# DynSec: On-the-fly Code Rewriting and Repair

Mathias Payer
*ETH Zurich and UC Berkeley*

Boris Bluntschli
*ETH Zurich*

Thomas R. Gross
*ETH Zurich*

## Abstract

Security patches protect an application from discovered vulnerabilities and should be applied as fast as possible. On the other hand, patching the application reduces the availability of the service due to the necessary restart. System administrators need to balance system availability with a potential compromise of system integrity.

A dynamic software update mechanism applies security updates on the fly but does not protect from unknown vulnerabilities. Software-based fault isolation on the other hand uses a sandbox to protect the integrity of a system by detecting unpatched vulnerabilities but provides no mechanism to repair any vulnerabilities.

This paper presents DynSec, a mechanism for on-the-fly code rewriting and repair that dynamically applies security patches for unmodified binary applications. A sandbox protects the integrity of the system while the dynamic update mechanism increases the availability of the application. A prototype implementation that needs no a-priori cooperation from the application incurs a combined overhead of 11% on the SPEC CPU2006 benchmarks for the sandbox and the dynamic update mechanism.

## 1 Introduction

Security vulnerabilities threaten the *integrity* and *availability* of running applications and systems by exploiting software bugs to run malicious code. A bug that is recognized (and for which the patch is available but not yet installed) might leave two venues of attack: the bug may provide a base for an exploit or, if the bug causes the service to be shut down, the bug may be used for a denial of service attack. In either case, the integrity of the system and the availability of the service are jeopardized.

The focus of past projects providing continuous protection has been either sandboxing or dynamic software modification. *Sandboxing* [11, 15, 23, 26] is a dynamic form of Software-based Fault Isolation (SFI) [18, 31, 33] that uses dynamic binary translation [16, 21, 23] to protect running applications against known and unknown bugs. The *integrity* of the system is guaranteed but the service is terminated when attacked. Attacks are detected after-the-fact (i.e., after the runtime state is compromised) and the application is terminated to protect the integrity of the system. *Aspect-oriented programming* and *dynamic software updating*, on the other hand, allow code to be inserted, removed, or updated at runtime [1, 7, 13, 17, 19, 20, 25, 27, 29]. Dynamic updating systems increase the *availability* of a service by applying software updates without any downtime. These approaches protect a service against attacks where a patch is available but not against unknown attacks.

We discuss how to combine sandboxing with a dynamic software update system, thereby both guaranteeing *system integrity* and maximizing *service availability*. The updating mechanism is integrated into the virtualization layer of the sandbox. The binary translation component of the sandbox makes the integration of patches straightforward. We implemented a prototype system (DynSec) to assess its effectiveness (i.e., can the system handle security-related updates) and its efficiency (i.e., what runtime overhead must be paid for this capability). Compared to related work DynSec handles any unmodified binary by injecting a user-space virtualization layer into the executing application to add both the sandbox and the update mechanism – without a-priori changes to the binary itself, the programming language, the source code, the compilation toolchain, or the runtime system. DynSec supports patches that change any set of instructions in the original application but does not recover types or data layout from binary code.

The sandbox component of DynSec protects the integrity of the service at all times. The dynamic update mechanism installs available patches on-the-fly when they become available to maximize the availability of the service. The contributions of this paper are:

1. Description of the design and implementation of *DynSec*, a dynamic on-the-fly software update mechanism for unmodified binaries to provide system integrity and service availability;

2. A performance evaluation of a prototype implementation of DynSec for unmodified x86 Linux applications using the SPEC CPU2006 benchmarks;

3. A discussion of the patch application process using multiple versions of the CoreHTTP server.

## 2 The dynamic update mechanism

Currently a system administrator must balance availability and integrity. Whenever a bug is discovered and a new patch is available all instances of an application must be updated and restarted to ensure the system's integrity. On the other hand, an administrator wants to minimize the downtime and the number of restarts to maximize availability. Most services are not designed with updateability in mind and they do not support partial upgrades of individual components. Related work approaches either the updateability or the integrity problem (see Section 5); DynSec is the first solution that combines a dynamic update mechanism for security updates with a user-space sandbox to provide both integrity and availability.

DynSec uses a dynamic binary translator (DBT) to *transparently* weave security updates on-the-fly into the executed application. At the same time the binary translation component implements a secure sandbox that provides code integrity, stack integrity, and a system call policy. A patch consists of a set of replaced instructions accompanied with a shared library. The shared library is used to add new functions and data that are reachable from the patched instructions (i.e., if additional helper functions are added through the security patch). This paper focuses on the details of the dynamic update system and the interaction into a software-based fault isolation sandbox; we assume that security patches are available.

Figure 1 shows the DynSec runtime system. The DynSec module builds on the DBT. The data structures are extended with a list of patches that is shared among all threads of the application. Patches are handled during the translation process of the DBT and the original memory layout of the application remains unchanged. Every application thread has its own DBT code cache. The translator emits custom-tailored code for each thread. The translation process is extended with a lookup in the list of patches before each instruction is translated and, if

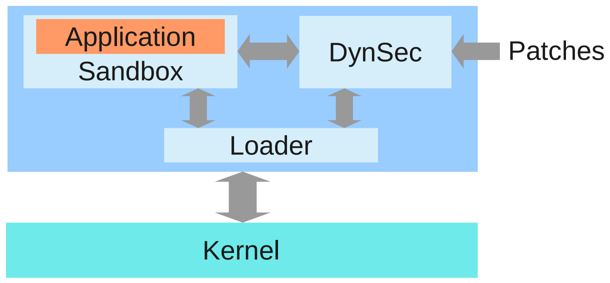

Figure 1: Overview of the DynSec runtime approach.

a patch is available, the patched instruction is translated instead of the original instruction.

This paper addresses two problems of existing dynamic software update mechanisms: (i) existing systems require the modification of either the source-code, compilation toolchain, or the virtual machine and (ii) many tools rely on a whole-system analysis to guarantee the validity of a patch. DynSec supports binary-only applications without prior modification or analysis and supports dynamically loaded shared libraries or modules as well.

### 2.1 Code translation

DynSec builds on top of a DBT and extends the translator to replace arbitrary instructions during the code translation process. A table-based DBT uses translation tables to translate individual instructions; translated basic blocks are placed in a code cache to reduce translation costs and a mapping table links instructions in the original code and instructions in the code cache.

A table-based translator translates individual instructions in three steps: (i) checking if the instruction has already been translated (if so, finish translation of the instruction and add a branch to the translated target in the code cache), (ii) decoding the original instruction with the translation tables, and (iii) emitting the translated instruction to the code cache.

DynSec applies patches indirectly by flushing the code cache. In addition, the translation process is extended at the second step with a check that looks for a patch at the current instruction. If the current instruction is patched then the patched instruction is translated (and all following instructions in the patch until the end of the basic block in the patch) and not the original instruction. This approach enables us to replace a sequence of instructions in the original code without changing the original code in place. Instead of executing the original instruction the virtualization layer takes care to transparently execute the new (patched) instructions. This approach of using a virtualization layer between original code memory pages and executed code solves the problem of patching code that is longer than the original code. Original instructions can be removed or replaced (with a sequence of instructions that is longer or shorter than the original instruction) without the need to re-layout code, i.e., adjusting all the relative offsets for control flow transfers.

In addition, the modified just-in-time code translator adds (virtual) safe-points at the end of each translated basic block. The safe-point code checks if the current thread should stop executing translated application code and branch into the translator to synchronize with the patching thread, i.e., to flush the code cache when a new patch is available.

## 2.2 Patching architecture

By extending the DBT-based virtualization layer DynSec uses information from the DBT loader component to locate the target locations of individual patches. Using the loader component from the DBT DynSec locates all loaded symbols, libraries, and modules. In addition, DynSec keeps an updated list of all running threads by observing thread creation and destruction system calls.

DynSec injects an additional patching thread into the application that runs as part of the privileged sandbox domain. This patching thread waits for incoming patches and synchronizes the code update between application threads and the DynSec module. A patch is applied in three steps: (i) the patching thread waits for an incoming patch and updates a shared data structure, (ii) the patching thread signals and synchronizes all running application threads to stop at the next (virtual) safe-point, and (iii) the patches are applied by the individual application threads by flushing their code cache. Sleeping threads are updated directly upon returning from kernel-space.

Code that is added through a patch is protected and translated by the sandbox like original application code.

## 2.3 Patch format

Patches are based on instruction-level granularity and allow replacement, removal, and addition of instructions. Such patches are expressive enough for most security updates (e.g., length checks against buffer overflows, range checks, or format string checks). Type changes and data layout changes (e.g., additional members in C-structures, new members in C++ objects, or massive code restructuring) are out of scope due to missing high-level information in binary-only applications.

Patches are defined in a binary format: the patch starts with the number of patched instructions, followed by the individual instructions. Each instruction consists of (i) the patched address (relative to a given code module), (ii) the length of the original instruction, (iii) the length of the patched instruction, and (iv) the machine code of the new instruction. In addition, each patch can specify a shared library that is loaded into a local scope to add additional functionality (e.g., a sanitizer function that is called from the patched instructions). This scope is only accessible by the patched instructions.

## 2.4 Patch extraction

The focus of this paper is not patch extraction but the design of a system that enables hot patching and code modification of binary-only applications. For our test patches we use a simple programmer-guided tool. The tool is built on `objdump` and analyzes differences between two versions of a binary on a per-function basis, changed instructions are added to the patch. We then check and fine-tune the generated patch manually.

Future work can extend this simple programmer-guided patch generation approach by using work in binary diffing. Binary diffing extracts changes between two versions of the same application. Work on binary diffing concentrates on different levels of granularity (e.g., the program level [7], the function level, the basic block level [5, 9, 10, 12], the instruction level, or the control flow [14]) to define the differences between two versions. Several tools [9, 10] work on the basic block level using graph-based comparison of the two application versions.

## 3 Implementation

The prototype implementation of DynSec extends TRuE [22, 23], which combines an SFI sandbox with a trusted loader and follows the design in Section 2. Advantages of TRuE are that the system separates user-space into two privilege domains: the application domain and the sandbox domain. Code in the application domain is dynamically checked for security violations and indirect control flow transfers are confined to valid targets.

TRuE uses a per-thread binary translator to dynamically translate all application code. DynSec modifies the translator in two ways: first, the translator checks for every translated instruction if a patch is available. This check for patches is executed during the translation and does not incur any runtime overhead in the translated code. Second, the translator adds a virtual safe-point at the end of every basic block to ensure atomicity and consistency when synchronizing between multiple application threads. Extended basic blocks end with either a relative control flow transfer to another basic block or in a trampoline for an indirect control flow transfer. The safe-point implementation keeps a list of all relative control flow transfers and all indirect control flow transfers. The patching thread signals the application threads to take the safe-point by overwriting the targets of the relative control-flow transfers and trampolines to a catch-function in the translator. The application thread will execute the relative control flow transfer or the trampoline and end up in the catch function in the translator. Such an implementation has no runtime overhead for executed translated application code and minimal overhead when a safe-point is taken.

The DynSec prototype implementation loads patches using an additional thread that is part of the sandbox. This additional thread takes care of (i) the shared patching data structures for all application threads as well as (ii) signaling running threads that they should stop at the next safe-point. Sleeping threads return into the DBT upon system call completion where updates are handled by flushing the code cache before any translated instruction is executed. The open-source prototype implementation of DynSec uses less than 2000 lines of code and 45 lines of code are changed in TRuE to add hooks for the DynSec functionality.

## 4 Evaluation

This section evaluates the DynSec implementation prototype. To evaluate a dynamic software update mechanism two aspects are important: (i) correctness of the patching infrastructure and (ii) performance of the prototype implementation.

We evaluate the performance using the SPEC CPU2006 benchmarks. In addition, the correctness of the patching infrastructure is evaluated using a set of CoreHTTP versions where successful updates are tested using specific security exploits. All benchmarks are executed on an Intel Core 2 Quad Q6600 with 2.64GHz and 8GB RAM on Ubuntu 11.04 with Linux kernel 2.6.38 and the GNU compiler collection 4.5.1.

### 4.1 SPEC CPU2006 performance

The performance of the sandbox and the DynSec module is evaluated using the SPEC CPU2006 benchmarks version 1.0.1. Some benchmarks (447.dealII, 481.wrf, 473.astar, and 483.xalancbmk) are not compatible with the recent GNU compilers. These benchmarks compile only with GCC 2.95 and are excluded from the performance evaluation.

Table 1 shows performance results for the SPEC CPU2006 measurements (executed with the default configuration using the `real` data-set and 3 iterations).

| **Benchmark** | **Nat.** | **SB** | **Ovhd.** | **DS** | **Ovhd.** |
|---|---|---|---|---|---|
| 400.perlbench | 578 | 1093 | 89% | 1117 | 93% |
| 401.bzip2 | 876 | 925 | 5.6% | 927 | 5.9% |
| 403.gcc | 490 | 659 | 35% | 659 | 35% |
| 429.mcf | 419 | 422 | 0.56% | 420 | 0.08% |
| 445.gobmk | 748 | 920 | 23% | 920 | 23% |
| 456.hmmer | 788 | 805 | 2.2% | 805 | 2.3% |
| 458.sjeng | 863 | 1303 | 51% | 1297 | 50% |
| 462.libquantum | 1117 | 1117 | 0.0% | 1110 | -0.60% |
| 464.h264ref | 1180 | 1685 | 43% | 1680 | 42% |
| 471.omnetpp | 519 | 688 | 33% | 684 | 32% |
| 473.astar | 736 | 821 | 12% | 820 | 11% |
| 410.bwaves | 901 | 929 | 3.1% | 931 | 3.3% |
| 416.gamess | 1633 | 1787 | 9.4% | 1753 | 7.4% |
| 433.milc | 855 | 875 | 2.4% | 874 | 2.2% |
| 434.zeusmp | 889 | 883 | -0.64% | 882 | -0.71% |
| 435.gromacs | 1823 | 1830 | 0.37% | 1827 | 0.18% |
| 436.cactusADM | 1807 | 1800 | -0.37% | 1803 | -0.18% |
| 437.leslie3d | 1030 | 1030 | 0.00% | 1033 | 0.32% |
| 444.namd | 780 | 788 | 1.0% | 789 | 1.2% |
| 450.soplex | 521 | 553 | 6.1% | 553 | 6.1% |
| 453.povray | 442 | 674 | 52% | 652 | 47% |
| 454.calculix | 1857 | 1880 | 1.3% | 1870 | 0.72% |
| 459.GemsFDTD | 1057 | 1073 | 1.6% | 1080 | 2.2% |
| 465.tonto | 1053 | 1173 | 11% | 1177 | 12% |
| 470.lbm | 1010 | 997 | -1.3% | 1009 | -0.07% |
| 482.sphinx3 | 933 | 948 | 1.6% | 951 | 2.0% |
| Mean | 958 | 1064 | 11% | 1062 | 11% |

Table 1: Overhead for the SPEC CPU2006 benchmarks for the sandbox (SB) and DynSec (DS). Runtime is in seconds, relative overhead is compared to native execution.

The SPEC CPU2006 programs provide long running tests that can be used to evaluate any overhead incurred by long running applications. The mean overhead for the sandbox is 11%. The main overhead comes from the execution of the translated indirect control flow transfers (e.g., indirect jumps, indirect calls, and function returns). The DynSec module, the patching thread, and the additional lookup for each translated instruction do not add any noticeable overhead.

### 4.2 CoreHTTP security study

CoreHTTP [32] is a minimalistic webserver with CGI support written in C. CoreHTTP is not as mature as Apache and therefore serves as a great example to study security bugs. This study uses the CoreHTTP versions 0.5.3alpha with one known security vulnerability CVE-2007-4060 [30] and 0.5.3.1 with two known security vulnerabilities CVE-2009-3586 [2] and ExploitDB-10610 [8].The vulnerabilities lead to remote code execution or remote command execution. We evaluated CoreHTTP (364kB compiled code) in a simple forking configuration where we patch one executing application thread using our additional patching thread.

All three vulnerabilities are located in the file `http.c` and can be fixed with simple patches. The simple patches currently patch the call instruction with a new target that contains the fixed code. The different vulnerabilities are:

CVE-2007-4060: Buffer overflow through missing input sanitation in static string passed to `sscanf`.

CVE-2009-3586: Off by one error in input sanitation, resulting in an 1 byte buffer overflow.

ExploitDB-10610: Arbitrary command execution due to missing input sanitation in URL parameter. CoreHTTP calls `popen` with an unescaped input string.

The dynamic patches fix the vulnerabilities by extracting the vulnerable function into a separate C file, fixing the vulnerability, and compiling the C file into a shared library (using the same compiler and flags as for the original binary). This shared library is then added to the patch and the patch redirects the vulnerable function to the newly loaded shared library.

The testing framework of the prototype implementation uses CoreHTTP and these three exploits as unit tests. DynSec successfully patches all three security vulnerabilities at runtime and protects the application from the exploits.

## 5 Related work

DynSec focuses on dynamic patch application and supports patches that may change any part of the code segment in the original program. Some related projects [4, 24] support high-level type reconstruction and data layout recovery that are out of scope for DynSec.

Dynamic software update mechanisms use some form of runtime system to handle patches and to synchronize patch application with the executing process (and threads). Drawbacks of current systems are that most of them do not support binary-only applications or threads and need changes in the compilation chain [1,3,7,13,17, 19,20,29] or changes in the programming language [13]. DynSec supports binary-only applications without prior compiler involvement or changes in the programming language.

Adding a runtime system in the compilation toolchain limits the possible changes that can be carried out by the patches. Many of the dynamic update systems allow global changes only, i.e., replacing the complete runtime image [7, 13]. Other systems work on the function level [1, 6, 28]. DynSec enables patching on instruction-level granularity for unmodified binaries.

Many of the existing systems state that they only support single threaded applications [13, 20, 27] while DynSec supports multiple concurrent threads and takes special care for synchronized patch application.

ClearView [24], LUCOS [6] and Band-aid patching [28] present dynamic software updating mechanisms that rely on virtualization. LUCOS relies on page-table modifications to overwrite function prologues while Band-aid patching and ClearView use a DBT. DynSec protects applications in a user-space sandbox and dynamically patches services at instruction level granularity.

## 6 Conclusion

Binary rewriting systems have allowed the construction of sandboxes for arbitrary binaries, i.e., without requiring a-priori knowledge of the binary translator. Such systems assure an application or system's integrity as the sandbox blocks any attempt to execute illegal code. The approach described here combines the sandbox with a dynamic on-the-fly updating system to rapidly install patches without requiring termination or restart of an application or service.

DynSec offers an interesting approach to protect applications from unknown software attacks using a sandbox to protect the integrity of the application. In addition, a dynamic update mechanism patches the application whenever a new patch is available to secure the executing process from software attacks until the service can be safely restarted to increase the availability.

## Acknowledgments

We would like to thank Kevin Borgolte, Albert Noll, and Lenx Tao Wei for their comments on an earlier draft of the paper. We also thank the anonymous reviewers for their valuable comments and suggestions. During the part of this work that was conducted at Berkeley, Mathias Payer was supported through the DARPA award HR0011-12-2-005.

## References

[1] ALTEKAR, G., BAGRAK, I., BURSTEIN, P., AND SCHULTZ, A. Opus: Online patches and updates for security. In *SSYM'05: In 14th USENIX Security Symp.*

[2] ARGYROUDIS, P. CVE-2009-3586: CoreHTTP web server off-by-one buffer overflow vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3586` (2009).

[3] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: automatic rebootless kernel updates. In *EuroSys'09*.

[4] BERNAT, A. R., AND MILLER, B. P. Structured binary editing with a cfg transformation algebra. *WCRE'12: Working Conf. on Reverse Engineering*.

[5] BRUMLEY, D., POOSANKAM, P., SONG, D. X., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE S&P'08*.

[6] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live updating operating systems using virtualization. In *VEE'06*.

[7] CHEN, H., YU, J., CHEN, R., ZANG, B., AND CHUNG YEW, P. Polus: A powerful live updating system. In *ICSE'07*.

[8] CONOLE, A. CoreHTTP arbitrary command execution vulnerability. `http://www.exploit-db.com/exploits/10610/` (2009).

[9] DULLIEN, T., AND ROLLES, R. Graph-based comparison of executable objects. In *SYMP'05*.

[10] FLAKE, H. Structural comparison of executable objects. In *DIMVA'04*.

[11] FORD, B., AND COX, R. Vx32: lightweight user-level sandboxing on the x86. In *USENIX ATC'08*.

[12] GAO, D., REITER, M. K., AND SONG, D. BinHunt: Automatically finding semantic differences in binary programs. In *Proc. 4th Int. Conf. on Information Systems Security*.

[13] HICKS, M., MOORE, J. T., AND NETTLES, S. Dynamic software updating. In *PLDI'01*.

[14] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential slicing: Identifying causal execution differences for security applications. In *IEEE S&P'11*.

[15] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *SSYM'02*.

[16] 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*.

[17] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Usenix ATC'09*.

[18] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *SSYM'06*.

[19] NEAMTIU, I., AND HICKS, M. Safe and timely updates to multi-threaded programs. In *PLDI'09*.

[20] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for c. In *PLDI6*.

[21] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[22] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11*.

[23] PAYER, M., HARTMANN, T., AND GROSS, T. R. Safe loading - a foundation for secure execution of untrusted programs. In *S&P'12: Proc. Int'l Symp. on Security and Privacy* (2012).

[24] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically patching errors in deployed software. In *SOSP'09*.

[25] POPOVICI, A., GROSS, T., AND ALONSO, G. Dynamic weaving for aspect-oriented programming. In *AOSD'02*.

[26] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. *ACSAC'02*.

[27] SEGAL, M. E., AND FRIEDER, O. Dynamic program updating: a software maintenance technique for minimizing software downtime. *Journal of Software Maintenance'89*.

[28] SIDIROGLOU, S., IOANNIDIS, S., AND KEROMYTIS, A. D. Band-aid patching. In *HotDep'07: Hot Topics in System Dependability*.

[29] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis mutandis: Safe and predictable dynamic software updating. *TOPLAS'07*.

[30] VADE79/V9. CVE-2007-4060: CoreHTTPD 0.5.3alpha (httpd) remote buffer overflow exploit. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4060` (2007).

[31] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP'93*.

[32] YAUL, F. CoreHTTP. `http://corehttp.sourceforge.net/` (2009).

[33] 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 S&P'09*.