libdetox: A Framework for Online Program Transformation

Mathias Payer
mathias.payer@nebelwelt.net
Purdue University

Abstract

Software is commonly available in binary form. Yet, the consumer would often like to gather information about the application, e.g., what functionality is available and needed or what security mechanisms are active. In secure environments, the code must also be hardened against attacks. So far, existing binary analysis and translation mechanisms are often ad-hoc and only target one aspect of the problem.

We propose libdetox, a principled framework for continuous binary analysis and instrumentation. Our framework builds on an efficient binary translator and a trusted program loader to enable the collection of vast information which is later used for binary hardening. We present several runtime monitors such as a shadow stack, control-flow integrity, system call monitor, or on-the-fly patch application.

Introduction

Software often exists only in binary form (e.g., COTS software or legacy code), prohibiting code consumers from analyzing, modifying, or hardening the binary. Binary analysis recovers some of the information that is lost during the compilation process and enables debugging or further code transformation on the consumer side. Leveraging the recovered information, code transformations can extend or enrich the original binary to, e.g., add new functionality, remove existing functionality, or harden the binary against attacks.

Binary translation enables principled modification of binary code. All binary translation mechanisms must support the non-trivial identity translation where the functionality of the binary remains unchanged. Building on identity translation, transformations can improve the code in different ways, e.g., for profiling, debugging, functionality extension, or hardening. Binary hardening increases the resilience of an application against different attack vectors, thereby improving the security of the system. Binary defense mechanisms either rely on runtime monitors to detect attacks or "patch" code to mitigate vulnerabilities. Runtime monitors, while detecting the attack, still allow denial of service attacks as compromised execution states cannot be recovered (e.g., a defense that detects code pointer corruption cannot recover the "correct" value as other data of the application may be corrupted as well).

Two fundamental approaches exist for binary analysis and transformation: static and dynamic. Both techniques come with trade-offs. Dynamic techniques profit from concrete values and allow on the fly code discovery but struggle with coverage as only executed code is analyzed. Static techniques struggle with accuracy due to, e.g., aliasing but achieve high coverage. The overhead for instrumented programs is slightly higher for dynamic binary rewriting due to the translation and analysis overhead in addition to the instrumentation overhead. As a rule of thumb, static techniques work well for complex analyses that can tolerate some imprecision due to aliasing while dynamic techniques achieve higher precision at slightly higher runtime cost. Combining the advantages of both, static-dynamic approaches offload complex analyses to a static component while increasing the precision dynamically.

Focusing on the dynamic component for now, we leveraged dynamic binary translation to enable more precise and stronger defense mechanisms that support realistic software like OpenOffice, video players, as well as full sets of standard benchmarks like SPEC CPU2006. Our framework for ongoing dynamic binary program analysis and transformation uses a table-based dynamic binary translator ideally suited for simple analyses. The proposed analyses only require "peephole" information about the last few basic blocks and context information that, e.g., the loader provides through symbol, segment, or module information. We argue that complicated analyses result in prohibitive overhead due to resources needed to construct the intermediate representation (IR) and running the actual analysis.

The binary translator itself must be protected, as the translator executes in the same process and address space as the main application. An adversary may attack not just the translated (and hardened) application but the binary translator and the supporting runtime system [13]. A lean and small runtime system that uses a simple translator without a high-level IR and overly complex analyses decreases this attack surface and can be protected efficiently.

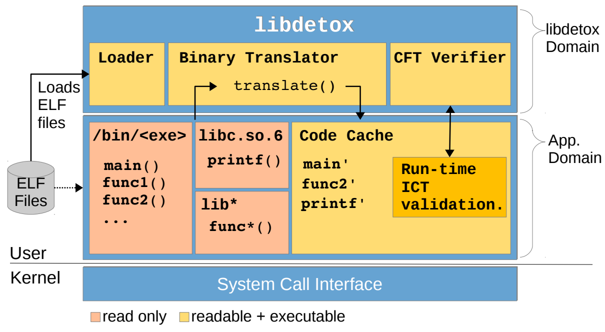

Figure 1: Overview of the lockdown framework, including the dynamic translator, the secure loader, and the control-flow transfer verification module.

Building on the translation framework [7, 6], we designed a set of runtime monitors that harden the executed binary. To protect the control-flow of the application, we enforce stack integrity through a shadow stack [8], protecting against return-oriented programming attacks [12, 11] and enforce control-flow integrity [1] for binaries at runtime [3]. We introduce a trusted loader [10] that protects the loader (and its data structures) from an adversary. A system call monitor enforces a policy on the level of individual per-process system calls [8] and injects a file metadata monitor to protect POSIX programs against the inherent time-of-check-to-time-of-use attacks when working with file names to file descriptor mapping [9].

Our contributions are (i) a framework for continuous program analysis and transformation at low overhead, (ii) a description of a set of program analyses, and (iii) a set of defense mechanisms based on runtime monitors and code transformations that run on this platform.

Binary Translation

Dynamic binary translation is often falsely associated with high overhead. Instead of translating machine code to a high-level IR, our open-source dynamic binary translator uses ISA specific translation tables. A table-based translator limits the achievable instrumentation complexity but results in negligible translation cost [6, 7]. Each decoded instruction is mapped to a function that abstracts the translation and, for same-ISA translation, most instructions can be copied verbatim. Control-flow changing instructions need special care with direct control-flow transfers being adjusted accordingly and indirect control-flow transfers being replaced with a lookup in a data-structure that maps between translated and untranslated code.

The current prototype implementation, shown in Figure 1, targets x86, other ISAs can be supported by adjusting the core translator (about 100 lines of code for x86) and generating the corresponding translation tables. The binary translation framework is implemented in about 20 kLoC. The binary translator uses 13 kLoC whereas the translation tables are about 5 kLoC, the trusted loader replacement of the standard loader uses another 4 kLoC, and the remaining LoC are used for the different code transformations and defenses.

Dynamic Binary Analysis

The lack of a high-level IR may at first seem restrictive, yet our framework allows a wide spectrum of analyses. Just by tracking translated instructions, we recover all executed basic blocks currently in the code cache. An execution-guided analysis distinguishes code from data as only code will be translated, recovering a precise control-flow graph (for executed code paths). In addition, with a simple counter we detect hot code paths, with several optimizations leveraging these counts. On the interface to the operating system we observe all executed system calls, their parameters, and the corresponding context and call stack of the application [8]. This information is the baseline for a set of runtime monitors that we developed. Other analyses are possible, depending and driven by the requirements of the runtime monitor or defense.

Runtime Monitors

Leveraging our framework, we have developed a set of runtime monitors that protect applications against control-flow hijack attacks and (coarse grained) data-only attacks, namely a shadow stack [8], a forward-edge CFI mechanism [3], a system call policy framework [8], and a file authentication mechanism [9].

Return-oriented programming is a common attack vector where an adversary modifies the return address to point to so-called gadgets, short sequences of code, often allowing arbitrary code execution. As we have shown in our control-flow bending work [2], CFI without stack integrity can be mitigated. A shadow stack leverages the relationship between function calls and returns and builds a secondary stack of “expected” return targets. When the function returns, observed and expected return addresses are compared, terminating the application if they mismatch. In our framework, the shadow stack reduces the overhead of binary translation with the side effect of enforcing stack integrity. We translate call instructions into pushes of the current instruction pointer to the regular stack and to the shadow stack, a second push of the translated address to the shadow stack, and a branch to the translated target. Return instructions are then translated into a comparison and a branch.

For forward branches (e.g., indirect function calls), our binary-only CFI policy leverages information from the trusted loader [10] to enforce a precise CFI policy based on imported and exported symbols [3]. Source-based CFI mechanism rely on function prototypes to increase the precision of the protection. Precise function type recovery in binaries is a challenging (and currently unsolved) problem. We leverage the context of functions to increase the CFI precision. Instead of, e.g., allowing all functions as targets as other binary-CFI mechanisms do [14, 15], we use the set of imported and exported functions to further narrow the set of valid targets to few exported functions available in a library. As we show in a case study [3] on nginx, the increased precision suffices to mitigate control-flow hijack attacks while weaker policies fail. As our performance evaluation shows, the majority of programs results in single digit overhead with an average overhead of below 20%.

In addition to these control-flow guards we have also developed a (i) coarse-grained system call authorization mechanism that validates individual system calls depending on the arguments and the context (call stack) of the application and (ii) a file authentication mechanism that protects against POSIX TOCTTOU race conditions when an application is mapping files to file descriptors [9].

Code Transformation

Code transformations allow modification of the code before, during, or after it is executed. We have developed a mechanism for on-the-fly application of security patches [4]. In a case study on the Apache web server, we found that 45 out of the 49 patches do not change data structures and our system can patch the code at runtime without restarting the server, permanently closing the vulnerability [5]. Our system dynamically replaces the code of the application and flushes any translated code blocks. Both the patched and the unpatched code then go through the same hardening process to further enforce stack integrity and CFI. While the current system allows hot patching of security updates, we identify code removal as a challenging extension of our work to reduce the total attack surface. If only a small subset of code is executable (i.e., not all features of the original binary are available), then an adversary has only access to a further restricted set of targets.

Conclusion

Continuous program transformation and binary hardening allows modification of binaries post-compilation and post-distribution. Binary rewriting has many different applications, e.g., gathering information for forensic analysis, debugging, or binary hardening to protect applications against unknown or future attack vectors. Existing binary analysis tools are often limited and ad-hoc. We have developed libdetox, a framework for dynamic binary analysis that leverages a simple, table-based binary translator to achieve low overhead. Our framework supports several binary analyses, from CFG recovery to function analysis and calling context. Building on our translation framework and the gathered information we design several binary hardening mechanisms to, among others, enforce stack integrity, control-flow integrity, and allow on-the-fly security updates.

The source code of the libdetox framework is available at `https://github.com/HexHive/libdetox`.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS: ACM Conf. on Computer and Communication Security*, 2005.

[2] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *SEC: USENIX Security Symposium*, 2015.

[3] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *DIMVA'15: 12th Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, 2015.

[4] M. Payer, B. Bluntschli, and T. R. Gross. DynSec: On-the-fly code rewriting and repair. In *HotSWUp'13: Workshop on Hot Topics in Software Upgrades*, 2013.

[5] M. Payer and T. Gross. Hot-Patching a Web Server: a Case Study of ASAP Code Repair. In *PST'13: Proc. Conf. on Privacy, Security, and Trust*, 2013.

[6] M. Payer and T. R. Gross. Requirements for Fast Binary Translation. In *AMAS-BT'09: 2nd Workshop on Arch. and Microarch. Support for Binary Translation*, 2009.

[7] M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *SYSTOR'10: Proc. 3rd Annual Haifa Experimental Systems Conf.*, 2010.

[8] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments*, 2011.

[9] M. Payer and T. R. Gross. Protecting Applications Against TOCTTOU Races by User-Space Caching of File Metadata. In *VEE'12: Proc. 8th Int'l Conf. Virtual Execution Environments*, 2012.

[10] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *S&P'12: Proc. Int'l Symp. on Security and Privacy*, 2012.

[11] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *SEC: USENIX Security Symposium*, 2011.

[12] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS: ACM Conf. on Computer and Communication Security*, 2007.

[13] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski. Exploiting and Protecting Dynamic Code Generation. In *NDSS: Network and Distributed System Security Symposium*, 2015.

[14] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Oakland: IEEE Symp. on Security and Privacy*, 2013.

[15] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *SEC: USENIX Security Symposium*, 2013.