relocations per benchmark run and the number of loaded DSOs for a subset of the SPEC CPU2006 benchmarks. The total number of relocations is low (between 1,381 and 1,597 relocations) for all the evaluated SPEC CPU 2006 benchmarks.

Table III shows the overhead of the secure loader compared to the standard loader. The performance of the secure loader is competitive to the standard loader. Comparing the columns of the secure loader to the secure loader with memory protection illustrates that the overhead of the secure loader to protect all writable sections except `.data` and `.bss` is negligible. The cost for protecting the memory pages that contain the loader data for each shared object is amortized during the runtime of the program.

Benchmark	Relocations	DSOs	Runs
400.perlbench	1,447	3	3
401.bzip2	1,368	2	6
403.gcc	1,437	3	9
429.mcf	1,381	3	1
445.gobmk	1,422	3	5
456.hmmer	1,431	3	2
464.h264ref	1,423	3	3
435.gromacs	1,597	5	1
470.lbm	1,377	3	1

Table II
Per benchmark average number of relocations, loaded DSOs, and number of binaries executed in a benchmark run for a subset of the SPEC benchmarks.

The last column displays the overhead of TRuE (including secure loader, memory protection from Section VI-C and full sandboxing of all application code). Most programs have low overhead and safe execution is feasible. Running all application code in a sandbox and checking all control flow transfers results in additional overhead between 0.5% and 85% for the SPEC benchmarks compared to the standard loader. The overhead results mostly from binary translation (i.e., the execution of indirect control flow transfers) and only little overhead is induced through the additional security checks.

Benchmark	SL	SL+mprot	TRuE
400.perlbench	-0.3%	-0.2%	85%
401.bzip2	-0.1%	-0.1%	4.9%
403.gcc	-0.9%	-0.9%	38%
429.mcf	-0.1%	-0.1%	0.5%
445.gobmk	0.0%	0.0%	32%
456.hmmer	0.0%	0.0%	5.3%
458.sjeng	0.0%	0.0%	58%
464.h264ref	-0.3%	-0.3%	41%
473.astar	0.1%	0.0%	8.3%
433.milc	-0.1%	0.0%	3.7%
434.zeusmp	0.0%	0.3%	-0.5%
445.gromacs	0.0%	0.0%	0.8%
436.cactusADM	0.2%	0.8%	0.6%
444.namd	0.0%	0.0%	1.1%
450.soplex	-0.2%	-0.2%	8.4%
459.GemsFDTD	-0.2%	-0.2%	3.0%
470.lbm	0.0%	0.1%	0.2%
482.sphinx3	0.1%	0.1%	0.5%
462.libquantum	0.0%	0.0%	2.2%
Average	**-0.1%**	**0.0%**	**15%**

Table III
Per benchmark average overhead compared to the standard loader. The columns are the secure loader, secure loader plus memory protection (Section VI-C), and TRuE: secure loader, memory protection and full sandboxing of all code.

Benchmarks with a very high number of indirect control flow transfers (these transfers incur a runtime check in the sandbox) have higher overhead (e.g., 400.perlbench, 403.gcc, or 464.h264ref). Every executed indirect control flow transfer needs a runtime lookup in the mapping table of the BT. The BT implementation reduces the cost of runtime through caching of source target pairs, fast paths for often executed targets, and other optimizations. Nevertheless they are the biggest factor in the overall overhead.

The average overhead for all evaluated benchmarks is 15% which is tolerable for the combination of safe loading and sandboxing. The overhead of TRuE for individual benchmarks is comparable to the sandboxing overhead of libdetox [37].

B. OpenOffice 3.2.1

We measured OpenOffice startup as a *stress test* and *worst-performance metric*, 145 DSOs are loaded, relocated, and executed with very low code reuse. OpenOffice was run on an Intel Core i7 CPU at 3.07GHz on Ubuntu Maverick.

OpenOffice 3.2.1 executes 265,067 relocations during the startup phase and loads 145 individual shared objects. The secure loader imposes an overhead of 44% for OpenOffice and 77% overhead for the additional memory protection. If

the full protection sandbox and the secure loader are used in combination then the start-up of OpenOffice is slowed down by 188%.

The overhead for OpenOffice results from additional checks that are carried out whenever a new shared object is loaded and all relocation entries need to be resolved. The OpenOffice startup sequence is evaluated as a worst-case scenario. Code is rarely reused and a huge number of references between objects need to be resolved. The overhead for the secure loader comes from less efficient loading and symbol resolving. The additional overhead between the secure loader and the secure loader plus memory protection comes from the additional `mprotect` system calls used to protect all runtime sections except `.data`, `.bss`, `.tdata`, and `.tbss`.

	standard loader	secure loader	loader ovhd.
native	178,336 kB	208,312 kB	(16.8%)
sandbox	256,156 kB	289,569 kB	(13.0%)
sandbox ovhd.	(43.6%)	(39.0%)	

Table IV
OPENOFFICE MEMORY CONSUMPTION SHOWING THE MEMORY OVERHEAD OF THE SECURE LOADER AND THE SANDBOX.

Table IV shows OpenOffice memory consumption as given by `ps -o vsz,command`. The secure loader consumes between 13.0% and 16.8% more memory than the standard loader. The standard loader uses `malloc` and `free` to allocate memory. The secure loader does not use any external libraries and relies on direct `mmap` calls and a less-efficient internal memory management system. The secure loader memory overhead can be reduced with a more efficient memory management system.

Sandboxing results in 39.0% to 43.6% memory overhead due to the internal data structures of the sandbox, the mapping cache, and the code cache for translated application code. An overhead of 77,820 kB to 81,257 kB to sandbox large applications like OpenOffice is both tolerable and feasible.

C. Discussion of TRuE's security features

TRuE combines a sandbox that enables the execution of untrusted code with a secure loader. The secure loader can load and relocate unmodified binaries and shared libraries that are then executed under the control of the sandbox. The sandbox uses the internal information of the secure loader to optimize the code layout of the internal cache. TRuE protects unmodified binary applications from code-based attack vectors and enables a safe foundation to execute applications that use shared libraries.

TRuE splits the user-space into two execution domains, the privileged sandbox domain that controls an application and the application domain that executes translated application code. The secure loader starts and initializes the sandbox before the application binary is opened. The loader then loads and relocates the application and all libraries. Any application code is executed under the control of the sandbox.

The sandbox ensures that no untranslated code is executed. The memory layout of the sandbox ensures that no code-injection attacks are possible. All memory regions are either executable or writable, but never executable and writable. The security guards that are woven into the translated code ensure that any direct or indirect control flow transfers only redirect control flow to already known and verified targets.

The secure loader only implements bare-bones functionality needed to load and relocate applications on a single platform. No inter-platform operability, no debugging features, no runtime-configurable settings, and no runtime-changeable settings are implemented. This bare bone paradigm drastically reduces the total number of lines of code needed to implement the loader functionality.

The standard loader executes extra checks that disable some features for privileged applications. These checks can contain bugs [17], [33], [34] that enable an arbitrary user to execute code as privileged user. These features are not available in the secure loader and cannot be exploited.

The combination of a secure loader with a sandbox offers several advantages. The secure loader enables a clean foundation to implementing a secure sandbox, unmodified binary applications are safely executed in the unprivileged application domain. Any requests for system calls, indirect control flow transfers, or functionality of the dynamic loader trap into the sandbox domain. The application domain has no privileges to map executable code. The sandbox ensures that no untrusted application code is executed outside of the sandbox. Any calls into the loader trap into handler functions in the privileged sandbox domain where the parameters can be checked and verified.

D. Limitations of the current implementation

TRuE protects from all code injection based attacks (on the stack and on the heap). Regular code sections of the application are mapped read-only and only translated application code in the code cache is executable. Other memory pages of the application are never mapped executable.

The shadow stack protects the return instruction pointer using a privileged shadow stack in the sandbox domain. This guard protects from all stack-oriented attacks (return to libc attacks and return-oriented programming [44]).

A limitation of the current approach is that jump-oriented programming attacks [10] and data-only attacks (application data is over-written using a malicious write to a memory page) are still possible. Jump-oriented attacks and data-only attacks can redirect the control flow to alternate locations in the code but the attacks can never introduce new code or break out of the sandbox. Only translated code is executable

and all outgoing edges at the end of a basic block in the code cache are either patched to other translated basic blocks or trigger a fallback into the sandbox to translate previously untranslated code.

Similar to libdetox [37] we can use a system call policy to ensure that the application code cannot break out of the sandbox and to protect from jump-oriented attacks and data attacks at the more coarse-grained system-call level. An advantage of moving the loader into the sandbox domain is that we do not need to consider the system calls needed by the loader in our application policy. The policy can be reduced to the functionality actually needed by the application and is not polluted by system calls that are needed for loader functionality.

A second limitation that is shared with libdetox is the inability to securely support self-modifying code (i.e., JIT compilers). A JIT compiler can generate arbitrary code. TRuE uses a privileged sandbox domain to handle code generation and module loading. If an application contains a JIT compiler then it is placed in the untrusted application domain. The application domain is not allowed to generate new code. If the application domain was allowed to generate new code then a JIT compiler would not be distinguishable from a code injection attack. A possible solution for applications that need a JIT compiler is to either promote the JIT compiler to the sandbox domain or to define a secure API that is used by the JIT compiler in the application domain to notify the sandbox domain of newly generated correct code. This extension is a topic for future work.

VIII. Related work

This section presents information about related work. Many different sandboxing techniques already exist. Most dynamic techniques use either an `LD_PRELOAD` based approach or rely on trusted application code to initialize the sandbox. Policy-based system call authorization checks all system calls and system call parameters of an application. Policy-based system call authorization can be used as an extension of sandboxing or by itself.

Sandboxing uses binary translation to encapsulate running code [11], [30], [32], [45]. Libdetox [37], Vx32 [22], Strata [43], [42], and program shepherding [29] implement software-based fault isolation using binary translation. Additional guards like non-executable memory regions, stack protection, and system call policies can be added during the dynamic translation of the machine code.

The basic SFI framework must be fast, extensible, and secure. Many different instrumentation frameworks exist and one must be aware of the limitations that several optimizations pose to security.

Policy-based system call authorization stops the application when system calls are executed. The arguments and the location of the system call are then matched against a given policy. The program is terminated if a policy violation is detected. Different techniques can be used to implement system call authorization, e.g., `ptrace`-support [26], trusted code in the kernel [6], [39], [21], [24], or binary translation [29], [37].

Apart from user-space isolation there exist other possibilities to secure a running systems. Dynamic systems add additional guards and checks to a running application. These systems all work at different levels of granularity. Full system virtualization [7], [18], [12], [5] encapsulates a complete running system and works at a very coarse-grained level of granularity [25], [28], [13]; system call interposition

Product/Feature	Techniques used[a]	System call interposition	System call policies	Full ISA supported	Completely transparent translation	Stack exploit protection (ret2libc)	Control flow integrity	No special kernel-module needed	No application changes needed	Separate secure stack for monitor[b]	Separate shadow stack for application	TOCTTOU aware using safe-guards	Source code available
TRuE	1	x	x	x	x	x	(x)	x	x	x	x	x	x
libdetox [37]	1	x	x	x	(x)	(x)	(x)	x	x			(x)	x
Vx32 [22]	1	x		[c]				x					x
Strata [43], [42]	1	x		?	?	?		x	x	?			
Prog. sheph. [29]	1	x	(x)[d]	x	x	(x)[e]	(x)	x	x	x			
Janus [26]	3	x	x		x			x	x				x
AppArmor [6][f]	3	x	x		x				x				x
SysTrace [39]	3	x	x		x				x				x
Switchblade [21]	3	x	x										
Ostia [24]	3	x	x										
NaCl [47]	2					x		x					x
PittSField [31]	2					x		x					x
CFI/XFI [1], [20]	2			[g]		x	x	x			x		
StackGuard [16]	4					x		x	(x)				x
libverify [4]	4					x		x	(x)				x
Propolice [27]	4					x		x	(x)				x
PointGuard [14]	4							x	(x)				(x)

The different features describe limitations and possibilities of each approach. x includes an available feature, (x) marks a limited feature, a blank marks a missing feature, a ? indicates that no information about this item is available.

[a] 1: dynamic BT; 2: static BT; 3: kernel module or kernel support; 4: compiler extension

[b] Monitor has a separate stack (e.g., permission check or code translation).

[c] Implements IA32 subset: no FPU, MMX, SSE, and 3 byte opcodes.

[d] Static hard-coded policy, only open and execve calls are intercepted.

[e] `ret` must target instructions immediately after any `call` instruction.

[f] MAPbox [2], SubDomain [15], and Consh [3] use a comparable approach.

[g] According to the paper at least no FPU, MMX, and SSE.

Table V
Summary of related work.

encapsulates the application at the system call level and works at the granularity level of individual applications and their system calls.

Static protection reduces the potential overhead but either restricts the instruction set or introduces complicated static analysis. Static verification allows only a sub-set of the instruction set or imposes other additional checks. Compiler extensions can be used as a quick fix to patch a specific static problem.

Table V presents a concise summary of related work and distinguishes features, design and implementation details of these different approaches.

IX. Concluding remarks

This paper presents a Trusted Runtime Environment (TRuE) consisting of a secure loader and a user-space sandbox. The secure loader enables *safe loading* that is a foundation for safe software-based fault isolation. TRuE replaces the standard loader with a security-hardened bare-bones implementation and uses user-space process sandboxing to execute application code under the control of dynamic security guards.

Bugs in the standard loader are often used to escalate privileges. The secure loader is restricted to the basic functionality. The restricted functionality protects from many exploits against the standard loader. Safe loading ensures that SFI is seamlessly integrated into the loader and guarantees that no unchecked code is executed. The trusted, secure loader enables additional security guards in the sandbox. The sandbox is aware of all loaded code regions and the connections between the different shared objects (i.e., the application, or libraries). This information is used to restrict applications to a secure execution model. The loader is no longer treated as a black box but integrated into the security concept. The secure loader and the sandbox run in the same protection domain and share information about the application. Calls from the application to the loader are redirected into the sandbox domain where the requests are verified. Applications running in the sandbox need fewer privileges, and code-oriented attacks are no longer possible. An additional advantage of the shared information between the loader and the sandbox is the potential to remove many indirect control flow transfers between modules. This optimization reduces the overhead of the sandbox and limits jump-oriented programming attacks. This approach bridges the security context of programming languages and operating systems by enabling a language-independent secure execution of applications.

TRuE enables a secure way to create and control applications in user-space with low overhead. Privileged applications and applications that are reachable over the network should be hardened and protected from security exploits: safe loading provides a foundation to solve this problem.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *CCS'05: Proc. 12th Conf. Computer and Communications Security* (2005), pp. 340–353.

[2] Acharya, A., and Raje, M. MAPbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proc. 9th Conf. USENIX Security Symp.* (2000), pp. 1–17.

[3] 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).

[4] Baratloo, A., Singh, N., and Tsai, T. Transparent runtime defense against stack smashing attacks. In *Proc. USENIX ATC* (2000), pp. 251–262.

[5] 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* (2003), pp. 164–177.

[6] Bauer, M. Paranoid penguin: an introduction to Novell AppArmor. *Linux J. 2006*, 148 (2006), 13.

[7] Bellard, F. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC* (2005), pp. 41–41.

[8] Bhatkar, E., Duvarney, D. C., and Sekar, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003), pp. 105–120.

[9] Bhatkar, S., Bhatkar, E., Sekar, R., and Duvarney, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: Proc. 14th USENIX Security Symp.* (2005), pp. 255–270.

[10] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security* (2011), pp. 30–40.

[11] Bruening, D., Garnett, T., and Amarasinghe, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (2003), pp. 265–275.

[12] Bugnion, E. Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache. US Patent 6704925, March 2004.

[13] Chow, J., Garfinkel, T., and Chen, P. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC* (2008), pp. 1–14.

[14] Cowan, C., Beattie, S., Johansen, J., and Wagle, P. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003).

[15] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., and Gligor, V. SubDomain: Parsimonious server security. In *Proc. 14th USENIX Conf. System Administration* (2000), pp. 355–368.

[16] 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: Proc. 7th USENIX Security Symp.* (1998).

[17] DANEN, V. CVE-2011-1658: ld.so ORIGIN expansion combined with RPATH. *https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2011-1658*.

[18] 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.

[19] DREPPER, U. How to write shared libraries. *http://www.akkadia.org/drepper/dsohowto.pdf* (Dec. 2010).

[20] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *OSDI'06* (2006), pp. 75–88.

[21] FETZER, C., AND SUESSKRAUT, M. Switchblade: enforcing dynamic personalized system call models. In *EuroSys'08: Proc. 3rd Europ. Conf. Computer Systems* (2008), pp. 273–286.

[22] FORD, B., AND COX, R. Vx32: lightweight user-level sandboxing on the x86. In *Proc. USENIX ATC* (2008), pp. 293–306.

[23] FRANTZEN, M., AND SHUEY, M. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proc 10th USENIX Security Symp.* (2001).

[24] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *NDSS'04: Proc. Network and Distributed Systems Security Symp.* (2004).

[25] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS'03: Proc. Network and Distributed Systems Security Symp.* (2003).

[26] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications: Confining the wily hacker. In *SSYM'06: Proc. 6th USENIX Security Symp.* (1996).

[27] HIROAKI, E., AND KUNIKAZU, Y. propolice : Improved stack-smashing attack detection. *IPSJ SIG Notes*, 75 (2001), 181–188.

[28] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys'06: Proc. 1st Europ. Conf. Comp. Sys.* (2006), pp. 29–41.

[29] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *SSYM'02: Proc. 11th USENIX Security Symp.* (2002), pp. 191–206.

[30] 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* (2005), pp. 190–200.

[31] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *SSYM'06: Proc. 15th USENIX Security Symp.* (2006), pp. 209–224.

[32] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (2007), pp. 89–100.

[33] ORMANDY, T. CVE-2010-3847: GNU C library dynamic linker $ORIGIN expansion vulnerability. *http://www.exploit-db.com/exploits/15274/*.

[34] ORMANDY, T. CVE-2010-3856: GNU C library dynamic linker LD_AUDIT arbitrary DSO load vulnerability. *http://www.exploit-db.com/exploits/15304/*.

[35] PAX-TEAM. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt.

[36] PAYER, M. String oriented programming - circumventing aslr, dep and other guards. In *28c3'11: Proc. 28th Chaos Communication Congress* (2011).

[37] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments* (2011), pp. 157–168.

[38] PRASAD, M., AND CKER CHIUEH, T. A binary rewriting defense against stack based buffer overflow attacks. In *Proc. 12th USENIX ATC* (2003), pp. 211–224.

[39] PROVOS, N. Improving host security with system call policies. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003).

[40] ROSENBERG, D. CVE-2010-0830: Integer overflow in ld.so. *http://drosenbe.blogspot.com/2010/05/integer-overflow-in-ldso-cve-2010-0830.html*.

[41] SCO. System V Application Binary Interface, Intel386 Architecture Processor Supplement. *http://www.sco.com/developers/devspecs/abi386-4.pdf* (1996).

[42] SCOTT, K., AND DAVIDSON, J. Strata: A software dynamic translation infrastructure. Tech. rep., University of Virginia, 2001.

[43] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. *ACSAC'02: Annual Comp. Security Applications Conf.* (2002), 209.

[44] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th Computer and Communications Security* (2007), pp. 552–561.

[45] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE'06: Proc. 2nd Virtual Execution Environments* (2006), pp. 175–185.

[46] VAN DE VEN, A., AND MOLNAR, I. Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf.

[47] 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. In *IEEE Symp. on Security and Privacy* (2009), pp. 79–93.