secuBT: Hacking the Hackers with User-Space Virtualization

Mathias Payer
Department of Computer Science
ETH Zurich

Abstract

In the age of coordinated malware distribution and zero-day exploits security becomes ever more important. This paper presents *secuBT*, a safe execution framework for the execution of untrusted binary code based on the *fastBT* dynamic binary translator.

secuBT implements user-space virtualization using dynamic binary translation and adds a system call interposition framework to limit and guard the interoperability of binary code with the kernel.

Fast binary translation is a key component to user-space virtualization. *secuBT* uses and extends *fastBT*, a generator for low-overhead, table-based dynamic (just-in-time) binary translators. We discuss the most challenging sources of overhead and propose optimizations to further reduce these penalties. We argue for hardening techniques to ensure that the translated program cannot escape out of the user-space virtualization.

An important feature of *secuBT* is that only translated code is executed. This ensures code validity and makes it possible to rewrite individual instructions. The system call interposition framework validates every system call and offers the choice to (i) allow it, (ii) abort the program, (iii) redirect to an user-space emulation.

1 Introduction

In a world of increasing software complexity and diversity it is important to sandbox and virtualize running applications. Staying up-to-date on all software applications and libraries is hard but necessary. On the other hand patches as well as malware discovery is reactive. *secuBT* is a step towards proactive security and fault detection.

Two interesting scenarios are (i) sandboxing of server processes, and (ii) execution of untrusted code. Server daemons that offer an interface to the Internet are under constant attack. If these daemons are virtualized and restricted to only specific system calls and parameters then an exploit is unable to escalate privileges and execute unintended code. The second scenario targets plugins and other downloaded untrusted code that a user would like to run in a special sandbox. Whenever the program issues a system call, the user can decide if it is allowed or not. The sandbox limits the possible interactions between the virtualized program and the operating system. System calls are checked depending on name, parameters, and call location.

Such a sandbox could be implemented as a kernel module. This would require additional code that runs in the kernel itself which poses an additional security risk. User-space virtualization uses binary translation to sandbox processes. The virtualization layer controls all instructions that can be executed by the virtualized process. System calls are rewritten by the translation system so that an authorization function is executed first. This authorization decides whether a system call (i) is allowed, (ii) aborted, or (iii) redirected to a sandbox-internal function that emulates the system call in user-space. To the virtualized application it appears as if the system call is executed directly.

We propose a sandbox that is built on a dynamic binary translation (BT) system. Using BT ensures that all executed code is translated first, and therefore a program cannot escape the sandbox. If the translator encounters unsafe or invalid code it aborts the program. This builds a safe execution framework that validates executed code. The translator can rewrite individual instructions, e.g. redirect system calls to handler functions.

Section 2 covers the design of the encapsulation framework and implementation of the binary translator *fastBT* [14]. Section 3 elaborates the security extension and the system call interposition framework. The system is evaluated in Section 4, followed by related work and our conclusion.

2 fastBT: Dynamic Binary Translation

Fast binary translation is the key component to implement user-space virtualization. The dynamic translation system checks and verifies every machine code instruction before it is executed. Static binary translation does not suffice because it is unable to detect hidden code or malicious code that targets the static binary translator itself. Direct control transfers are translated and redirected to the code cache. Indirect control transfers are translated into an online lookup and dispatch to guarantee that only translated branch targets are reached. The translation system can change, adapt, or remove any invalid instruction. Using the translation system, system calls are rewritten and redirected to an interposition system.

To limit the overhead of binary translation the translation process must be fast and must produce competitive code. This section presents implementation details of *fastBT* [14], a fast and flexible binary translator that is used to implement the *executable space protection* and *system call interposition*. An important feature of *fastBT* is that the return addresses on the stack remain unchanged. This adds additional complexity to handle return instructions as they are translated to a lookup and an indirect control transfer. An unchanged stack has the following advantages: (i) the program can validate and read its own call stack for debugging, (ii) exception handling uses return addresses to escape to the correct frame, (iii) the program does not know that it is translated, and (iv) the address of the code cache is hidden from the program.

2.1 Basic Translator

The translator processes *basic blocks* of the original program, places them in the *code cache* and adds entries to the *mapping table*. The mapping table is used to map between program locations in the original program and the code cache. This translation process ensures that the execution flow always stays in the code cache. If the program is about to branch to an untranslated block of code then the translator is invoked to translate that block. The translated block is then placed in the code cache and the execution of the program resumes at the newly translated block in the code cache. See Figure 1 for an overview of such a generic translator.

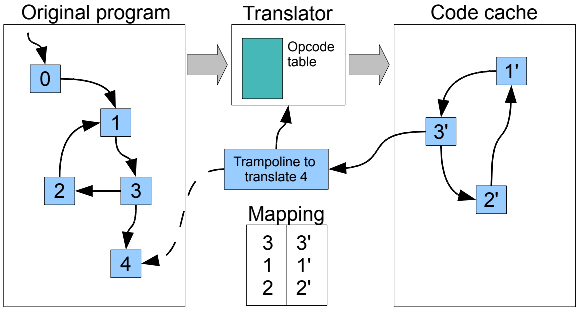

Figure 1: Runtime layout of the binary translator.

2.1.1 Translation Tables

The basic translation engine is a simple table-based iterator. When the translator is invoked with a pointer to a basic block, it first adds a reference to the mapping table from the original program location to the location in the code cache. The iterator then loops through the basic block one instruction at a time.

To decode the variable-length x86 instructions a large multidimensional translation table is constructed that encodes all possible combinations of machine code instructions and parameters. Starting with a base table each byte of the current instruction is checked. If the decoding of the instruction is not finished in the current table, a pointer redirects to the next table and the decoding process continues with the next byte. This process is repeated until the instruction is completely decoded.

As a next step the instruction and its arguments are passed to a corresponding *action function* that handles the translation of this particular instruction. Action functions can generate arbitrary code, alter, copy, or remove the instruction. Generated code is emitted into the code cache.

The translation process stops at recognizable basic block (BB) boundaries like branches or return instructions. Some BB boundaries like backward jumps are not recognizable, in such a case a part of the BB will be translated a second time.

At the end of a BB the translator checks if the outgoing edges are already translated, and adds jumps to the translated targets. If a target is not translated, the translator builds a trampoline that starts the translation engine for the corresponding BB, and adds a jump to the trampoline.

2.1.2 Predefined Actions

The translator needs different action functions to support the identity transformation. All safe instructions that are executable in user-space are copied to the code cache. Privileged instructions (e.g. all instructions that are not allowed to execute in user-space) are intercepted and the program is aborted, otherwise the kernel would signal a segmentation fault or general protection fault.

Special care is needed for control instructions. These are handled by specific action functions that check the target of the branch instruction. These action functions emit extra code that gives the illusion as if the original code was executed.

- The *call action handler* issues code that saves the correct instruction pointer of the original location on the stack, but branches into code that is located in the code cache.
- The *ret action handler* emits code that pops the return instruction pointer from the stack, finds the translated target and branches to the corresponding translated target.
- The *indirect jump action handler* emits code for a runtime lookup of the corresponding target and issues a branch to the translated target.

If the target is not already translated then the action function generates a trampoline that translates the target when it is executed for the first time.

2.1.3 Code Cache

During the execution of a program it is likely that certain code regions are executed multiple times. Therefore it makes sense to keep translated code in a cache for later reuse. As a result code is only translated once, reducing the overall translation overhead.

fastBT uses a per thread cache strategy to increase code locality. An additional advantage of thread local caches is that the action functions can emit hard-coded pointers to thread local data structures, otherwise the code would need to call expensive lookup functions.

The combination of the basic translator that stops at BB boundaries and the design of the code cache lead to a greedy trace extraction. These traces are formed as a side effect of the first execution and can possibly speed up the program.

2.1.4 Mapping Table

The mapping table maps between code locations in the original program and code locations in the code cache. The translator adds entries for all translated basic blocks. An entry consists of two pointers, the first pointing to the original location, the second pointing to the translated location.

The hash function used in any lookup function returns a relative offset into the mapping table. Adding the result from the hash function to the base pointer of the mapping table results in the address of a table entry. This process is kept simple so that inlined machine code is able check these entries efficiently.

2.2 Optimizations

The basic, simple binary translator is able to use hardcoded references in the code cache for direct control transfers. The translator must emit an online lookup of the target if the control transfer is indirect (e.g. indirect jump, indirect call, function return, or newly translated block), and the target is not known at translation time. This *indirect_jump* routine resolves the indirect target, issues a mapping table lookup, translates new targets, and redirects the control flow to the specified target.

The execution of these indirect control transfers sums up to the majority of the overhead introduced through BT. Different optimization strategies like function inlining, an inlined lookup and dispatch, and inlined predictions are used to reduce the overhead of indirect control transfers.

2.2.1 fastRET

Function returns are indirect control transfers where the target lies on the top of the stack. The basic translator calls the *indirect_jump* routine and handles the return instruction like an indirect jump.

The *fastRET* optimization translates a return instruction into a thread local lookup in the mapping table and a branch to the translated target without an additional call.

Using the implementation shown in Figure 2 the *fastRET* optimization uses 13 instructions compared to more than 20 instructions if the general indirect jump routine is used. The *fastRET* optimization has two advantages: (i) no lookup is needed for pointers to local data structures, they can be embedded in the code cache, (ii) the code is inlined in the code cache, and no extra function call is executed.

```
pushl %ebx
pushl %ecx
movl 8(%esp), %ebx # load rip
movl %ebx, %ecx
andl HASH_PATTERN, %ebx
subl maptbl_start(0,%ebx,8), %ecx
jecxz hit
popl %ecx
popl %ebx
pushl tld
call ind_jmp
hit:
movl maptbl_start+4(0,%ebx,8), %ebx
movl %ebx, tld->ind_jump_target
popl %ecx
popl %ebx
leal 4(%esp), %esp # adjust stack
jmp *(tld->ind_jump_target)
```

Figure 2: A translated return instruction with the fast return optimization, *rip* is the return instruction pointer and *tld* the pointer to thread local data.

2.2.2 Indirect call prediction

The *indirect call prediction* caches the last lookup target and destination (see Figure 3). If the target is the same then the control can be transfered without a mapping table lookup, otherwise the cache must be updated. This optimization speculates that the target does not change often. *fastBT* uses this optimization for indirect jumps relative to a memory address (e.g. `jmp *0x11223344`).

```
cmpl $cached_target, (%esp)
je hit_tr
pushl target
pushl tld
pushl $addrOfCachedTarget
call indcall_fixup
hit_tr:
pushl rip
jmp translated_target
```

Figure 3: Translated indirect call instruction with an included prediction, *target* is the target of the call instruction, *rip* is the return instruction pointer, and *tld* the pointer to thread local data.

3 Security Features

A program is secure if the execution of unintended code is not possible. Unintended code does not meet the users expectations and executes unintended system calls. If all system calls of an application are redirected to an interposition framework and inspected then the program is unable to execute unintended actions like, e.g., deleting files, escalating privileges, or spawning processes.

secuBT offers the possibility to inspect system calls before they are executed. Using this framework malicious code is unable to break out of the sandbox, even if some unwanted code is executed. To support this user-space virtualization the underlying binary translator must ensure that a program is not able to escape BT. If only safe instructions are translated and all system calls are checked then a program will not be able to escape out of the virtualization. Unsafe code is caught during the translation process, and the program is terminated. Unsafe system calls are caught before they are executed and the program is terminated.

The system call interposition framework can be extended by a policy-based user-space system call authorization framework [13].

User-space virtualization does not guard against overwriting data structures in files that are already mapped to memory.

3.1 Executable Space Protection

Modern processors support a memory management extension called executable bit. Only code on pages where this bit is set can be executed. This protection guards against introduced code in unexecutable regions like the stack.

secuBT implements such a protection mechanism on a more precise per-section granularity for regions defined in the ELF headers. The translator checks for every memory location before it is translated if it is in an executable section of any loaded library or the program itself. The program is terminated if the target is not in an executable section. This mechanism protects against code introduced into non-executable regions like the stack and the heap.

The code regions of the original program are marked as non-executable and only the code cache contains executable code. As the user program does not know the location of the code cache it is unable to change already emitted code.

3.2 Protecting Internal Data Structures through `mprotect`

A translated program is not able to directly discover addresses of internal data structures of the binary translator. But because the binary translator and the user program share the same address space there is a probabilistic chance that the user program can change memory locations that belong to the binary translator.

An explicit protection can be achieved by write-protecting all memory pages used by the binary translator. As soon as the binary translator switches to translated code write-bits are cleared for all pages used by secuBT. If an untranslated basic block is translated then the write privileges are added before the translator is started. These transitions are controlled by secuBT and make it impossible for the user program to change emitted code in the code cache. This extensions adds a lot of additional `mprotect` system calls. The `mprotect` system calls are responsible for a lot of the translation overhead but result in a higher level of security.

3.3 System Call Interposition

To be effective an exploit must execute unintended system calls (e.g., I/O, opening network sockets, executing other programs, and privilege escalation). A mechanism that restricts the system calls a virtualized program can execute is an effective safeguard.

The *fastBT* framework supports system call rewriting of `sysenter` instructions and `int 80` instructions. Whereas the Linux kernel uses both systems alongside each other [9].

The system call interposition framework extends the binary translator and redirects all system calls to specific user-defined functions. It is possible to define a specific function per system call that checks the parameters, call stack, and the system call number and either allows the system call, denies the system call and aborts the program, or denies the system call and returns a fake value.

4 Evaluation

This section provides an extensive analysis of the overhead introduced through the *fastBT* virtualization and *secuBT* sandboxing. This overhead is separated into (i) BT overhead alone, (ii) additional overhead for syscall validation and executable space protection, and (iii) full virtualization using `mprotect` to guard the internal datastructures from attacks against the sandbox.

The benchmarks were run on an Ubuntu 9.04 system with an E6850 Intel Core2Duo CPU running at 3.00GHz, 2GB RAM, and GCC version 4.3.3.

4.1 Overhead for different security configurations

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

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

secuBT: This configuration shows the overhead of *secuBT* with executable space protection and system call interposition.

secuBT (full): The last configuration shows full encapsulation including protection of internal data structures using explicit memory protection through `mprotect`.

Benchmark	**fastBT**	**secuBT**	**full sBT**
400.perlbench	66.87%	67.70%	72.22%
401.bzip2	4.34%	3.89%	4.19%
403.gcc	32.20%	31.97%	84.81%
429.mcf	0.25%	0.00%	0.25%
445.gobmk	15.71%	15.71%	18.00%
456.hmmer	4.54%	5.51%	5.94%
458.sjeng	36.04%	35.90%	35.76%
462.libquantum	0.98%	0.98%	0.98%
464.h264ref	8.19%	10.21%	10.21%
471.omnetpp	16.53%	15.73%	15.93%
473.astar	5.49%	5.32%	5.16%
483.xalancbmk	30.19%	29.38%	32.35%
410.bwaves	2.35%	2.46%	2.68%
416.gamess	-3.50%	-2.80%	-2.10%
433.milc	2.30%	2.18%	2.54%
434.zeusmp	-0.25%	-0.25%	-0.13%
435.gromacs	0.00%	0.00%	0.00%
436.cactusADM	-0.66%	0.00%	0.00%
437.leslie3d	0.00%	0.00%	0.00%
444.namd	0.49%	0.49%	0.49%
447.dealII	46.38%	44.02%	45.11%
450.soplex	4.83%	4.46%	6.69%
453.povray	39.23%	39.78%	41.44%
454.calculix	-1.12%	-1.12%	3.35%
459.GemsFDTD	1.79%	1.79%	2.68%
465.tonto	11.35%	10.27%	13.51%
470.lbm	0.00%	-0.11%	0.00%
482.sphinx3	0.35%	0.95%	1.42%
Average	**11.60%**	**11.59%**	**14.41%**

Table 1: Overhead for different configurations executing the SPEC CPU2006 benchmarks (relative to an untranslated run). The configurations are fastBT, secuBT, and secuBT with full memory protection.

The average slowdown for *fastBT* below 12% is tolerable. *secuBT* security extensions do not add any measurable overhead compared to *fastBT*. The full protection mechanism results in an overhead of 14.41%. The overhead for *fastBT* and basic *secuBT* protection for most programs is between 0% and 10% whereas some benchmarks like 400.perlbench, 433.gcc, 453.sjeng, 483.xalancbr, 447.dealII, and 453.povray result in a higher overhead of 30% to 67%. *secuBT* adds static overhead per translated block and per system call. The SPEC CPU20006 benchmarks have a low number of system calls and high code reuse which is typical for server applications. Therefore the *secuBT* extensions add no measurable overhead to these programs.

secuBT with full protection leads to more overhead because the number of system calls increases. But the overall overhead is low in these benchmarks, although the translation overhead is a lot higher. The translation overhead is small compared to the runtime overhead of the translated program. As soon as all active code is translated no further memory protection calls are necessary. For short running programs with low code-reuse the translation overhead would be higher.

5 Related Work

Related work to *secuBT* 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.

As *secuBT* is tightly coupled to the *fastBT* binary translator this section covers related work from binary translation as well.

5.1 Enforcing Security

Security can be enforced on many levels. Some of them are limiting the system calls a program can use, limiting the instruction set, or using hardware extension to limit the program.

- The Google Native Client [18] is able to execute x86-code in a sandbox. Native client uses techniques similar to software-based fault isolation (SFI) [17] systems. The instruction set is limited to a save subset of the IA-32 ISA, making illegal operations impossible. Before the execution a verifier checks if the program is valid, then the program is executed without any additional virtualization. This limits the possible used instructions, the programs are linked statically and no dynamic libraries can be used. Programs must be compiled with a custom-tailored compiler.

- Janus [10] 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, this poses a potential security problem, and (ii) the overhead of switching between the inspecting process and the application is high.

- Vx32 [8] 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.

5.2 Binary Translation

Complete system virtualization by QEMU [3] offers full encapsulation, but comes with high overhead. Other full system virtualization tools like VMware [7; 6] and Xen [2] rely on kernel support. A disadvantage of system virtualization is that every VM is an independent system with its own configuration. From a security and safety perspective the encapsulation of such an approach is needed without the complexity of individual systems is needed. *secuBT* offers user-space virtualization, combining encapsulation with simple configuration on a single system.

The three binary translators that are most similar to *fastBT* in either architecture or function are HDTrans, DynamoRIO, and PIN:

- HDTrans [16; 15] is a light-weight, table-based instrumentation system. A code cache is used for translated code as well as trace linearization and optimizations for indirect jumps. HDTrans resembles fastBT most closely with respect to speed and implementation, but there are significant differences in the optimizations for indirect jumps. Additionally HDTrans only covers a subset of the IA-32 instruction set.

- Dynamo is a dynamic optimization system developed by Bala et al. [1]. DynamoRIO [4; 5; 11] is the binary-only IA-32 version of Dynamo for Linux and Windows. The translator extracts and optimizes traces for hot regions. Hot regions are identified by adding profiling information to the instruction stream. These regions are converted into an IR, optimized and recompiled to native code. DynamoRIO was recently acquired by Google and newer versions are released as open-source.

- PIN [12] is an example of a dynamic instrumentation system that exports a high-level instrumentation API that is available at runtime. The system offers an online high-level interface to all instructions. PIN uses the user-supplied definition and dynamically instruments the running program.

The drawback of these systems is that they were not designed with security in mind. A binary with special crafted instructions is able to escape the binary translation system and can execute untranslated code, circumventing the protection mechanisms.

6 Conclusion

We present *secuBT*, a low overhead binary translation framework that implements security in user-space and offers the ability to limit programs in their use of system calls and privileged instructions.

The *secuBT* framework uses dynamic binary translation to support the full IA-32 ISA without kernel support. Full binary translation escapes dangerous instructions at runtime and interposes system calls with an interposition framework.

secuBT combines the advantages of full system translation without the disadvantages. Applications are virtualized and encapsulated while using a shared system image with a single system configuration and no additional overhead to run multiple operating systems.

The source code of the secuBT virtualization framework can be downloaded at *http://nebelwelt.net/projects/secuBT*.

References

[1] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic op-

timization system. In *PLDI '00* (Vancouver, BC, Canada, 2000), pp. 1–12.

[2] 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.

[3] Bellard, F. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proc. conf. USENIX Ann. Technical Conf.* (Berkeley, CA, USA, 2005), pp. 41–41.

[4] 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).

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

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

[7] 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.

[8] 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.

[9] Garg, M. Sysenter based system call mechanism in linux 2.6 (http://manugarg.googlepages.com/systemcallinlinux2_6.html).

[10] 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).

[11] Hazelwood, K., and Smith, M. D. Managing bounded code caches in dynamic binary optimization systems. *TACO '06: ACM Trans. Arch. Code Opt. 3*, 3 (2006), 263–294.

[12] 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.

[13] Payer, M., and Gross, T. secuBT: Enforcing security through user-space process virtualization. Tech. rep.

[14] Payer, M., and Gross, T. Requirements for fast binary translation. In *2nd Workshop on Architectural and Microarchitectural Support for Binary Translation* (2009).

[15] Sridhar, S., Shapiro, J. S., and Bungale, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[16] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. P. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE '06* (New York, NY, USA, 2006), pp. 175–185.

[17] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1993), ACM, pp. 203–216.

[18] 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.