Enclosures: language-based restriction of untrusted libraries Extended Abstract

Adrien Ghosn
EPFL, Switzerland

Marios Kogias
EPFL, Switzerland

Mathias Payer
EPFL, Switzerland

James R. Larus
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

1. Motivation

Programming has changed; programming languages have not. Modern software development has embraced abstraction and reusable software components. Programs build on open-source libraries (aka packages) that offer diverse, tested functionality and increase programmer productivity. In the limit, an application becomes a collection of packages orchestrated by application-specific code. Modern languages have evolved extensive public libraries and tools to publish, find, download, use, and update public packages, for example, Python modules [3], Golang packages [16], Ruby gems [4], and Rust crates [5].

Although languages support the use of packages, few, if any, provide a strong mechanism to deal with their inherent insecurity and fragility. Packages come with challenges: (1) they have no formal specification of what they do (or do not do); (2) their developer is typically unknown, thus untrusted; (3) they lack traceable dependency management – a package can import unknown, untrusted dependencies; and (4) most important, programs run in a single trust domain that does not segregate code or data from different packages. In general, a developer's trust in a public package often seems to be based on its popularity or a shallow code review. Careful inspection is both impractical, since importing a single package may incorporate hundreds or thousands of transitively dependent packages [29, 30], or infeasible, as a package's code may change frequently. As a result, an application becomes a patchwork of code from untrusted and unverified sources.

Malevolent individuals have been quick to exploit the opportunity to insert malicious code in a popular package [9, 10, 14, 44], via IDEs [35], or to substitute modified clones [11–13, 25]. These attacks are easy to implement and provide unimpeded access to hundreds, if not thousands, of applications, as they use conventional programming constructs to insert code that steals private information or opens backdoors. Examples include malicious Python packages that stole SSH and GPG keys [10, 13] from the local file system. More generally, even legitimate third-party libraries may implement undocumented functionality that operates outside of its advertised scope. For example, the Facebook iOS SDK, intended to identify users, shared device information with Facebook without user consent [43].

2. Limitations of the State of the Art

Although software isolation is a very long-studied topic in the systems, programming languages, and security communities [7, 8, 15, 17, 18, 20, 22, 22–24, 26, 28, 31, 32, 34, 37, 39, 40], previous approaches do not offer a clear solution for package isolation and security: (1) pure systems approaches introduce new low-level abstractions that may not match programming language requirements [7, 15, 22, 26, 31, 32, 40] or may require application refactoring [8, 23, 24]; (2) pure language approaches, *e.g.,* Rust or Javascript isolates, are limited to a single language and thus only apply to code written in this language; (3) mixed approaches, *e.g.,* Erim [37], Hodor [17], and Glamdring [20], do not take into account deep dependencies in the transitive dependency graph and the consequent need for complex access rights. These mixed systems offer the same isolation guarantees across a program, which requires a developer to manually modify import dependencies to achieve their desired isolation granularity, a process that can be cumbersome or infeasible.

3. Key Insights

Packages, while the source of security and fragility problems, have characteristics that make a solution possible. They consist of code and data usually written to be able to run as part of any program, which means they must have clearly defined entry points, no dependencies on the program's environment, and must bring along their dependencies. Languages lack a mechanism for taking a package and executing it and its dependencies in a restricted environment where the package cannot access the state of the entire program or the system on which it is executing.

We propose a new programming language construct that provides a developer with fine-grain control over the resources that a package can access, even in modern software with complex dependency graphs. The abstraction underlying this construct is language-independent, so it can be incorporated into most languages. Its implementation in these languages needs support from a hardware isolation mechanism that is not tied to a single language implementation since programs are typically constructed from components written in several languages. Fortunately, architectural features provide trustworthy, fine-grain, hardware-based mechanisms that can enforce access control within a virtual address space [1, 6, 19, 36, 41]. These

features are low level and difficult to use, so they have not been widely accepted. Moreover, without agreement on how to use them, they prevent language inter-operability.

This paper introduces *enclosure*, a programing construct that binds a memory view and system calls allowed to a closure, restricting its access to a program's resources according to user-defined policies. The memory view defines the code and data accessible by the closure, and is automatically derived from the closure's package dependencies. User-defined policies can restrict the closure's memory view or extend it, by selectively enabling read, write, or execute access rights on packages. They can also selectively authorize system calls.

Enclosure policies are enforced at run time by LITTERBOX, a language-independent framework that uses hardware mechanisms to provide strong uniform isolation guarantees, even for packages written in unsafe languages. LITTERBOX exposes a high-level API that abstracts the language-specific program resources and is thus reusable across programming languages. LITTERBOX can utilize different hardware technologies for isolation and hides the intricacies of hardware.

4. Main Artifacts

Enclosures consist of two separate parts: (1) *frontend* language-specific support, implemented by a language's compiler and runtime, and (2) the *backend* responsible for using hardware to enforce the closure's memory view and filter system calls (see Figure 3).

Language support for *enclosures* requires programming language's syntax, compiler, and runtime extensions. The syntax is extended to declare *enclosures* and specify user-defined policies. The compiler identifies the closure's package dependencies and relies on the linker to segregate their code and data on separate memory pages. The runtime manages dynamically allocated objects on a per-package basis on separate memory segments, called arenas. Creation, modification, and transitions to *enclosures* restrictive execution environments are managed by calling the backend that enforces isolation based on hardware mechanisms.

We implemented a full-fledged *enclosure* extension for Go, and a prototype one for Python, both based on the LITTERBOX backend.

The LITTERBOX backend exposes a language-independent small API to manage *enclosures*, hides low-level intricacies and supports different hardware isolation mechanisms (Intel VT-x and Intel MPK).

We evaluate our Go *enclosure* extension based on LITTERBOX. The evaluation uses popular Github Go packages and proposes to benchmark the performance of small applications, derived from each package's "hello world" sample code, to determine the worst-case performance overheads of LITTERBOX. In these applications, *enclosures* are used, in vastly different ways, to safely leverage the *unmodified* public package. In one example, we isolate `FastHTTP` [38], a popular library with 370K lines of code from over 100 contributors so that it cannot access the memory outside of the *enclosure*, and can only perform network system calls. In another example, we isolate an untrusted, user-specific webserver handler from the main HTTP stack. We also leverage *enclosures* to safely expose senstive data to the popular `Bild` [33] image processing library, while preventing modifications or leakage (*e.g.,* via system calls).

To understand our performance results, we perform a combination of microbenchmarks that exercise the low-level hardware mechanisms (changes in protection keys for MPK, transitions with VT-x), combined with the performance evaluation of our macrobenchmarks. The results show that VT-x has lower overheads for memory-bound workloads than MPK. System call interposition with VT-x is comparable in cost to the use of a protected container layer such as Dune [7] or gVisor/Sentry [42], but more expensive than the eBPF patched [27] seccomp [2, 21] approach taken by our MPK implementation.

5. Key Results and Contributions

This paper makes the following contributions:

- The *enclosure* programming construct, a simple way to safely execute closures leveraging untrusted packages with deep dependency graphs according to fine-grained user-defined policies.
- LITTERBOX, a language-independent framework that enforces *enclosure*-defined policies with strong hardware isolation mechanisms. LITTERBOX currently supports either Intel VT-x (and its general-purpose extended page tables) or the emerging, specialized Intel Memory Protection Keys (MPK).
- The implementation of *enclosures* based on LITTERBOX for the Go language, demonstrating low overheads for real-world applications, and for Python, exhibiting support for highly dynamic languages.

6. Why ASPLOS

This paper spans all three aspects of ASPLOS. We address a significant deficiency of programming languages with a language-agnostic system that uses processor mechanisms to extend the operating system address space model with isolation, which can be used to protect against threats from software packages.

7. Citation for Most Influential Paper Award

This paper describes the original proposal to use enclosures (which we now understand as closures with restricted memory and system privileges) as a way to express trust in software components used to assemble applications.

References

[1] Intel SGX - software guard extensions programming references. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[2] Linux seccomp. https://code.google.com/archive/p/seccompsandbox/wikis/overview.wiki, 2020.

[3] Python Packge Index. https://pypi.org/, 2020.

[4] Rubygems stats. https://rubygems.org/stats, 2020.

[5] Rust the cargo book. https://doc.rust-lang.org/cargo/commands/, 2020.

[6] ARM. Arm1136jf-s and arm1136j-s technical reference manual. https://developer.arm.com/documentation/ddi0211/latest/, 2020.

[7] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2008.

[9] Catalin Cimpanu. Somebody tried to hide a backdoor in a popular javascript npm package. https://www.bleepingcomputer.com/news/security/somebody-tried-to-hide-a-backdoor-in-a-popular-javascript-npm-package/, 2018.

[10] Catalin Cimpanu. Backdoored Python Library Caught Stealing SSH Credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, 2019.

[11] Catalin Cimpanu. Malicious Python libraries targeting Linux servers removed from PyPi. https://www.zdnet.com/article/malicious-python-libraries-targeting-linux-servers-removed-from-pypi/, 2019.

[12] Catalin Cimpanu. Twelve malicious Python libraries found and removed from PyPi. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/, 2019.

[13] Catalin Cimpanu. Two malicious Python libraries caught stealing SSH and GPG keys. https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/, 2019.

[14] Catalin Cimpanu. Malicious npm packages caught installing remote access trojans. https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/, 2020.

[15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 191–206, 2015.

[16] Google. Golang add dependencies to the module and install them. https://golang.org/cmd/go/#hdr-Add_dependencies_to_current_module_and_install_them, 2020.

[17] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 489–504, 2019.

[18] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM Conference on Computer and Communications Security*, pages 393–405, 2016.

[19] Intel. Intel®64 and IA-32 Architectures Software Developer's Manual, 2020.

[20] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 285–298, 2017.

[21] Linux. SecComp Load Filter. https://man7.org/linux/man-pages/man3/seccomp_load.3.html, 2020.

[22] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–64, 2016.

[23] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the 2012 Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[24] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *ACM Conference on Computer and Communications Security*, pages 1607–1619, 2015.

[25] Lukas Martini. Fake version of dateutil and jellyfish. https://github.com/dateutil/dateutil/issues/984, 2019.

[26] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[27] Michael Sammler. seccom: Add pkru into seccomp data. https://marc.info/?l=linux-api&m=154039581615478&w=2, 2018.

[28] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 269–284, 2019.

[29] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *ACM Conference on Computer and Communications Security*, pages 736–747, 2012.

[30] Nikola Đuza. JavaScript Growing Pains: From 0 to 13,000 Dependencies. https://blog.appsignal.com/2020/05/14/javascript-growing-pains-from-0-to-13000-dependencies.html, 2020.

[31] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *ACM Conference on Computer and Communications Security*, pages 477–487, 2009.

[32] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] Anthony N. Simon. Bild: A collection of parallel image processing algorithms in pure Go. https://github.com/anthonynsimon/bild, 2020.

[34] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *ACM SIGOPS European Workshop*, pages 102–107, 2002.

[35] Trend Micro. The XCSSET Malware: Inserts Malicious Code Into Xcode Projects, Performs UXSS Backdoor Planting in Safari, and Leverages Two Zero-day Exploits. https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf, 2020.

[36] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[37] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, pages 1221–1238, 2019.

[38] Aliaksandr Valialkin. FastHTTP: Fast HTTP implementation for Go. https://github.com/valyala/fasthttp, 2020.

[39] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[40] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, pages 380–395, 2010.

[41] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[42] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019.

[43] Eric S. Yuan. Zoom's Use of Facebook's SDK in iOS Client. https://blog.zoom.us/zoom-use-of-facebook-sdk-in-ios-client/, 2020.

[44] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, pages 995–1010, 2019.