Towards Practical Application-level Support for Privilege Separation

Nik Sultana
Illinois Institute of Technology
USA

Henry Zhu
UIUC
USA

Ke Zhong
University of Pennsylvania
USA

Zhilei Zheng
University of Pennsylvania
USA

Ruijie Mao
University of Pennsylvania
USA

Digvijaysinh Chauhan
University of Pennsylvania
USA

Stephen Carrasquillo
University of Pennsylvania
USA

Junyong Zhao
University of Arizona
USA

Lei Shi
University of Pennsylvania
USA

Nikos Vasilakis
Brown University & MIT
USA

Boon Thau Loo
University of Pennsylvania
USA

ABSTRACT

Privilege separation (privsep) is an effective technique for improving software's security, but privsep involves decomposing software into components and assigning them different privileges. This is often laborious and error-prone. This paper contributes the following for applying privsep to C software: (1) a portable, lightweight, and distributed runtime library that abstracts externally-enforced compartment isolation; (2) an abstract compartmentalization model of software for reasoning about privsep; and (3) a privsep-aware Clang-based tool for code analysis and semi-automatic software transformation to use the runtime library. The evaluation spans 19 compartmentalizations of third-party software and examines: **Security:** 4 CVEs in widely-used software were rendered unexploitable; **Approximate Effort Saving:** on average, the synthesis-to-annotation code ratio was greater than 11.9 (i.e., 10× lines of code were generated for each annotation); and **Overhead:** execution-time overhead was less than 2%, and memory overhead was linear in the number of compartments.

ACM Reference Format:
Nik Sultana, Henry Zhu, Ke Zhong, Zhilei Zheng, Ruijie Mao, Digvijaysinh Chauhan, Stephen Carrasquillo, Junyong Zhao, Lei Shi, Nikos Vasilakis, and Boon Thau Loo. 2022. Towards Practical Application-level Support for Privilege Separation. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3564625.3564664

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12…$15.00
https://doi.org/10.1145/3564625.3564664

1 INTRODUCTION

Privilege separation (privsep) is an effective technique for software security that stymied viable exploits [17] but it needs to be tailored for each piece of software. It involves structuring programs to run as multiple processes and assigning to each process the least amount of access it needs to function, thus instantiating the principle of least privilege [56]. Privsep refines the isolation granularity of applications into a cooperating set of processes, and leverages OS-provided isolation—including application firewalls and containers [48]—to protect those processes from each other.

By separating subsets of a program and assigning them different privileges, privsep enables the containment of vulnerabilities in buggy or compromised parts of the program and its dependencies. Thus an exploitable parser library used by a daemon will not readily give an adversary access on par with that daemon. The risk from dependencies includes supply-chain attacks [41]. In these attacks, exploits are deliberately inserted in upstream codebases, sometimes in deeply-nested dependencies that users are unaware of [61].

Because of the restructuring it requires, applying privsep is laborious and error-prone. There is a lack of privsep-supporting techniques and tools that could enable wider use of this security approach. Currently programmers apply privsep in an ad hoc way, without a framework for managing separation over code and data.

Privsep is applied to *widely-used* software but—because of the effort required to apply privsep—it is not *widely applied* to many types of software, including games, system administration utilities, and office productivity tools. The best-known examples of privsep's use include servers such as OpenSSH [54] and Apache httpd—where the network-facing part of the system is separated from the core server—and clients such as most web browsers [22, 32, 46] to isolate each tab from each other and the rest of the system.

This paper introduces the Pitchfork approach for privsep. This approach is designed to work with a large class of software with minimal impact on their portability and maintainability, and to provide compile-time and run-time support for privsep. Pitchfork

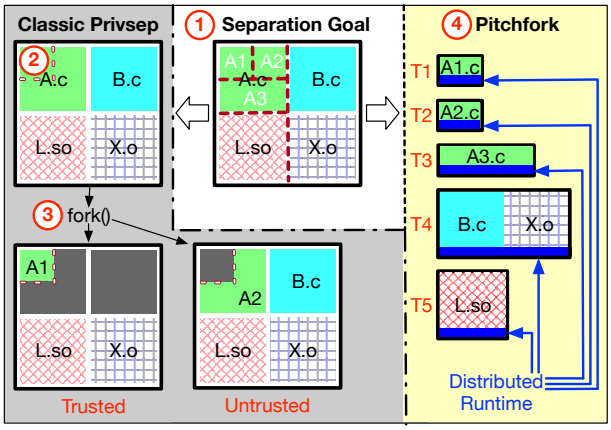


Figure 1: Privsep involves splitting a program and restricting the access of its derivative parts. This figure compares classic Privsep with Pitchfork and is explained in §1. Classic privsep derives Trusted and Untrusted parts, and in this illustration A3 has to be combined with A2 or A1 as a result.

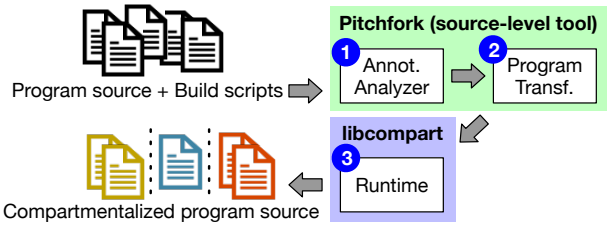


Figure 2: Stages of processing, which have all been implemented in the prototype toolchain: ❶ An annotated C program is analyzed by Pitchfork, a Clang-based tool, to determine cross-compartment synchronization needs. ❷ The annotated program is transformed to use libcompart by the same tool, saving programmer effort. ❸ libcompart provides useful abstractions for privsep. While portable, libcompart uses a pluggable interface to leverages OS-specific isolation and communication primitives between separated program parts. For increased flexibility, programmers can use libcompart directly instead of going through the Pitchfork source-level tool.

is a stepping stone towards the further generalization and automation of privsep to reduce the burden on developers. One of our realizations in this work is that **privsep is not a single problem but a complex of problems that need a cohesive solution.** A solution to privsep spans program decomposition, concurrency, synchronization, resource discovery and configurability, distributed debugging, and the maintainability of the decomposed program.

Pitchfork generalizes classic privsep as shown in Fig 1. ① A program consists of source files (A.c, B.c) and linked code (X.o and L.so). A *separation goal* (shown as a dotted red line) states a specific structuring of the program into separate processes. ② Classical privsep targets program sources and involves a separation between Trusted and Untrusted parts. ③ Classical privsep isolates the two parts at runtime typically through `fork()` followed by a privilege drop. ④ Pitchfork supports several, user-defined trust levels (T1-T5 in this example), separation of third-party dependencies, separation at compile-time as well as run-time, and provides a runtime to manage the resulting distributed program.

Pitchfork provides an abstract software model that partitions code and data into *compartments*. A compartment is an abstraction that represents a part of the program that is to be separated. It can be instantiated as a separate process or even a separate binary.

Fig. 2 shows the realization of a toolchain based on this model, implemented for the C language. This toolchain consists of (1) libcompart, a runtime library that provides the model's functionality directly to C programmers, and (2) Pitchfork, a tool that analyzes privsep code annotations in a C program and source-to-source transforms the program to use the runtime library, saving the programmer manual effort. The toolchain's implementation was structured to balance flexibility (using the runtime library) with convenience (using the source-level tool). The runtime library can be used independently of the source-level tool for more flexibility, at the cost of some convenience. The source-level tool carries out an unsound but pragmatic analysis, inspired by the approach used by widely-used software analysis [29].

Pitchfork is orthogonal to sandboxing [18, 40, 59] and memory protection [35, 39, 51] techniques, and differs from other security-oriented decomposition techniques. In particular, enclave-oriented program splitting [25, 49] requires specific target hardware and only splits software into two parts, both of which must be co-resident on the same machine. Mapping compartments to hardware-enforced environments such as enclaves is left as future work. Pitchfork's model is inspired by approaches for concurrency [28, 34, 38, 43] but is deliberately restricted to avoid creating opportunities for vulnerabilities in the runtime library's implementation. For example, channels cannot be dynamically created at runtime, and cannot carry other channels. Related work is described further in §10.

The repo at https://gitlab.com/pitchfork-project provides the source code for Pitchfork, libcompart, and their example application to third-party software. This paper contributes: **(1)** An abstract compartmentalization model (§5) for structuring software into disjoint units that are secured separately. This model is used to reason about privsep and systematize different types of memory sharing between program parts. **(2)** A prototype implementation (§8) that consists of: **(2a)** libcompart: A lightweight and portable runtime library for privsep (§6). This library can be used by new or existing software to apply privsep according to Pitchfork's abstract model. **(2b)** Pitchfork: A Clang-based source-level analysis and transformation tool that uses source-level annotations to automatically split C programs (§7). **(3)** An evaluation of the runtime library and the source-level tool implemented in the Pitchfork prototype. This evaluation includes: general applicability of Pitchfork to existing software, security gains, and execution overhead. It involves a variety of widely-used open-source software including games, system administration utilities, and productivity tools across three operating systems (§9).

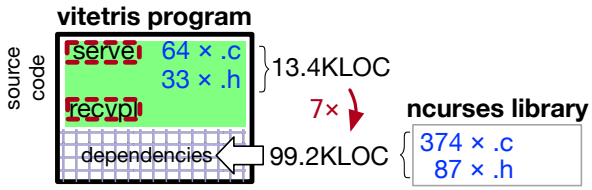


Figure 3: Our *separation goal* is to split Vitetris into: `serve` (creates a listening port), `recvpl` (receives and parses player data), and `main`—a special compartment in Pitchfork that contains everything else. In addition we could isolate dependencies such as the libncurses library whose source is over 7× larger than Vitetris. In §2.1 and §C we describe dependency isolation for the Evince viewer and for netbpm's tifftopnm tool respectively.

2 MOTIVATION AND BACKGROUND

This section describes typical software security concerns, *separation goals* that address those concerns, and typical challenges faced when applying privsep to meet those goals.

Threat Model. We assume that an adversary may control all inputs to a program, including inputs from files and the network. An adversary may also control third-party libraries used by the program. We assume that the adversary does *not* control the compiler toolchain, host, and OS on which the compartmentalized software is being run.

2.1 Examples

Privsep helps contain unknown vulnerabilities in software, and is usually applied as a precautionary measure based on separation goals. Each goal provides a—typically informal—reason why a particular software split would improve security. This section provides separation goals for two example programs.[1] These include an example of compartmentalizing a dependency that we do not modify, and that could be available to us only in compiled form.

Example 1: Vitetris. This is a portable implementation of Tetris that is small but rich in features [24]. Fig. 3 illustrates our separation goal. Vitetris has zero known vulnerabilities at this time [7], and the size of its source code is relatively small at 13.4KLOC. But it relies on several sizeable libraries which may potentially have vulnerabilities [47]. For example, its libncurses dependency has had several known vulnerabilities [6]. Further, Vitetris might have exploitable bugs in its rich set of features that include support for different input devices, different output encodings, and IP-based or UNIX domain socket-based two-player games.

Example 2: Evince. Even if a widely-used program does not use privsep, it is still likely to benefit from it. Evince [8] is a document viewer for GNOME that supports a wide variety of document formats. On top of its 83.8KLOC, Evince has several dependencies that supply its UI and format-support features. Among them is libspectre which provides support for viewing PostScript files. It does not have any known vulnerabilities, but its main dependency, GhostScript, has had past vulnerabilities [10] and provides an embedded interpreter which is often a security risk [37]. Our separation goal for Evince is to isolate libspectre. In this example we do not modify libspectre, and the compartmentalization is done entirely within Evince. In practice, this compartmentalization enables applications to use closed-source third-party dependencies more securely.

2.2 Separation heuristics and implementation

Separation goals are typically based on heuristics for separating parts of an application that differ in their trustworthiness. We reduce these heuristics to three patterns of separation. They involve isolating: **1)** Parts of the program that only access specific objects, *e.g.*, the data that needs processing. **2)** Libraries and other dependencies. **3)** Cross-domain interfaces, such as network-facing code. These heuristics capture common-sense patterns of separation that are applied in practice: for example, when isolating part of the system that has a history of containing bugs, or if it handles data from untrusted sources.

Though these heuristics are simple, implementing them is usually complex: **a)** parts of the program might need to be cleaved off to form new programs, and the build scripts might need modification; **b)** language-level reasoning is required to safely separate the program, to handle aliasing, variadic functions, etc; **c)** state needs to be synchronized between programs to preserve the original program's behavior; **d)** non-serializable state, such as abstract types and OS resource handles, needs to be shared across programs; **e)** the resulting distributed system of programs needs to be coordinated and monitored for partial failures.

This all needs to be done in a least-invasive, maximally-maintainable way that preserves the original program's portability, with the lowest performance overhead, and without introducing new bugs or vulnerabilities. While demanding, this pattern of needs is faced every time privsep is applied to a new application. This observation provides the basis for the support that Pitchfork, and the prior work it builds on, seeks to make privsep more widely-applicable.

2.3 Survey of C code-bases

To evaluate the characteristics of C code-bases—their size and C variant—to provide context for the code-bases used in the evaluation of this research, we surveyed a significant collection of real-world C software. This section summarizes the survey results, and Appendix G describes the methodology and further result details.

This survey targeted FreeBSD's ports [1] collection, a set of third-party applications that have been ported to run on the base system. This consists of 29156 projects in total, taking up 164GB. We wrote a script to analyze this to measure the size of C codebases and which variants of C they use.[2]

The results of this survey are that the median size was 24 KLOC and average size was 121 KLOC. Most projects did not specify which C version to compile them against. From the projects that *did*

[1] This paper describes 19 compartmentalizations of various kinds of open-source software. Many people volunteered their time to develop that software. By including their software in our study we do not wish to single it out for criticism of its potential or reported lack of security. Rather, we are grateful to be able to study real examples through open-source projects in our search for software-architecture patterns that can be targeted by a scalable form of privsep to improve general software security.

[2] The script and data related to this survey are included with the artifact release.

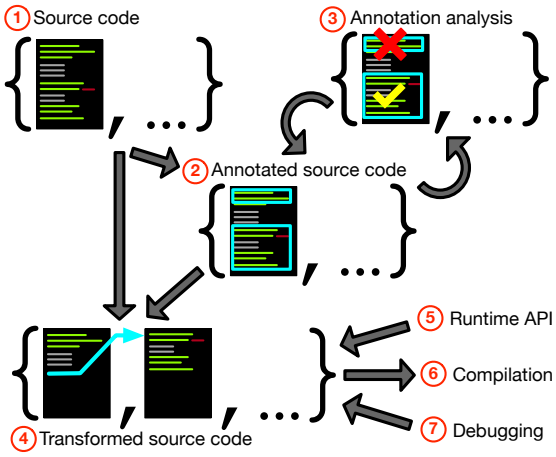


Figure 4: Pitchfork workflow across a codebase (set of files).

specify which C version to use, c89 was the most popular version. The surveyed projects rely on different versions of C including c11, gnu11, c99, and gnu99. The properties of the chosen use-cases for this paper are described in §H.

3 PITCHFORK OVERVIEW

This section outlines how Pitchfork works before the rest of the paper goes into more detail and evaluates the prototype.

Pitchfork provides compile-time and runtime support to programmers to structure a program into *compartments* as independently-executing but cooperating units. Compartments are an abstraction for a subset of the original program's code and data. They are given subsets of the privileges of the original, monolithic application, but by acting together compartments produce behavior very similar to that of the original program.

Fig. 4 shows Pitchfork's workflow. We start by forming *separation goals* based on informal security objectives, similar to the examples in §2. These goals are captured in the ① program source code either through compartmentalization *annotations*, leading to ②, or through directly using libcompart (the Pitchfork runtime API) to reach ④. Using the API directly would forego automated analysis and transformation but affords the programmer more flexibility.

Examples of both choices will be given. We return to ④ soon, but we start with ②: the source code being annotated.

Annotations serve as in-program separation documentation that can be ignored to compile the original program, or processed by Pitchfork to produce a compartmentalized program. In addition to segments of code, as shown above, programmers can annotate regions of code, including (type, function, and variable) declarations. The Pitchfork prototype uses C function syntax for annotations. These functions are interpreted by Pitchfork during analysis and transformation, and never reach the C compiler.

③ Annotated source code is transformed into an abstract representation that is used to check compliance with Pitchfork's compartmentalization model (§5). Compliance checking (§7.1) is done to avoid incorrect behavior at runtime—this is like type-checking but at the level of compartments. It involves abstract program reasoning to determine the synchronization requirements between compartments and to ensure that adequate precautions are taken in cross-compartment communication.

Writing annotations and iterating through invocations of Pitchfork is intended to be less burdensome for the programmer when compared to chasing function definitions to gather information manually while also heeding compliance criteria for compartments to avoid complications at run-time.

④ Source code is transformed to implement the intent captured by the separation goals. This transformation is either done manually or using the Pitchfork to carry out a source-to-source (§7.2) transformation.

⑤ During both manual and automatic translation, the program is modified to use libcompart (§6). This library manages the distributed system derived from the original program, including the communication between its compartments. As will be described in §6, libcompart uses common OS features such as processes and IPC. It can also use OS-specific features through wrappers.

Part of the modification made to Vitetris to use libcompart is shown in Listing 1. This snippet shows the initialization of the system. Registering a segment allows it to be invoked from another compartment. Once `compart_start()` is called (line 165) then no more registrations are possible.

```
156 +#include "vitetris_interface.h"
158
159  int main(int argc, char **argv)
160  {
161 +  compart_check();
162 +  compart_init(NO_COMPARTS, comparts, default_config);
163 +  listen_ext = compart_register_fn(C_NAME_LISTEN_K, &
          ext_listen);
164 +  recvplayer_ext = compart_register_fn(
          C_NAME_RECVPLAYER_K, &ext_recvplayer);
165 +  compart_start(C_NAME_MAIN_K);
166 +
168    setcfgfilename(argv[0]);
169    readoptions();
170    timer_init();
```

Listing 1: libcompart used on Vitetris' main.c.

Failure handling is provided by libcompart (§6) to improve the resilience of the distributed program in case of partial failure. This is done through hooks to which programmers can provide code for reacting to failure.

⑥ Since the Pitchfork source-level tool fully parses C code, it requires knowledge of the compiler parameters to be able to locate header files, use the right conditional compilation branches, etc. Typically such parameters are generated by a build system. To automate this parameter discovery at compile-time, the Pitchfork prototype includes a meta-Makefile that observes the compilation of the original program to extract the necessary information.

⑦ The distributed nature of the compartmentalized program makes it more tedious to debug compared to a monolithic program. We sometimes encountered misbehavior in compartmentalized code

that could not be reproduced in the original program. Often this misbehavior stemmed from different permissions that different parts of the program have when they run as a compartmentalized system. To mitigate this, two compartment-aware debugger prototypes were developed. Both use an extensibility hook in libcompart to run additional code at run-time. The simpler prototype inlines invocations to a debugger interface at specific lines of a distributed program, as hard-coded breakpoints. The more advanced prototype embeds a restricted gdb controller in the program at compile time, and provides a gdb front-end wrapper that allows switching between different gdb controllers running in different compartments. This paper focuses on development support for privilege separation, and the debugging approaches for isolated compartments is discussed in a paper about a project dedicated to that topic [62].

4 EXAMPLE: BEEP

Beep is a 372-line C program that allows users to control the PC speaker from the command line. It is packaged to run on widely-used Linux distributions such as Debian and Ubuntu. It typically runs as superuser with the `SETUID` bit set as it needs to send data to the sound device.

Despite being small and innocuous, a local privilege escalation vulnerability was found in beep [5, 12]. It is a "time of check to time of use" (TOCTTOU) [60] vulnerability and the exploit consists of racing against beep to change which "file" beep is writing to,[3] between TOC and TOU and thus manipulating beep to perform an unintended action as superuser. Specifically if the user directs beep to write to a symbolic link to the speaker device, and then quickly rewrites the link to point to an arbitrary file, beep will write to that file as the super-user.

```
105 if(console_type == BEEP_TYPE_CONSOLE) {
106   pitchfork_start("Privileged");
107   if(ioctl(console_fd, KIOCSOUND, period) < 0) {
108     putchar('\a'); /* Output the only beep we can, in an
            effort to fall back on usefulness */
109     perror("ioctl");
110   }
111   pitchfork_end("Privileged");
112 } else {
113   /* BEEP_TYPE_EVDEV */
114   struct input_event e;
115   e.type = EV_SND;
116   e.code = SND_TONE;
117   e.value = freq;
118   pitchfork_start("Privileged");
119   if(write(console_fd, &e, sizeof(struct input_event)) <
            0) {
120     putchar('\a'); /* See above */
121     perror("write");
122   }
123   pitchfork_end("Privileged");
124 }
```

Listing 2: Annotated beep.c.

[3]Following the UNIX philosophy that virtually everything is a file, devices are represented in Linux's file namespace.

We privsep beep into two: a **Privileged** compartment that makes calls to `write(2)`, `read(2)`, `ioctl(2)`, and the **Main** compartment that executes everything else.

Listing 2 shows the compartmentalized program, with Pitchfork annotations highlighted in orange. Code form the application is enclosed between annotations as on lines 106 and 111. Except for adding annotations, no other changes were made to the program. The Pitchfork source-level tool emitted code for the entire compartmentalization, including the transfer of context between compartments. In addition to transforming code, the Pitchfork tool also checks compartment configuration. The generated C code could then be compiled to produce the compartmentalized binary.

The Pitchfork compartmentalization model is described in §5. The model includes named *compartments* (e.g., **Privileged**) and organizes compartments into one or more *segments*. Listing 2 shows two segments of the same compartment. Naming allows us to coordinate and configure different compartments, and segmentation allows us to coordinate disjoint pieces of code that we wish to place in the same compartment. A compartmentalized program also has an implicit *main compartment* which is necessarily unique.

Turning back to our beep compartmentalization, it prevents the attacker from overwriting a file by i) restricting important library-function calls to take place within a privileged compartment, ii) restricting the privileged compartment's visibility of the file system. Note that in this instance compartmentalization does *not* fix the race condition, but it removes the ability to exploit it to write to arbitrary files as the super-user. This vulnerability is defanged by confining the privileged compartment to only access device files.

5 COMPARTMENT MODEL

This section presents the foundational model for software compartmentalized using Pitchfork. Compartmentalization choices made by programmers (using the runtime library or through the source-level tool) instantiate this model in different ways: how many compartments to use, whether to configure them at compile-time or runtime, which IPC to use between them, whether to generate separate binaries, etc.

Fig. 5a shows Pitchfork's model. It is structured into one or more *domains*, each containing one or more *compartments*, where each compartment contains one or more *segments*. Compartments may communicate with other compartments across domains using IPC. Domains provide execution contexts for compartments, and can be instantiated as hosts, VMs, or containers. By requiring two compartments to be placed in separate domains, a user is allowing stronger separation between the compartments. Consequently, the separation cannot be `fork()`-based, and the IPC used between the two compartments cannot be local—like pipes or UNIX domain sockets.

There are four compartments arranged across three domains in Fig. 5a. Recall that a compartment represents part of the original program's control and data, and confines access to state, functions, or other resources such as files, libraries, and devices.

There are three types of compartments in Pitchfork. The main compartment is special and is always placed in domain0. It contains the program's entry point—the `main()` function in C. Occasionally the main compartment transfers control to *segment* compartments

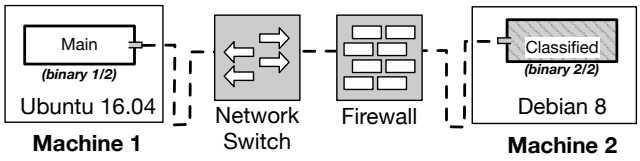


Figure 6: Compartments from Fig. 5c were placed on different machines for increased isolation. Compared to other software compartmentalization approaches, Pitchfork makes it easier to separate compartments in this manner. In this case the machines ran different Linux distributions.

which eventually hand back control. Segment compartments are shown as compart1, compart2, and compart3 in Fig. 5a. These compartments contain an arbitrary number of terminating snippets of code, called *segments*, carved out of the original program. Segments share the memory and privileges of a compartment. Finally, the monitor is a special compartment that is always placed in domain0 together with main. It observes the interactions between the main and segment compartments, and handles exceptional events: communication time-outs or broken communication channels.

Fig. 5b instantiates the model for the tifftopnm compartmentalization that separates a tool from a dependency that has a history of vulnerabilities. This example is detailed in §C. Fig. 5c shows an example reproduced from other research [50, Fig. 2] that separates between classified information and untrusted code. The Pitchfork-annotated code related to this example is in §C.1 which we compartmentalized across two separate machines as shown in Fig. 6.

All communication in Pitchfork is synchronous. The main compartment communicates with all other compartments. Other compartments do not communicate directly with each other but a compartment's segment may call another segment in the same compartment. We call this *short-circuiting* and it relies on knowing in which compartment the currently-executing code is located. Pitchfork's libcompart tracks this information during the program's lifetime.

Limitations. Using a model binds us to a specific pattern of compartmentalization but makes it easier to generalize to a class of privsep applications. The examples in our evaluation (§9) suggest that this class contains practically useful compartmentalizations. This model-based approach to compartmentalization contrasts with the "ad hoc" approach which works for specific cases [54, 57] but by its nature is difficult to generalize, scale, and build tooling for. In this prototype, the number of compartments is static and fixed at compile time—e.g., we cannot have one compartment per user—and compartments cannot be nested, to simplify communication.

6 LIBCOMPART (RUNTIME LIBRARY)

This library implements the features of the Pitchfork model that take place at run-time, including: system initialization, segment registration, inter-compartment communication, and failure detection. The model can be instantiated in different ways (to use different IPCs, or use separate binaries instead of process-forking for compartment) but it presents a uniform API for all instantiations. §A shows part of this API.

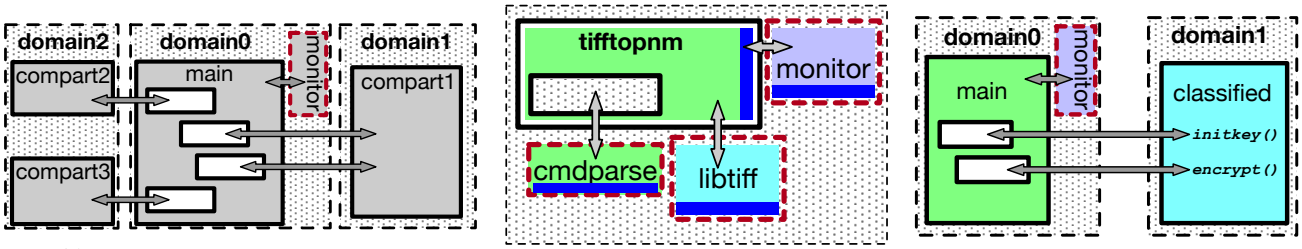
Pitchfork-compartmentalized programs are linked to this runtime library—similar to how C programs typically link to an implementation of the C standard library. The runtime library is in turn linked to other libraries that implement the pluggable components described below, such as wrappers for platform-specific IPC.

The core components in libcompart require very little from the host OS: just that it provides processes and IPC. This pluggable approach helps mitigate the divergence in availability of different abstractions across different OSs [27].

The library provides four key features. **Pluggable communication**: OS-provided IPC are wrapped and presented across a uniform interface shown in §B. libcompart currently has wrappers for anonymous and named pipes, and TCP. **Pluggable de/marshalling**: Externally-provided de/marshalling code is invoked to process the contents of a buffer that libcompart oversees to synchronize compartments. **Monitoring**: The monitor compartment handles exceptions such as communication time-outs or crashed compartments through callback functions that are registered at initialization. **Pluggable sandboxing**: OS-provided isolation mechanisms are wrapped for use on compartments. The Pitchfork implementation currently has wrappers for Linux's SECCOMP [19], FreeBSD's Capsicum [59], and portable POSIX functions such as `chroot()`. In addition to this, compartments can execute in separate containers or VMs, but that is an orchestration feature that the current prototype does not automate.

7 PITCHFORK SOURCE-LEVEL TOOL

This tool is built on Clang [4] and it carries out two tasks: **1)** Parse C and analyze privsep annotations. **2)** Source-to-source translate C code to replace annotations with invocations to libcompart's



(a) Compartment model in Pitchfork.

(b) Model instance for tifftopnm.

(c) Model instance for PSF2.

Figure 5: The model's structure, and two example instantiations with different domain requirements.

API, and to make other alterations such as initialization and configuration of the runtime. Once transformed, the code can be compiled with the user's preferred compiler.

7.1 Analysis

Each C source file in a project is parsed and scanned for Pitchfork annotations. When annotations are found they are checked for well-formedness and the code sandwiched between `pitchfork_start()` and `pitchfork_end()` is extracted to form segments. Segments that contain statements that affect control flow, such as `break` or `continue` are excluded unless the enclosing loop syntax is also captured in the segment. This analysis is incomplete and unsound [29] and it is complemented with empirical checks consisting of running the software's regression suite. Segments are mapped to the compartment named in the enclosing annotations. For example, the code between lines 106 and 111 in Listing 2 is mapped to the compartment called "Privileged".

After this analysis terminates, we generate the *context* for each segment: the set of program variables that are live to the code contained in the segment. Each segment's AST is traversed to gather the identifiers that are read from or written to. Since in general this can be incomplete—in case a function called in the segment accesses a global variable through a pointer for example—the context gathering can be instructed to include other variables by the user, by including their identifiers in the `pitchfork_start()` annotation. At the end of the gathering process we have a collection of identifiers $\vec{V}$ whose values will be serialized for the compartmentalized segment code to operate on them, and be returned into the main compartment for the program to continue.

7.2 Transformation

Algorithm 1 describes the transformation. We abbreviate '$S := S \cup a$' to '$S :\cup a$'. Functions(F) is the set of function declarations in source file F. Before we reach this stage, Pitchfork has checked the program and emitted metadata that will be used during this stage. This metadata includes Segments(F) which lists the segments in a file, and Context(S) which lists the variables that need to be synchronized when handing-over to segment S. The key lines in the algorithm are: **1)** obtaining the relevant metadata $\vec{V}$ generated by Pitchfork; **2)** generating a function stub S_f that will be executed in the segment's compartment; **3)** defining this new function's behavior to be that of S but sandwiched between demarshalling and marshalling steps; **4)** redefining S to marshall $\vec{V}$, use the libcompart API to reach S_f, and demarshall the state to be synchronized; **5)** updating `main()` to initialize libcompart and register all segments when the main compartment starts, as we saw in Listing 1. An example of Pitchfork's translation is shown in §D.

7.3 De/marshalling

The de/marshalling of C memory objects is decoupled from Pitchfork. The user can choose to use third-party tools and libraries [2, 3, 9, 16, 23, 58] or devise a customized approach that better suits their needs. Pitchfork emits code templates that a de/marshalling tool then completes, and which is invoked by libcompart during cross-compartment calls.

Input: 'Files' (set of annotated files—seen at step ② in Fig. 4)
foreach $F \in$ Files **do**
  foreach $S \in$ Segments(F) **do**
    $\vec{V}$:= Context(S) ; /* 1 */
    S_f := Fresh() ; /* 2 */
    Functions(F) :∪ S_f;
    def S_f : marsh ∘ $Def(S)$ ∘ demarsh($\vec{V}$) ; /* 3 */
    def S : demarsh ∘ `compart_call_fn`(S_f) ∘ marsh($\vec{V}$) ; /* 4 */
  if f = `main`() ∈ Functions(F) **then**
    foreach $S \in$ Segments(S) **do**
      R :∪ `compart_register_fn`(S_f);
    InitRegisterStart(f, R) ; /* 5 */

Algorithm 1: Transforming C translation units ('Files')

To create an end-to-end prototype we wrote a semi-automated, compile-time, de/marshalling helper tool that parses the generated template using Clang, analyzes the types in $\vec{V}$ and serializes them using `memcpy()`. Structs are handled recursively. When the tool encounters pointers and arrays it prompts the user for input, for which answers can be pre-scripted. The input consists of a bounds indication. The system used in Pitchfork can handle pointers—included repeated pointers such as `int**` and `struct my_type***`—as well as dynamic data structures such as linked lists, doubly-linked lists, and trees.

8 IMPLEMENTATION

Altogether, the Pitchfork prototype took around 3.3 person-years to develop and evaluate. The prototype is written to be performant and portable, and has minimal dependencies. libcompart consists of 1.3 KLOC of C99, including the modular IPC interface, and has no external library dependencies thus making it easier to use on different platforms. libcompart was carefully checked for memory leaks and profiled to reduce hotspots and optimize performance. Pitchfork and the de/marshaller rely on LLVM [45] and its Clang front-end for the C language. Pitchfork consists of 2.5 KLOC of C++14 and uses Boost libraries, while the de/marshaller consists of 1.7 KLOC of Python 2.7. In addition to the compartmentalizations described in this paper, the tools have been applied to a small suite of unit, system and regression tests.

9 EVALUATION

We evaluate the following: **(1)** what security goals can be achieved using Pitchfork (§9.1); **(2)** when achieving security in this way, what is the overhead to performance (§9.2) and annotation (§9.3); **(3)** how generally applicable is the Pitchfork model for privsep (§9.4).

This paper reports on 19 compartmentalizations: 10 evaluated for applicability, 5 evaluated for security (4 based on CVEs, and 1 reproduced from another paper), and 4 for performance (including 2 reproduced from another paper, and a microbenchmark). A subset of programs evaluated for applicability were further analyzed for security and performance. Software size and version information is given in §H. Usage examples for the toolchain are provided in §C.

Software	CVE-∗-∗	Vulnerability
beep	2018-0492	Race condition
PuTTY	2016-2563	Stack buffer overflow
wget	2016-4971	Arbitrary file writing
wget	2017-13089	Stack buffer overflow

Table 1: CVEs mitigated using Pitchfork in our evaluation.

Software	Work.	Time (s)	Memory (KB)	O/H (Avg% ± Std.Dev)	
				Time	Memory
wget-late	HTTP	109.9	4073.5	0.2 ± 0.3	141.5 ± 2.1
	FTP	112.1	4095	0.2 ± 0.3	141 ± 2.9
wget-early	HTTP	110.8	4103.5	0.4 ± 0.6	106.7 ± 3.5
	FTP	112.1	3993	0.4 ± 0.4	111.6 ± 2.8
netpbm	small	0.07	6.2	1.3 ± 26.2	123 ± 24.5
	large	0.40	79.7	1.58 ± 2.11	133.3 ± 13

Table 2: Overhead (O/H) analysis for compartmentalized software on different workloads. Both FTP and HTTP workloads were 5MB. "early" and "late" refer to two different wget compartmentalizations described in §9.2.2.

9.1 Security

Our security evaluation process involves **(i)** reproducing exploits described in CVEs of widely-used open-source software, **(ii)** applying privsep by using Pitchfork, then **(iii)** verifying that the exploit was no longer viable. Table 1 lists CVEs that were mitigated using Pitchfork. For each program, our *separation goal* is to isolate the part of the program that is believed to contain the vulnerability.

Setup. We used the software versions described in the CVEs and developed our own proof-of-concept exploit code when it could not be found online. For our security evaluation we used VMs running Ubuntu 16.04 LTS (kernel 4.4.0-137-generic) managed by KVM in a host running the same distribution. For 'beep' we also used a physical machine running Debian 8.11 (kernel 3.16.57-2) since the VMs had no way of relaying the beeping sound to be emitted by the host.

9.1.1 beep. beep is a 372-line, long-used, and widely-distributed terminal program for controlling the PC speaker. It was found to have a local privilege-escalation vulnerability (see Table 1). We reproduced this vulnerability on Debian 8.11 and Ubuntu 16.04. Running the exploit can take variable time because it relies on a race condition. The compartmentalized version isolates the exploitable code, undercutting its effect. We also checked that the compartmentalization did not break beep's functionality: it continued beeping. Our compartmentalization is detailed in §4.

9.1.2 PuTTY. PuTTY is a popular multi-platform client suite for SCP (secure copy), telnet, and SSH. CVE 2016-2563 reports a stack-based buffer overflow vulnerability in the SCP client. A malicious server could exploit this to remotely execute code in the client. We reproduced this vulnerability to start a shell in the client.

Our compartmentalization involves moving the exploitable string-parsing code to a different compartment and restricting the access of that compartment.

9.1.3 wget. wget is a popular console-based tool for downloading content served over HTTP, FTP, and secure encapsulations such as HTTPS. We used Pitchfork to mitigate two CVEs.

CVE-2016-4971 reports that a malicious server could deceive the the wget client to overwrite a local file by using a redirection URL. wget is run using the user's ambient privileges, thus a malicious server could use this to get a foothold in the user's machine.

Our compartmentalization consists of executing the code download in a separate compartment that is given limited visibility of the file system.

CVE-2017-13089 describes an exploit outcome that is similar to that of CVE-2016-4971 but using a different approach. When processing chunked responses, the client was not checking the chunk length. A crafted payload could exploit this to overflow a stack-based buffer.

Our compartmentalization involves executing the vulnerable code in a compartment that is given limited privileges and limited access to the file system.

9.2 Overheads

The overhead of using Pitchfork and its invocation of libcompart is measured using three metrics: **(i)** size of the resulting binary (when producing a single binary), **(ii)** execution time, and **(iii)** memory usage. Table 2 shows our results. Additional graphs are provided in §E.

Setup. We ran experiments on a machine that has 8GB RAM and 4-core Intel Core i7-3770 3.40GHz CPU running Ubuntu 16.04 LTS (kernel 4.4.0-137-generic).

Methodology. We compare measurements for the original and the compartmentalized versions of the software. Execution time overhead shows wall-clock time overhead as experienced by a user. Memory utilization was obtained by polling the kernel using `ps(1)` for the process' resident memory, to measure the memory pressure that compartmentalization is inducing on the system. We added sleep stages to ensure that we are getting an up-to-date reading of memory usage.

9.2.1 Software and Separation Goals. We evaluate time and space overhead using the following compartmentalized programs:

- wget v1.18, using the same version and security goals as in the evaluation of PtrSplit [50] to make our evaluation more comparable. These goals are described below.
- netpbm v10.73.28, is a command-line toolkit for converting and manipulating images, where we focused on the *TIFF conversion tool*. It is sometimes used by websites to automatically process images uploaded by users, who might feed it malicious content.
- "stress", a custom program we wrote to microbenchmark the IPC costs under different types of workload.

wget. Our compartmentalization of wget 1.18 is similar to that done using PtrSplit [50]. It has two main differences: (i) using libcompart the execution-time overhead is under 0.5% while that in PtrSplit is 6.5%; we expect this is because libcompart does not use runtime support for determining the bounds of dynamic data structures. (ii) we execute the HTTP and FTP functions in a separate compartment instead of generating separately-compartmentalized versions of the same software as done in PtrSplit. Methodologically, we set up dedicated HTTP and FTP servers, rate-limited for consistency and fairness—to eliminate uncertain network conditions—and widened the servers' configuration to allow carrying out more simultaneous downloads, to avoid throttling by the server. Using this setup we downloaded 200 copies (via both HTTP and FTP) of $\{1, 5, 10, 20, 50\}$MB-sized files in a single invocation of wget, to effect 200 inter-compartment calls. Each experiment was run 32 times.

netpbm. netpbm is split into 3 compartments: the main program runs in "netpbm", command-line parsing takes place in "cmdparse" and the TIFF-conversion, including TIFF parsing, takes place in "libtiff". Our separation goal is to contain potential bugs in the command-line and TIFF parsers. We measure 100 invocations of netpbm on $\{728, 477\text{K}\}$byte-sized files consisting of 16-bit images of resolution 100×100 and 5120×2880 respectively taken from the TESTIMAGES archive [26].[4] Our code is provided in §C.2.

stress. This program loops for I times, each time calling a function that accepts a B-byte null-terminated `char*`. The function simply returns the argument string back, doing no processing. In the compartmentalized version we put the function into a separate compartment, thus forcing the B-length string to be sent through libcompart machinery to the other compartment. This experiment serves to measure the overhead of inter-compartment calls as we vary I and B. We vary I from 1–500, and B from 32–496 at steps of 32.

9.2.2 Memory. As detailed below, we observe that memory usage increases over 100% in a compartmentalized program when compared to the original. This is because we now have multiple processes running simultaneously—one for each compartment—where we previously had a single process. But unlike with binary size, it tends to be much less than $O(n)$ for an n-compartment setup since not all of the process' memory needs to be resident in all compartments.

Table 2 shows two compartmentalizations of the same version of wget (1.18): an "early" one in which compartment initialization is done at startup, and a "late" one where we delay compartmentalization until the compartment is needed. In practice the "late" variant avoids us having to marshall wget's `struct options`, saving effort both at compile-time and at run-time. This struct contains the result of parsing wget's command-line parameters and its subsequent initialization.

For the "early" wget compartmentalization, Fig. 7 shows the memory use of each compartment, the total memory use across compartments, and the memory use of unmodified wget when running the same workloads. We show a similar breakdown for netpbm in §E.

[4]https://testimages.org

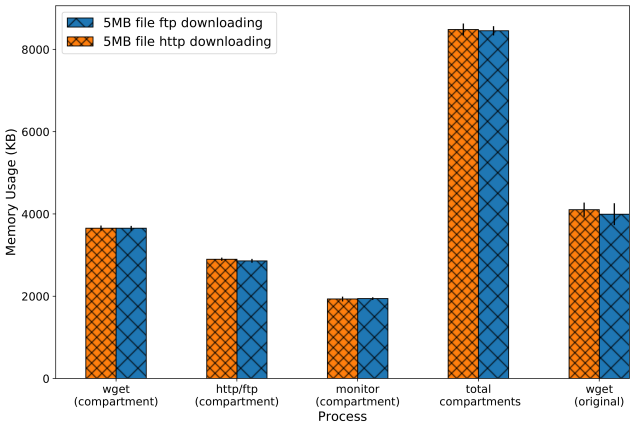


Figure 7: Memory utilization of different compartments, their total, and the original program. This shows the average and standard deviation measured from 100 runs, for both the HTTP and FTP workloads.

The "late" compartmentalization is a form of "lazy initialization" [42] which, as noted by Gudka et al., can complicate compartmentalization if the confinement decision is taken too early—since we might later find that a compartment lacks permission to carry out a needed action.

One surprising observation is that the "late" compartmentalization appears to use more memory (Table 2). We believe this to be because the resident memory of the compartmentalized system is double-counting the same accessed memory in the different compartments' processes. If compartmentalization happens earlier then the compartments' memory access patterns can appear to be more distinct, particularly for the monitor compartment which is not doing much. In contrast, if compartmentalization happens later then they appear to be more similar and lead to a larger total.

9.3 Synthesis/Annotation ratio (SAR)

We use SAR as a measure of programmer convenience:

$$\text{SAR} = \frac{\#\text{LOC Synthesized}}{\#\text{Lines of Annotation}}$$

If few lines of annotation can replace many more lines of code that the programmer would have to otherwise write, then Pitchfork can save the programmer effort through the convenience of automatically converting the annotations into code for the programmer.

Table 4 shows the SAR for different compartmentalizations in our evaluation. Two kinds of code are synthesized: compartmentalization code—such as compartment initialization, hand-over, etc—and de/marshalling code.

For example, our beep compartmentalization adds 9 lines of annotation: one line for `#include "pitchfork.h"` and four `pitchfork_start()`–`pitchfork_end()` pairs, each specifying a segment in a compartment. Pitchfork expanded these annotations into 750 lines for compartmentalized beep.[5]

[5]Since the beep CVE was published, the beep code was forked, improved and expanded into the spkr-beep project [21]. Compared to beep's single file containing 372 lines, spkr-beep has 15 files totaling 1634 lines.

Software	Plat.	Separation Goal
cURL	L	Command invocation, parsing, file transfer.
Evince	L	libspectre dependency—see §2.
git	L	Historical vulnerability [11].
ioquake3	m	Applying server updates.
tifftopnm	L	Separating parsers—see §C.
nginx	L	HTTP request parsing
redis	L	Isolating low-use commands.
tcpdump, uniq	F	Leveraging Capsicum [59].
Vitetris	L	Network-facing code—see §2.

Table 3: libcompart use-cases, providing examples of its applicability to different software, platforms, and separation goals. Platforms: FreeBSD, Linux, macOS. In each example we used a single domain (§5). For tcpdump and uniq we show that libcompart can leverage already-implemented isolation that uses platform-specific mechanisms to secure processes.

Soft.	#LOC	#Annot.	#LOC Synthesized		SAR
			Compart.	De/marsh.	
beep	372	9	133	245	42
PuTTY	123K	6	52	29	13.5
wget[6]	62.6K	3	65	168	77.7
wget[7]	62.8K	8	57	38	11.9

Table 4: Measure of convenience: how much code is synthesized by Pitchfork per annotation.

9.4 Applicability

To test the applicability of Pitchfork's model we compartmentalized the software listed in Table 3 to use libcompart. Some of these compartmentalizations and their separation goals were described earlier in the paper, so here we focus on the remaining ones. Through their separation goals we can classify the compartmentalizations into four sets. The first set consists of **ioquake3** [13] which is derived from the open-sourcing of Quake 3. The separation goal is similar to that of the privsep for wget's CVE-2016-4971 (§9.1.3). When connecting to a server the game might download files from that server, and we want to confine the download to a specific location. The second set consists of **uniq** and **tcpdump**, which we use to show that libcompart can coexist with OS-specific sandboxing that is already in place. The third set consists of **git**, **redis**, and **cURL**, for which we applied the first heuristic in §2.2 to restrict part of the program to only the data it needs to process. Finally, the fourth set consists of **nginx** for which we applied the first and third heuristics in §2.2 to confine the processing of untrusted input.

[6]wget v1.17.1.50-2bdfc used to reproduce CVE-2016-4971
[7]wget v1.19.1 used to reproduce in CVE-2017-13089

10 RELATED WORK

Privman [44] provides both a compartmentalization API and a configuration language. This API was used through manual modification of a program's source code. The API largely shadowed the POSIX API and required extra care to learn and use the configuration language. Privman only distinguished between two compartments: the unpriviliged compartment and the privileged monitor. Through libcompart, Pitchfork provides a much simpler API that does not need to scale with that of POSIX, and allows the use of arbitrarily-many compartments in which arbitrary segments are executed. In Pitchfork the monitor is not privileged, it only serves as a watchdog to ensure that compartments do not block indefinitely.

Wedge [30] presented a new API for compartmentalization but relied upon new, kernel-provided facilities such as a default-deny forking model and flexible tagged-memory permission system at the interface between compartments. In comparison, Pitchfork works with existing commodity kernels. Pitchfork's compartmentalization model is more rigid—compartments cannot be arbitrarily spawned at runtime and can only communicate with the main compartment. In practice so far this restriction has not proved to be limiting.

Since modifying software's source code can both be difficult and risks introducing bugs, researchers proposed annotation-driven transformation as in Privtrans [33], Glamdring [49] and PtrSplit [50]. As with Privman the model used by these systems targets a separation between a privileged and an unprivileged compartment. In comparison, Pitchfork supports multiple compartments. This improves security since it supports an arbitrary number of differently-trusted compartments—different parts of the program and dependencies can be granted different privileges. Pitchfork provides better support for de/marshalling complex types compared to Privtrans [33] and Glamdring [49]; this adds convenience to the programmer. Compared to PtrSplit [50], Pitchfork does not rely on runtime memory tracking in software, and this leads to better performance because of lower overhead, but exploring the use of hardware-provided support for memory tracking is future work.

ConfLLVM [31] adapts LLVM's IR to use source-level annotations to identify and protect confidential variables. The annotations consist of adding a `private` qualifier to declarations. In comparison, the Pitchfork approach involves using the source-to-source tool or libcompart directly; in either case users caninspect the processed source code. Pitchfork does not replace the compiler, thus avoiding disruption to portability. Pitchfork does not rely directly on hardware enforcement and does not assume a trusted-vs-untrusted distinction—instead, it allows an arbitrary number of compartments which can be differently-trusted.

11 FUTURE WORK

A future research direction involves generalizing Pitchfork's model further to overcome the limitations described at the end of §5. Another direction for future work involves automating the insertion of source-level Pitchfork annotations. This could build on the model and tooling contributed by this paper, and gradually replace human guidance with automation.

ACKNOWLEDGMENTS

We thank Federico Bento for help with his CVE-2015-6565 PoC, and the ACSAC reviewers for their feedback. Code sizes were generated using David A. Wheeler's 'SLOCCount'. This material is based upon work supported by a Google Research Award and by the Defense Advanced Research Projects Agency (DARPA) under contracts HR0011-19-C-0106 and HR0011-20-C-0191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funders.

REFERENCES

[1] About FreeBSD Ports. https://www.freebsd.org/ports/. (????).
[2] Apache Thrift. https://thrift.apache.org/. (????).
[3] C serialization library. http://www.happyponyland.net/cserialization/readme.html. (????).
[4] Clang: a C language family frontend for LLVM. https://clang.llvm.org/. (????).
[5] CVE-2018-0492. https://nvd.nist.gov/vuln/detail/CVE-2018-0492. (????).
[6] CVE Details for ncurses. https://www.cvedetails.com/google-search-results.php?q=ncurses. (????).
[7] CVE Details for vitetris. https://www.cvedetails.com/google-search-results.php?q=vitetris. (????).
[8] Evince document viewer. https://wiki.gnome.org/Apps/Evince. (????).
[9] FlatBuffers. https://github.com/google/flatbuffers. (????).
[10] GhostScript CVEs. https://www.cvedetails.com/vulnerability-list.php?vendor_id=7640&product_id=0. (????).
[11] git CVE-2010-2542. https://www.cvedetails.com/cve/CVE-2010-2542/. (????).
[12] Holey Beep. https://holeybeep.ninja/. (????).
[13] ioquake3. https://ioquake3.org/. (????).
[14] libtiff CVEs. https://www.cvedetails.com/product/3881/Libtiff-Libtiff.html?vendor_id=2224. (????).
[15] Netpbm home page. http://netpbm.sourceforge.net/. (????).
[16] Protocol Buffers. https://developers.google.com/protocol-buffers/. (????).
[17] Revised OpenSSH Security Advisory. https://www.openssh.com/txt/preauth.adv. (????).
[18] seccomp API. https://github.com/torvalds/linux/blob/master/Documentation/userspace-api/seccomp_filter.rst. (????).
[19] Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html. (????).
[20] SLOCCount. https://dwheeler.com/sloccount/. (????).
[21] spkr-beep project. https://github.com/spkr-beep/beep. (????).
[22] The Chromium Projects: Process Models. https://www.chromium.org/developers/design-documents/process-models. (????).
[23] TPL: easily store and retrieve binary data in C. http://troydhanson.github.io/tpl/. (????).
[24] VITETRIS - Virtual terminal *tris clone. https://github.com/vicgeralds/vitetris. (????).
[25] 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/tsai
[26] Nicola Asuni and Andrea Giachetti. 2013. TESTIMAGES: A Large Data Archive For Display and Algorithm Testing. *Journal of Graphics Tools* 17, 4 (2013), 113–125. https://doi.org/10.1080/2165347X.2015.1024298 arXiv:https://doi.org/10.1080/2165347X.2015.1024298
[27] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 19, 17 pages. https://doi.org/10.1145/2901318.2901350
[28] Nick Benton, Luca Cardelli, and Cédric Fournet. 2004. Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst.* 26, 5 (sep 2004), 769–804. https://doi.org/10.1145/1018203.1018205
[29] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (feb 2010), 66–75. https://doi.org/10.1145/1646353.1646374
[30] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *NSDI*, Vol. 8. 309–322.
[31] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. 2019. ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 15 pages. https://doi.org/10.1145/3302424.3303952
[32] Peter Bright. 2016. Firefox takes the next step toward rolling out multi-process to everyone. (Dec 2016).
[33] David Brumley and Dawn Song. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*. 57–72.
[34] David R Butenhof. 1997. *Programming with POSIX threads*. Addison-Wesley Professional.
[35] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/3052973.3052983
[36] Brian Caswell, James C. Foster, Ryan Russell, Jay Beale, and Jeffrey Posluns. 2003. *Snort 2.0 Intrusion Detection*. Syngress Publishing.
[37] Haogang Chen, Cody Cutler, Taesoo Kim, Yandong Mao, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2013. Security bugs in embedded interpreters. In *Proceedings of the 4th Asia-Pacific Workshop on Systems - APSys'13*. ACM Press. https://doi.org/10.1145/2500727.2500747
[38] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[39] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. *SIGOPS Oper. Syst. Rev.* 42, 2 (March 2008), 103–114. https://doi.org/10.1145/1353535.1346295
[40] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia
[41] Dan Goodin. 2019. The year-long rash of supply chain attacks against open source is getting worse. (Aug 2019).
[42] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611
[43] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (aug 1978), 666–677. https://doi.org/10.1145/359576.359585
[44] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track*. 273–284.
[45] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[46] Michael Lee. 2012. Google strengthens Chrome for Android with sandbox. (Sep 2012).
[47] E. Levy. 2003. Poisoning the software supply chain. *IEEE Security Privacy* 1, 3 (May 2003), 70–73. https://doi.org/10.1109/MSECP.2003.1203227
[48] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 418–429. https://doi.org/10.1145/3274694.3274720
[49] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 285–298. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind
[50] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066
[51] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Not.* 44, 6 (June 2009), 245–258. https://doi.org/10.1145/1543135.1542504
[52] N. Nguyen, P. Reiher, and G. H. Kuenning. 2003. Detecting insider threats by monitoring system call activity. In *IEEE Systems, Man and Cybernetics Society-Information Assurance Workshop, 2003.* 45–52. https://doi.org/10.1109/SMCSIA.2003.1232400
[53] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-time. *Comput. Netw.* 31, 23-24 (Dec. 1999), 2435–2463. https://doi.org/10.1016/S1389-1286(99)00112-7

[54] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369

[55] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. 2004. Attestation-based Policy Enforcement for Remote Access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 308–317. https://doi.org/10.1145/1030083.1030125

[56] Jerome H Saltzer and Michael D Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[57] Nik Sultana, Achala Rao, Zihao Jin, Pardis Pashakhanloo, Henry Zhu, Ke Zhong, and Boon Thau Loo. 2018. Making Break-Ups Less Painful: Source-Level Support for Transforming Legacy Software into a Network of Tasks. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '18)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/3273045.3273046

[58] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

[59] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1929820.1929824

[60] Jinpeng Wei and Calton Pu. 2010. Modeling and Preventing TOCTTOU Vulnerabilities in Unix-style File Systems. *Comput. Secur.* 29, 8 (Nov. 2010), 815–830. https://doi.org/10.1016/j.cose.2010.09.004

[61] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in Npm with Latch. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1139–1153. https://doi.org/10.1145/3488932.3523262

[62] Henry Zhu, Nik Sultana, and Boon Thau Loo. 2020. Debugging strongly-compartmentalized distributed systems. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 538–547. https://doi.org/10.1109/IPDPSW50202.2020.00096

Appendix A LIBCOMPART API

```
// Configuration for a single compartment.
struct compart {
  const char * const name;
  uid_t uid;
  gid_t gid;
  const char * const path;
  void *comms;
};

// Configuration for the monitor.
struct compart_config {
  int call_timeout;
  int activity_timeout;
  void (*on_call_timeout)(int compart_idx);
  void (*on_activity_timeout)();
  void (*on_termination)(int compart_idx);
  void (*on_comm_break)(int compart_idx);
  unsigned start_subs : 1;
};

// Flat buffer for de/marshalled data in cross-compartment
    calls.
struct extension_data {
  size_t bufc;
  char buf[EXT_ARG_BUF_SIZE];
};

// Handle for registered segments.
struct extension_id;

// Initialize by detailing a fixed number of compartments
    and the monitor config.
void compart_init(int no_comparts, struct compart comparts
    [], struct compart_config config);
// Start compartmentalization.
void compart_start(const char * const new_compartment_name);
// Start a specific compartment.
void compart_as(const char * const compartment_name);
// Register a segment in a compartment.
struct extension_id *compart_register_fn(const char * const
    new_compartment_name, struct extension_data (*fn)(
    struct extension_data));
// Call a segment in a compartment.
struct extension_data compart_call_fn(struct extension_id *,
     struct extension_data);
void compart_log(const char *buf, const size_t count);
// Name of the containing compartment.
const char * compart_name(void);
```

Listing 3: Part of libcompart's API

Appendix B PLUGGABLE COMMUNICATION API

This API was designed to serve the inter-compartment communication needs in Pitchfork's compartmentalization model (§5). Different instantiations of this API wrap different, possibly OS-specific, IPCs.

```
// Opaque communication channel, wraps IPC.
struct compost;

// Initialize with compartment information.
void compost_init(int local_no_comparts, struct compart *
    comparts);
// Start communication system.
void compost_start(const char * const compartment_name);
// Start for a specific compartment.
void compost_as(const char * const compartment_name);
// main -> monitor channel.
const struct compost *compost_m2mon(void);
// monitor -> main channel.
const struct compost *compost_mon2m(void);
// main -> segment channels.
const struct compost *compost_m2(int compart_idx);
// segment -> main channels.
const struct compost *compost_2m(int compart_idx);
// Send data to compartment's channel.
ssize_t compost_send(const struct compost *cp, const void *
    buf, size_t count);
// Receive data from compartment's channel.
ssize_t compost_recv(const struct compost *cp, void *buf,
    size_t count);
// Close communication channel.
void compost_close(struct compost *cp);
```

Listing 4: libcompart's communication API

Appendix C USAGE EXAMPLES

This section provides a tutorial-style description of achieving specific security goals using the Pitchfork tool and libcompart. In §C.1 we protect secret data from being exfiltrated through vulnerable code. We place the secret data in a compartment. For additional safety we place the compartment on a secured machine. In §C.2 we protect against potentially vulnerable parts of a program by confining parsing code, trading overhead for security.

C.1 Example from PtrSplit

Our next example is adapted from the PtrSplit paper [50, Fig.2] to show how Pitchfork can be applied to that example. Listing 5 shows the annotated program code. Note that the program has a format-string vulnerability on line 11, using which a user could extract the value of the `key`, for example.

We create a single compartment to contain the sensitive information. Note that this compartment consists of two segments. Line 1 brings Pitchfork annotations into the namespace.

```
1 #include "pitchfork.h"
2 #include <stdio.h>
3 ...
4 char key[64];
5
6 void initkey() {
7 ...
8 }
```

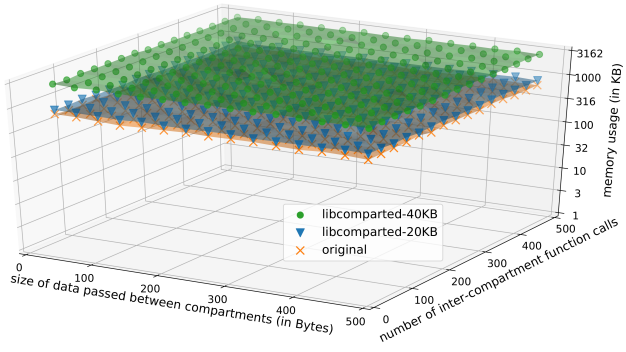


Figure 8: stress: Increasing the size of libcompart's serialization buffer used by the external de/marshalling tool.

```
9
10 void greeter (char *str) {
11   printf(str); printf(", welcome!\n");
12 }
13
14 char * encrypt (char * plaintext, int sz) {
15 ...
16 }
17
18 int main (int argc, char **argv) {
19   char username[64];
20   char *text = malloc(sizeof(*text) * 1024);
21   char *key_ptr = key;
22
23   pitchfork_start("Classified");
24   initkey();
25   pitchfork_end("Classified");
26
27   printf("Enter username: ");
28   fgets(username, sizeof(username), stdin);
29   greeter(username);
30
31   printf("Enter plaintext: ");
32   fgets(text, sizeof(text), stdin);
33
34   char *ciphertext = NULL;
35   pitchfork_start("Classified");
36   ciphertext = encrypt(text, strlen(text));
37   pitchfork_end("Classified");
38
39   printf("Cipher text: ");
40   for (unsigned i=0; i<strlen(text); i++) {
41     printf("%x ",ciphertext[i]);
42   }
43   printf("\n");
44   return 0;
45 }
```

Listing 5: Pitchfork-annotating an example from PtrSplit.

In Listing 5 note the fine-grained scoping of compartmentalization. Pitchfork allows compartments to be as small as a single line of code. Compartments may also span entire libraries, as in the next example.

Pitchfork helps programmers generate a binary for each compartment. To do this the program is compiled multiple times, once for every compartment. Fig. 6 shows how we ran this example over the network. The compartments communicated over TCP. Running compartments on different machines can be done to quarantine the machine running untrusted code [52] or to protect against clients that are untrusted or unvouched [55].

Running compartments on separate machines exposes more security dimensions for compartmentalized software. One is to improve security by leveraging use of host-based policy or network-based intrusion-detection [36, 53] for individual compartments. Another is through balancing security against performance overhead—in this case the application will be subject to the network round-trip time (RTT) with its compartments.

C.2 netpbm

netpbm is a command-line toolkit for converting and manipulating images. It is extremely portable and is sometimes used by websites to automatically process images uploaded by users [15], who might feed it malicious content.

Like other image processing tools, netpbm relies on specialized libraries for different graphic formats. Over the years several vulnerabilities have been discovered in these kinds of libraries where a malicious user-supplied image can lead to code execution. For example, the widely-deployed library for TIFF[8] has had several such occurrences over the years [14].

We compartmentalize netpbm's TIFF conversion tool to confine the use of libtiff. Out of caution, we also create a separate compartment for the parsing of command-line options. Sections 4 and C.1 showed multiple segments in a single compartment; this section shows multiple compartments in a single program.

Instead of showing annotations in this example, we lift the hood and show what these annotations are translated to. We show the "diff" between the original netpbm program and our compartmentalized version in Listing 6.

The compartmentalized code uses libcompart (§6). Line 4 initializes libcompart and configures it. Then segments are registered with each compartment. In this case we have two non-main compartments containing one segment each, registered on lines 5 and 6. Finally, the main compartment is started on line 7; it automatically starts other compartments.

The rest of the "diff" involves transferring code to compartments and invoking those compartments. Line 4 calls the command-line parsing function; this is moved to a compartment and called on line 19—using a handle obtained by registering the segment on line 6. The call is sandwiched between context preparation and transfer to and from the compartment on lines 18 and 20 respectively.

The invocation of the "libtiff" compartment follows the same pattern. Note the comment on line 30. libcompart has flexible failure-handling designed in its API, and its default action propagates the failure unless the default configuration (line 4) is changed by the programmer.

[8] https://gitlab.com/libtiff/libtiff

```
+#include "netpbm_interface.h"
int
main(int argc, const char * argv[]) {
  +compart_init(NO_COMPARTS, comparts, default_config);
  +convertTIFF_ext = compart_register_fn("libtiff", &
       ext_convertTIFF);
  +parseCommandLine_ext = compart_register_fn("cmdparse"
       , &ext_parseCommandLine);
  +compart_start("netpbm");

  struct CmdlineInfo cmdline;
  TIFF * tiffP;
  FILE * alphaFile;
  FILE * imageoutFile;

  pm_proginit(&argc, argv);
  -parseCommandLine(argc, argv, &cmdline);
  +struct extension_data arg;
  +args_to_data_CommandLine(&arg, argc, argv);
  +arg = compart_call_fn(parseCommandLine_ext, arg);
  +args_from_data(&arg, &cmdline);
  -tiffP = newTiffImageObject(cmdline.inputFilename);
  -if (cmdline.alphaStdout)
...
  -TIFFClose(tiffP);
  +args_to_data(&arg, &cmdline);
  +arg = compart_call_fn(convertTIFF_ext, arg);
  pm_strfree(cmdline.inputFilename);

  /* If the program failed, it previously aborted with
       nonzero completion
     code, via various function calls.
  */
  return 0;
```

Listing 6: libcompart used on tifftopnm.c.

Appendix D SEGMENT-CALLING EXAMPLE

Listing 7 shows example output from our CVE-based compartment evaluation for 'beep' (§9.1.1). The de/marshalling template is included on line 1, and the marshalled value is passed to the libcompart API function `compart_call_fn()`, together with an opaque reference generated by libcompart when we registered the function to be called from another compartment. This reference is declared on line 14 in Listing 7, and defined as follows: `Privileged_1_call_ext = compart_register_fn("Privileged", &Privileged_1_call_remote)` ; This registration takes place after libcompart initialization and before start-up, as we saw in Listing 1.

```
#include "Privileged1_demarsh.c"

struct extension_data Privileged_1_call_remote(struct
    extension_data __arg) {
  int console_fd;
  struct input_event e;
  Privileged_1_call_demarshall(__arg, &console_fd, &e);
    if(write(console_fd, &e, sizeof(struct input_event)) <
        0) {
      putchar('\a'); /* See above */
      perror("write");
    };
  return Privileged_1_call_marshall(console_fd, e);
}

struct extension_id *Privileged_1_call_ext = NULL;

void Privileged_1_call(int *console_fd, struct input_event *
    e) {
  struct extension_data __arg = Privileged_1_call_marshall(*
      console_fd, *e);
  struct extension_data __result = compart_call_fn(
      Privileged_1_call_ext, __arg);
  Privileged_1_call_demarshall(__result, console_fd, e);
}
```

Listing 7: Synthesized cross-compartment segment calling.

Appendix E FURTHER DETAILS ON OVERHEAD

Further to the evaluation presented in §9.2, Fig. 9 presents the compartment-memory breakdown for netpbm; this is similar to how Fig. 7 showed this evaluation for wget. Fig. 10 shows how the performance of compartmentalized wget varies with protocol and download size. Fig. 11 graphs the results described in §E.3 and Fig. 8 shows a version of Fig. 11b where we used larger de/marshalling buffers.

For Fig. 9 we calculated the standard deviation for each column but it was too small to observe in the graph, so we provide its values here. For the large workload the std.dev is 676.2 for original netpbm and the largest std.dev is 992.0, for the libtiff. For the small workload the std.dev is 167.9 for original netpbm and the largest std.dev is 218.1, for the libtiff.

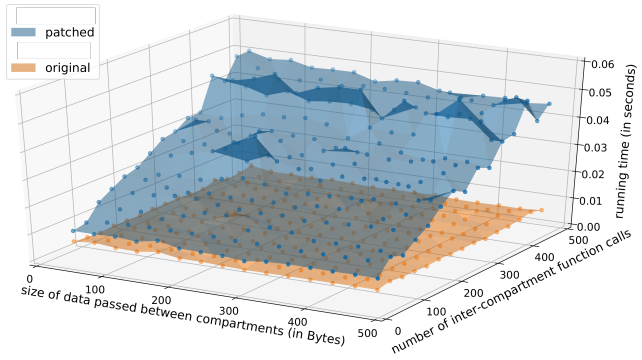
Fig. 8 shows the different memory usage when we increase the size of libcompart's fixed-size buffer. The graph shows that the memory costs are fixed. The memory size can be tuned by the programmer or the external de/marshalling tool.

E.1 Binary Size

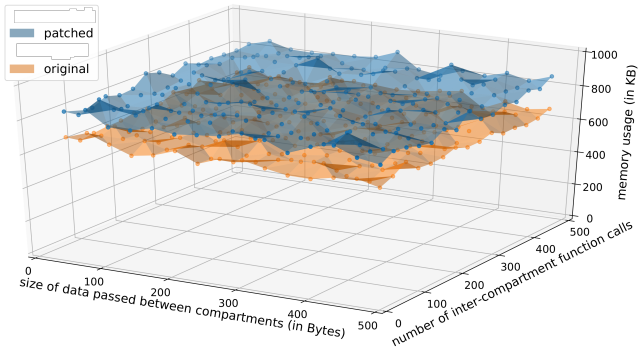
The size overhead on single-compartment binaries was measured to be less than 6% across our experiments. The contributors to this overhead consist of libcompart, the de/marshalling code, and the transformations made by Pitchfork. For separate binaries we see per-compartment duplication since we do not eliminate code that is unused in a compartment—that is an orthogonal problem. So in the worst case the total size of separate binaries of an n-compartment program is in $O(n)$.

E.2 Execution time

For the software, compartmentalizations and workloads we used, there was hardly any difference in execution time between the original and compartmentalized version. This is likely because the separation goals did not result in a compartmentalization that caused the execution-time overhead of the compartmentalization to accumulate sufficiently to become significant.



(a) Time overhead observed when varying the number of calls across compartments and varying the amount of data sent in each call.



(b) Memory overhead observed when varying the number of calls across compartments and varying the amount of data sent in each call.

Figure 11: stress: Cross-compartment communication overheads.

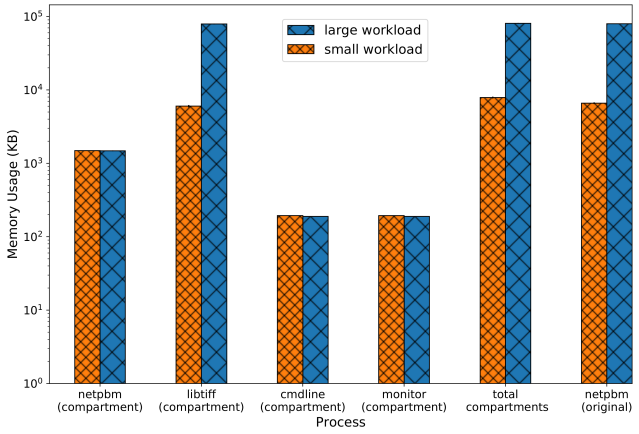


Figure 9: netpbm: Memory utilization of different compartments, their total, and the original program. This shows the average and standard deviation measured from 100 runs.

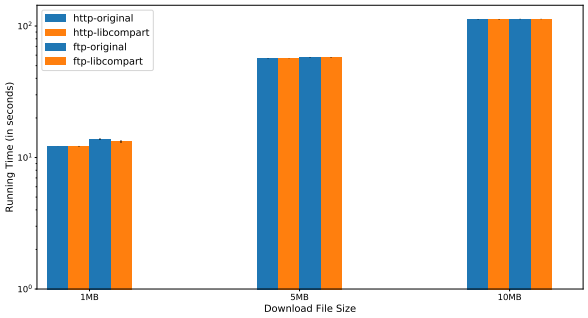


Figure 10: wget: Effect of increasing the downloaded-file size on execution-time overhead, for both FTP and HTTP downloads. This experiment is described in §9.2.

In order to measure pessimistic conditions in which compartmentalization's overheads repeatedly featured in a program's critical path, we designed the experiment described in the next section.

E.3 Communication overhead

Kilpatrick [44] had found inter-compartment communication to be the main source of overhead in his system. This section reports our measurements of communication overheads between compartments by using the "stress" program that repeatedly initiates transfers between compartments.

We find that execution-time overhead increases linearly with the number of cross-compartment calls. The full graph is provided in Fig. 11 in §E. In the parameter range that we consider, memory overhead does not vary with the number of inter-compartment function calls or size of data crossing the compartment. It is stable from run to run, and is only associated with the number of compartments initialized.

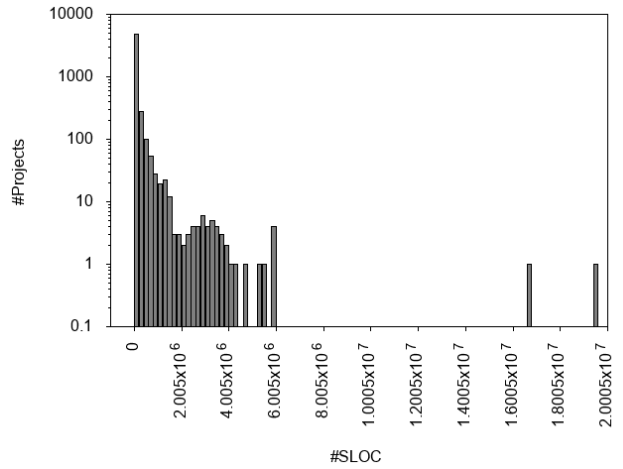
Appendix F MAINTAINABILITY AND EVOLVABILITY (M&E)

Software is changed when bugs are fixed and features are added. Once privsep is applied to software, the presence of privsep-related logic should not obstruct or slow down that software's evolution. This section describes two interactions we encountered between Pitchfork -based privsep and the maintainability or evolvability of software after it has been compartmentalized. Neither of the two resulted in additional complications, which suggests a positive result, but a larger-scale study is needed to get a better understanding.

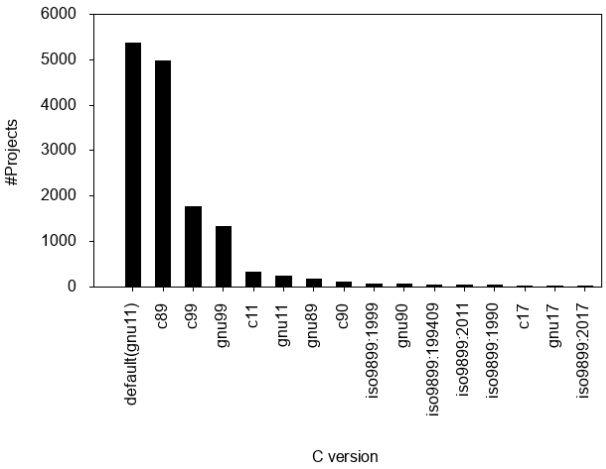
We use the 3 versions of wget compartmentalized earlier: **(i)** 1.17.1.50-2bdfc for CVE-2016-4971, **(ii)** 1.18 used for the overhead measurements in §9.2, and **(iii)** 1.19.1 for CVE-2017-13089.

The first example involves (ii), and only involved a small change to the annotation.

For the second example we combined (i) and (iii), as the furthest-apart versions of wget we had used, by modifying (iii) to additionally include the compartmentalization we carried out for (i). This produced a copy of v1.19.1 that compartmentalized as the union of (i) and (iii). This too only involved a small change to the annotation. Updates to wget could continue to take place in the annotated version of the software—② in Fig. 4—and Pitchfork would be rerun to generate the transformed source code.



(a) Projects of at least 5000 lines of C.



(b) Versions of C used.

Figure 12: Histograms generated from surveying C usage among all FreeBSD 12.2-STABLE ports.

Appendix G SURVEY OF FREEBSD PORTS

We sought to use representative software use-cases in our evaluation. Since representativeness is a vague property, we narrowed this down to project size in source-lines-of-code (SLOC) and to variants of the C language. We were targeting open-source software in our evaluation but could not find a recent analysis of such code-bases, so we made our own. This section describes our methodology and results, expanding on the description in §2.3.

The survey was done over FreeBSD ports that use C, summarized in Fig. 12. This analysis took 12.4 hours to execute. To measure size we used sloccount [20] v2.26.

We heuristically excluded projects based on programming language (i.e., non-C code-bases) and non-source archives—such as .jar, .rpm, .gem, .cabal, and .nupkg. This left 11100 projects. Many of these projects contained very small pieces of C, many of which consisted of header files and wrappers—for example, to make a C library usable in a Rust project. Small codebases (i.e., fewer than 5KLOC) are cropped to avoid biasing towards smaller codebases, and to make the histogram more legible. This left 5336 projects.

Fig. 12b shows the versions of C used by different projects, and Fig. 12a graphs the project sizes. A few projects are huge—millions or tens of millions of lines. The largest projects consisted of kernels—such as Linux and NetBSD—QEMU, GCC and browsers—both Firefox and Chromium.

Appendix H SIZES OF EVALUATED PROJECTS

Table 5 shows version and size information for some of the use-case software used in this paper. Size was measured using sloccount [20] v2.26. The projects rely on different versions of C including c11, gnu11, c99, and gnu99.

Software	Version	LOC
vitetris	0.57.2	13.4K
evince	2.24.0	83.8K
wget	1.19.1	84K
nginx	1.19.4	144K
git	2.28.0.394.ge197136.dirty	234K
redis	58e5feb3f49c50b9c18f38fd8f6cad2317c02265	140K
curl	7.71.1	152K
netpbm	10.73.28	192K
ioq3	1.36	392K

Table 5: Version and size information for use-case software used in this paper.