Clean Application Compartmentalization with SOAAP

Khilan Gudka
University of Cambridge
Cambridge, UK
khilan.gudka@cl.cam.ac.uk

Robert N.M. Watson
University of Cambridge
Cambridge, UK
robert.watson@cl.cam.ac.uk

Jonathan Anderson
Memorial University
St. Johns, NL, Canada
jonathan.anderson@mun.ca

David Chisnall
University of Cambridge
Cambridge, UK
david.chisnall@cl.cam.ac.uk

Brooks Davis
SRI International
Menlo Park, CA, USA
brooks.davis@sri.com

Ben Laurie
Google UK Ltd.
London, UK
benl@google.com

Ilias Marinos
University of Cambridge
Cambridge, UK
ilias.marinos@cl.cam.ac.uk

Peter G. Neumann
SRI International
Menlo Park, CA, USA
neumann@csl.sri.com

Alex Richardson
University of Cambridge
Cambridge, UK
alr48@cam.ac.uk

ABSTRACT

Application compartmentalization, a vulnerability mitigation technique employed in programs such as OpenSSH and the Chromium web browser, decomposes software into isolated components to limit privileges leaked or otherwise available to attackers. However, compartmentalizing applications – and maintaining that compartmentalization – is hindered by *ad hoc* methodologies and significantly increased programming effort. In practice, programmers stumble through (rather than overtly reason about) *compartmentalization spaces* of possible decompositions, unknowingly trading off correctness, security, complexity, and performance. We present a new conceptual framework embodied in an LLVM-based tool: the Security-Oriented Analysis of Application Programs (SOAAP) that allows programmers to reason about compartmentalization using source-code annotations (*compartmentalization hypotheses*). We demonstrate considerable benefit when creating new compartmentalizations for complex applications, and analyze existing compartmentalized applications to discover design faults and maintenance issues arising from application evolution.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Security, Compartmentalization, Vulnerability Mitigation

1. INTRODUCTION

Application compartmentalization decomposes software into isolated but collaborating components to mitigate exploitation of vulnerabilities. It seeks to employ the *principle of least privilege* by granting each compartment only the rights that it requires to operate [38]. It is used in software ranging from OpenSSH [36] to the Chromium web browser [37] to limit rights available to an attacker following a successful exploit. Recently disclosed vulnerabilities such as Heartbleed [30], Shellshock [32], and a symlink attack vulnerability in Wget [31], illustrate the importance of protecting software not just from known exploit techniques (e.g., stack canaries mitigating buffer overflows), but also from unknown future classes of vulnerabilities.

Application compartmentalization proves surprisingly subtle: contrary to the implied suggestion of simplistic descriptions ("Application X has been privilege separated"), there are many possible compartmentalizations of most programs within a *compartmentalization space* that reflects different tradeoffs among correctness, security, complexity, and performance. Selecting a suitable point in this space is challenging, as implementing compartmentalization requires considerable effort and can disrupt code structure. This choice is crucial, as it can be difficult to change strategy after implementation: the non-trivial code changes made to application structure are often specific to the selected compartmentalization. Design choices and tradeoffs are infrequently revisited – even as understanding of program security grows due to discovered vulnerabilities, or as program functionality evolves beyond its compartmentalization.

Reasoning about compartmentalization tradeoffs is difficult for many reasons. Information about past vulnerabilities – often a good predictor of future vulnerabilities – is not easily accessible. Call graphs of compartmentalized applications are extremely complex, and simple control-flow analysis cannot follow manually encoded cross-domain actions – typically via *Inter-Process Communication* (IPC) – found in real-world applications. Even where call graphs can be extracted, control flow is only a starting point: in the compartmentalization adversary model, naively written code may be vulnerable, but it is exposure to malicious data that causes exploits – requiring reasoning about a combination of vulnerabilities, information flow from dangerous sources, and uncompartmentalized code. Application failures due to compartmentalization are frequently mysterious: the libraries they link against are often designed without the

CCS'15 October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10$15.00
DOI: http://dx.doi.org/10.1145/2810103.2813611.

expectation of sandboxing, and failures may not be detected by testing. Finally, performance characteristics are critical, with compartmentalization boundaries often selected that are not based on ideal security boundaries, but rather on acceptable performance (e.g., in Capsicum where `gzip` rather than `zlib` is sandboxed [42]). Even then, performance impacts are difficult to predict.

Application developers would ideally be able to explore how new or changing compartmentalization would affect factors such as security and performance without paying the full cost of implementation, letting them more easily identify a preferred spot in the compartmentalization space. For example, partitioning decompression from decryption can prevent decompression-related vulnerabilities from leaking keying material – but at great costs to both code complexity and run-time performance. The overhead of this compartmentalization may be too high for production use, requiring another approach, such as having per-client compartments.

This exploration is comparable to the use of performance-profiling tools to identify the hot spots in an application that would benefit most from optimization. Similarly, security profiling tools could enable the developer to determine whether a particular compartmentalization was "worth it", namely that it would give sufficient security gains while not exceeding tolerable performance overheads. Such an approach would also allow continuous automatic reevaluation as application source code evolves, ensuring that, for example, refactorings do not accidentally move vulnerable code to a more privileged context. To this end, this paper:

- Describes a conceptual framework for compartmentalization grounded in *isolation* and *controlled communication*. Possible program compartmentalizations, each with differing design tradeoffs in terms of complexity, performance, security, and functionality, exist within *compartmentalization spaces*. We also describe a set of *compartmentalization design patterns* reflecting common approaches to compartmentalization.
- Presents the Security-Oriented Analysis of Application Programs (SOAAP), an LLVM-based tool for evaluating proposed (or extant) compartmentalizations based on annotated *compartmentalization hypotheses*, which document programmer expectations and designs for compartmentalization and security. Source-code annotations represent sandboxing strategies, cross-domain communication, past vulnerabilities, sensitive or risky data, and security goals. SOAAP scales to multi-million-LoC applications, providing efficient control-flow and information-flow analysis of whole programs.
- Demonstrates the effectiveness of our approach through analysis of existing real-world compartmentalizations – including OpenSSH and the Chromium web browser – and discover bugs that were introduced by developing these without the aid of a SOAAP-like tool. We also show the use of SOAAP in adding new compartmentalization to two existing applications: FreeBSD's `fetch` and the Okular document viewer from KDE. SOAAP has identified various weaknesses, including untrustworthy code unexpectedly executed outside of sandboxes, and provides a new methodology for evaluating software compartmentalization.

2. CONCEPTUAL FRAMEWORK

Application compartmentalization rests on the principle of least privilege: compromised components yield fewer rights to attackers, and offer fewer opportunities to compromise the rest of the system. Karger first proposed using access control for vulnerability mitigation while working on capability systems [19], and took as his threat model the *trojan horse*: subverted software working on behalf of the adversary. Contemporary compartmentalization for vulnerability mitigation, sometimes referred to as *privilege separation*, saw its foundations in work by Provos et al. [36] and Kilpatrick [20], and has been applied to complex, security-relevant programs such as OpenSSH and the Chromium web browser, both of which hold substantial effective privilege and perform complex processing of untrustworthy, network-originated data.

Compartmentalization adopts a *de facto*, and quite strong, threat model in which attackers gain total control of compromised compartments as a result of poorly crafted C code (or other program weaknesses) being being exposed to malicious input. For example, buffer-overflow attacks triggered by network input might allow malicious code to be executed, or program control flow to be fully subverted by the attacker. We generalize this model, adding software supply-chain vulnerabilities involving explicitly trojaned software without the need for malicious input – e.g., back doors being inserted following compromise of a software vendor's source-code repository. Weaker attacks, which fall short of arbitrary code execution, are also important: the OpenSSL Heartbleed vulnerability allows read access to victim memory, which might reveal keys allowing further access, but does not immediately allow arbitrary manipulation.

Compartmentalization rests on three underpinnings: a strong *Trusted Computing Base (TCB)* able to protect its own integrity [3]; *compartment isolation* (implemented by the TCB), which provides strong prevention of interference between isolated program instances [11]; and *controlled communication*, which allows safe communication between compartments – subject to suitable policies. Applications may, themselves, construct application-specific TCBs on top of the compartmentalization substrate – for example, components that run with ambient authority, or that store sensitive keying material, etc. – but for our purposes, TCB refers to mechanisms underlying compartmentalization itself.

2.1 Isolation

Mechanisms for isolation vary, but will often be an operating-system (OS) process model based on a hardware Memory Management Unit (MMU). The conventional OS process model limits the scope of accidental damage from buggy programs, and also provides modest (often discretionary) isolation between users to limit the damage malicious user can cause to other users. To construct stronger isolation, other access control mechanisms are used to narrow down access to system resources. For example, across various systems, OpenSSH uses mechanisms such as `chroot`, Linux `seccomp`, SELinux [25], Mac OS X [41], and Capsicum [42] to restrict the OS facilities available to its compartments. The literature sometimes refers to a constrained process as a *sandbox*, but we prefer the term *compartment* as 'sandbox' is often taken to refer to the near total isolation of a whole application (or component) rather than being part of a more nuanced decomposition in which processes may be delegated more subsets of (perhaps quite powerful) rights.

Pattern	Description
Sandboxing	Compartments handling untrustworthy data or executing untrustworthy code are delegated minimal rights – e.g., for web-page rendering [19, 13, 28, 40, 37].
Assured pipelines	A series of linked compartments perform staged processing of data while limiting the access of (and exposure of) compartments in the chain – e.g., when processing protocol stages between two firewall interfaces [8].
Horizontal compartmentalization	Compartments contain multiple instances of the same code operating on different data instances – e.g., one sandbox for each image processed [43].
Vertical compartmentalization	Compartments encapsulate different layers when performing a task – e.g., isolating web-page download from rendering [43].
Temporal compartmentalization	Reuse of compartments over time is avoided to prevent loss of confidentiality for prior tasks, and loss of integrity or availability for later tasks [43].
Work-bounded compartmentalization	Compartments are permitted to perform bounded work, limiting potential denial-of-service impact (e.g., following a control-flow exploit) [43].
Library compartmentalization	Compartmentalization occurs entirely 'behind' an existing API, allowing a library to isolate its processing while leaving programming and binary interfaces unperturbed – e.g., placing compartmentalization in `zlib` rather than in `gzip` [42, 43].

Table 1: Common *compartmentalization design patterns* found across a broad range of compartmentalized systems.

Not all compartmentalization substrates are created equal. `seccomp` and Capsicum are both available to unprivileged processes, and intended to support extremely limited access for isolated processes. In contrast, `chroot` depends on OS privilege to use (requiring, for example, Chromium to have a `setuid` helper), but constrains only unprivileged processes. Further, `chroot` limits access only for the filesystem namespace (unless extended via a mechanism such as FreeBSD jails [18]), leaving (on many systems) interfaces for several forms of IPC, and also administrative interfaces, available to notionally isolated processes. Differing delegation mechanisms used to assign rights to compartments can also prove challenging for programmers: whereas Capsicum allows system calls to be *refined* on a per-file-descriptor basis, but a fixed set of permitted system calls, `seccomp` provides a configurable filter on system calls, but not the ability to refine rights on particular file descriptors. These and other differences (e.g., with respect to SELinux and Mac OS X) present a substantial challenge to the authors of portable applications, who must implement compartmentalization over different substrates, and hence may experience variable expressiveness and mitigation properties.

2.2 Controlled communication

Controlled communication allows compartments to communicate both with the OS and one another. This communication takes the form of a *call gate*: a call between compartments that provides mediated access to a well-defined set of entry points in a piece of code running with different privileges. This allows subsets of global system rights to be delegated to compartments, providing access to data or storage required for computation (e.g., a web-browser cache), or the ability to connect sockets to remote systems. Also important are intra-application rights; for example, different compartments within applications might have different exposure to malicious data, with those protecting access to critical information (e.g., user payment details or private keys) providing only narrowly constrained interfaces to compartments at greater risk (e.g., those rendering web pages).

A key concern in compartmentalized systems is the explicit or implied delegation of data rights, which may be considered in three classes: OS rights that represent access to resources outside the compartmentalized application (e.g., an open file descriptor), rights that represent the ability to communicate within the compartmentalized application (e.g., a local domain socket allowing message passing with another compartment), and data itself (which may flow via either of these mechanisms). The initial and ongoing configurations of these rights and data are critical to reasoning about the security state of an application (and to the protective properties of compartmentalization) as any communication right not delegated to a compartment is an attack surface avoided, in terms of potential exploits to gain privilege, and also opportunities for denials of service. Data itself presents a number of problems from the perspective of confidentiality and integrity. When a compartment has been compromised, it may have current access to confidential data, either in its memory space or via rights it holds, as well as residual data from earlier execution. Likewise, a compromised compartment affects the integrity of future computations performed in the compartment, which in turn impacts the confidentiality and integrity of data that may later be received.

2.3 Compartmentalization patterns

Compartmentalization is a general technique that configures a network of communication and delegated rights between "cuts" in an application. Many compartmentalized applications turn out to implement similar *portable* patterns. The *sandboxing pattern*, for example, delegates as few rights as possible to compartments processing high-risk data – e.g., a web-browser renderer that is able to send data to the display and receive limited input. By delegating only limited rights to the sandbox – e.g., a connected socket and output file rather than global access to the filesystem – vulnerabilities in the sandboxed code, such as bugs in TLS processing, can be mitigated. Table 1 illustrates a set of common patterns [43]. We expect many further design patterns will emerge as compartmentalization sees further deployment.

2.4 Compartmentalization spaces

Compartmentalization spaces, illustrated in Figure 1, are a key concept: applications may be partitioned in many different ways, requiring compartments to have more or fewer rights, and distributing both code of varying risk, as well

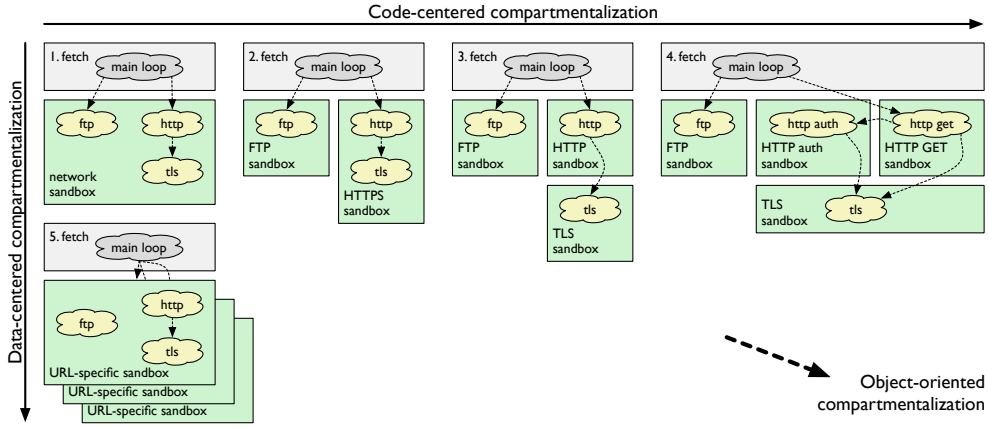


Figure 1: `fetch` and `libfetch` can be compartmentalized along different dimensions, offering different tradeoffs.

as access to data of varying degrees of risk and sensitivity, around the compartment graph. The figure illustrates two possible dimensions of further refined compartmentalization for FreeBSD's `fetch`/`libfetch`, a command-line HTTP/FTP file retrieval tool and supporting library: *data-centered compartmentalization*, in which different instances of data, processed by the same code, will be isolated from another; and *code-centered compartmentalization*, in which different code components are isolated from one another. Combined, these may be viewed as *object-oriented compartmentalization*, which provides a more granular decomposition. For `fetch`, data-centered compartmentalization might process the contents of different sites, or even different URLs, in their own sandbox, preventing cross-contamination with respect to confidentiality, integrity, and availability. Code-centered compartmentalization also offers benefits: by isolating different phases of TLS, vulnerabilities granting access to a TLS session key need not, for example, provide the attacker access to a client-side certificate.

Different points in an application's compartmentalization space offer markedly different correctness, complexity, security, and performance. Correctness is particularly impacted by *distributed-system problems*: data replicated between compartments may fall out of sync, leading to bugs. Greater compartmentalization often implies greater complexity, due to the need to select suitable rights to delegate to each compartment, but also because compartments may not naturally align with an application's class structure. Security is a key concern: vulnerabilities are mitigated only if rights assigned to compartments are limited. Predicting the location of future vulnerabilities is an art rather than a science, but notions of *risky* code (e.g., string parsing in C), and also the history of past vulnerabilities in code, often motivate software components in sandboxes. For example, it is common to place compression and video CODECs in sandboxes. Finally, performance is also critical: selecting "unnatural" cuts within applications may lead to substantial performance overheads resulting from frequent domain crossings or the need to copy (rather than delegate a right to) data. The latter may be due to mismatches between API design and rights delegated between compartments.

The effort required to compartmentalize a program means that adopting a particular point in the design space is a substantial commitment. Once a point is implemented, some further points become more accessible – e.g., further subdivision of existing compartments, or those that introduce greater data-centered granularity – but most will now require greater application refactoring.

3. THE SOAAP ANALYSIS TOOL

Application compartmentalization is a powerful vulnerability mitigation technique, but it can require substantial program refactoring and careful reasoning about interactions with the underlying isolation and communication substrates. To help developers explore the implications of current or planned compartmentalization more easily, we have developed Security-Oriented Analysis of Application Programs (SOAAP), a tool implemented using LLVM [22]. Figure 2 illustrates the SOAAP workflow. SOAAP accepts, as input, C/++ source code annotated with *compartmentalization hypotheses*. The annotations may originate with the code author or, commonly, a third party retrofitting code with compartmentalization after vulnerabilities are discovered. They may include:

- Code that should run in a sandbox. This may be entire functions or a portion of a function.
- Global state that should be accessible to sandboxes together with the types of accesses allowed.
- File descriptors that sandboxes can access and what operations are allowed on them.
- System calls that sandboxes can perform.
- Call gates available to sandboxes for performing privileged operations.
- Information-flow restrictions – for example, that certain data should not leak out of, or into, a sandbox.
- Information-flow relaxations for declassification and explicitly-permitted flows, e.g., transmitting a password hash for authentication.
- Performance overhead goals.
- Past-vulnerability information.

SOAAP also takes descriptions of sandboxing platforms that capture the protection provided – e.g., which system calls are allowed within a sandbox, at what granularity file accesses can be restricted, and whether sandboxes are completely isolated from one another or inter-sandbox communication might be permitted. These descriptions allow tradeoffs to be compared across different sandboxing technologies.

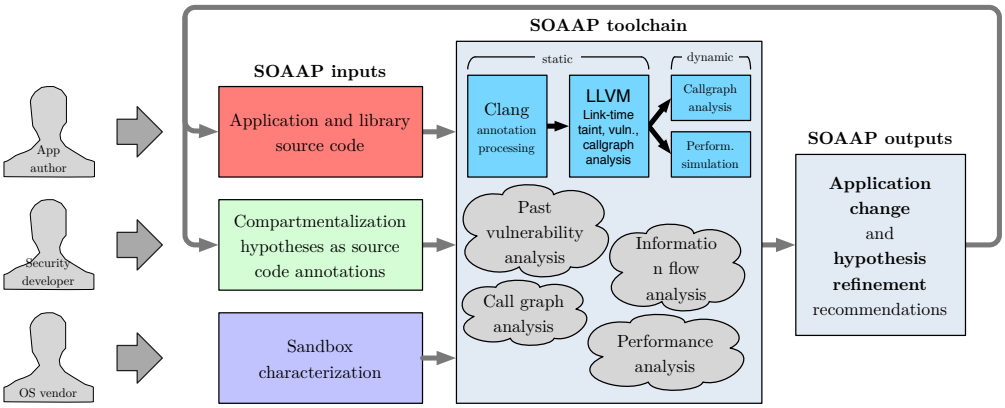


Figure 2: SOAAP engages the developer in an iterative cycle of hypothesis development and testing.

SOAAP uses LLVM's Clang front end to compile the annotated program into LLVM's *intermediate representation* (IR). Source files are compiled to individual IR files before being statically linked together into a single application IR file for analysis. SOAAP assumes a closed world, meaning that all dependencies whose behavior should be analyzed must be statically linked with the main program's IR. Most of SOAAP's analysis is static, to check whole-program behavior. We model the structure and behavior of the program by building a compartmentalization control- and data-flow graph. This captures the distributed nature and different security domains that would be present in the proposed compartmentalization. SOAAP performs dynamic analysis to evaluate performance prior to implementation by simulating sandbox management and cross-domain communication costs to measure execution overhead. If this overhead exceeds annotated performance goals, a warning is given. Unlike static analysis, the results of performance simulation will typically be sensitive to input parameters; representative results require multiple runs with representative workloads.

The SOAAP workflow engages the programmer in an iterative dialog as the compartmentalization approach is refined: programmers may initially annotate sandbox boundaries and then add or remove these and other annotations as they learn more about the compartmentalization. This cycle is comparable to the regular iteration of a developer correcting warnings through successive compiler passes. SOAAP can also be used in a continuous integration environment to validate that changes in program structure and compartmentalization do not introduce regressions over the longer-term application life cycle. SOAAP may not identify all possible shortcomings in a proposed approach, and it cannot make design decisions, but it can help developers express compartmentalization choices and understand their impacts.

3.1 SOAAP workflow example

To demonstrate SOAAP's workflow, we annotate and analyze a simple program that decrypts and decompresses files (see Listing 1). Passing `-l` lists only archive contents.

3.1.1 *Specifying sandbox boundaries*

Developers begin by annotating code that should run within a sandboxed compartment, the kind of sandbox, and when it should be created. For our example, both decryption and decompression are known to be frequent sources of security vulnerabilities. One possible compartmentalization isolates the `dec2()` function, which would prevent an attacker from gaining access to the global filesystem namespace. We annotate a persistent sandbox, i.e., a reusable global sandbox, with the name `dec2`:

```
int l_flag = 0; // list file contents

int main(int argc, char** argv) {
  l_flag = read_arg("l");
  char* in, out;
  while ((in = process_next_file())) {
    out = in_to_out(in);
    int ifd = open(in, O_RDONLY);
    int ofd = open(out, O_WRONLY);
    dec2(ifd, ofd);
  }
  return 0;
}

void dec2(int ifd, int ofd) {
  // decrypt input to tmp file
  char key[256];
  read_stdin("Password: ", key);
  // ...
  char* tmp = tmp_file();
  int tfd = open(tmp, O_RDWR);
  // ...
  read(ifd, buffer, buffer_size);
  // ...
  if (l_flag) {  // list contents
    // ...
  } else {
    // decompress tmp to output file
    // ...
  }
}
```

Listing 1: Example program that decrypts and decompresses files. Passing `-l` allows only listing the archive contents.

```
__soaap_sandbox_persistent("dec2")
void dec2(int ifd, int ofd) ;
```

We also annotate that this sandboxed compartment will be created at the start of `main()`, which allows SOAAP to understand the implications of sandboxes created via `fork()`. The subsequent sections present some of SOAAP's output.

3.1.2 *Finding data dependencies*

An initial run of SOAAP identifies that a global variable is read by the `dec2` sandbox:

```
* Sandboxed method "dec2"
```

```
* [dec2] read global variable "l_flag"
* but has not been granted the right to
+ Line 70 of file dec2.c
```

The programmer must determine whether such accesses are intended. If not, then the sandbox scope may need to be reassessed or code refactored. In this case, the access is intended, so we inform SOAAP through an annotation:

```
__soaap_var_read("dec2")
int l_flag = 0; // list file contents
```

Running SOAAP again now identifies a potential data-consistency bug that arises because `l_flag` is initialized to a command-line argument *after* the compartment is created:

```
* Write to shared variable "l_flag" in
* method "main" may not be seen by the
* sandboxes: [dec2]
+ Line 4 of dec2.c
```

If the compartment were realized as a UNIX process, a bug would result: `fork()` copies global variables from the parent to the child, and modifications to globals after the fork will not be seen in the compartment. The developer must thus ensure, when implementing this compartmentalization, that the sandbox receives the correct value of `l_flag`. Otherwise, the archive will be decompressed even when the user just wants to list the contents! This error is similar in nature to that we discovered during the original Capsicum-based compartmentalization of `gzip`, in which `gzip` performed faster with sandboxing than without. This unlikely result was due to the compression level not being propagated to the sandboxed child, leading to lower (hence faster) compression being applied [42].

3.1.3 Finding information leaks

The `dec2` sandbox handles sensitive keying material. An important security goal might be to ensure that it cannot leak out of the sandbox, which we can express as:

```
char key[256] __soaap_private;
```

SOAAP informs us that our key could leak through several `extern` library functions whose behavior we know nothing about. If we are certain these functions are safe, we can annotate them as such. SOAAP also identifies that reuse of the sandbox could leak an earlier key, as the key buffer is not scrubbed. One solution would be to zero out the buffer before returning from the sandbox; another might be to use an ephemeral sandbox instead. The former prevents against accidental leakage, but could be bypassed by an attacker able to compromise the sandbox. The latter approach provides more security, but at the cost of having to create and destroy sandboxes more frequently.

3.1.4 Finding required privileges

SOAAP uses platform descriptions to reason about sandbox restrictions, allowing the programmer to understand what can be executed by each sandbox. For example, `chroot()` sandboxes are allowed to perform any system call, whereas `seccomp` permits only `read()`, `write()`, `sigreturn()` and `exit()`. Capsicum may require explicit capabilities on file descriptors to be able to perform operations on their corresponding files. We tell SOAAP to model Capsicum behavior with our example, and it reports:

```
* Sandbox "dec2" performs system call
* "open" but it is not allowed to, based
* on the current sandboxing restrictions.
+ Line 35 of file dec2.c

* Sandbox "dec2" performs system call
* "read" but is not allowed to for the
* given fd arg.
+ Line 45 of file dec2.c
```

The first warning is specific to the Capsicum model: opening the temporary file will not work in a Capsicum sandbox. The second is more general, indicating that `read()` will fail on the file descriptor passed into the sandbox as an integer, because the parent and child process will have different file-descriptor tables. We can declare that we will delegate the read capability by annotating the input file descriptor with `__soaap_fd_permit(read)`.

3.1.5 Past vulnerabilities and supply-chain attacks

The number of mitigated past vulnerabilities is a useful measure for how effective a proposed compartmentalization might be. Annotating past vulnerabilities is not only worthwhile documentation but SOAAP can use these annotations to inform which are mitigated. SOAAP is able to enumerate the rights an attacker would gain if another vulnerability were to be exploited there. We annotate a fictitious CVE in the decryption portion of our example:

```
// decrypt input to tmp file
// ...
__soaap_vuln_pt("CVE-XXXX-YYYY")
```

SOAAP is now able to report on the effects of compartmentalization directly, showing the leaked rights:

```
*** Sandboxed function "dec2" [dec2]
*** has past-vulnerability annotations for
*** [CVE-XXXX-YYYY]. Another vulnerability
*** here would not grant ambient authority
*** to the attacker but would leak the
*** following restricted rights:
+++ Call [read] on file descriptor "ifd"
+++ Call [write] on file descriptor "ofd"
...
```

Relevant vulnerabilities are not limited to just the application being annotated but also its dependencies, as exploiting them would also give the attacker control of the application. SOAAP allows hypothesizing the occurrence of vulnerabilities in specific libraries or a particular vendor's code (so-called *supply-chain trojans*):

```
$ soaap --vulnerable-libs=libc ...
$ soaap --vulnerable-vendors=badvendor ...
```

SOAAP will again provide feedback on rights that might be leaked by the application. We provide details about this in our technical report [14].

3.2 SOAAP's analysis

3.2.1 Sandbox-platform descriptions

An important SOAAP design goal is to aid in the development of *portable* compartmentalized applications. As discussed in Section 2, different OSes provide markedly different isolation primitives with different security properties that can impact the semantics of compartmentalization. SOAAP's *platform descriptions* identify the functionality and protection properties provided by platforms such

as Capsicum and `seccomp`. The descriptions, implemented as C++ classes passed to the SOAAP analysis, also include performance and communication models.

3.2.2 Data-flow analysis

SOAAP implements a data-flow framework that can perform both *may* and *must* as well as *flow sensitive* and *flow-insensitive* analysis, depending upon the type of analysis and precision required. It uses the control-flow and call graphs computed by LLVM. For information flow analyses, such as for tracking sensitive data or file descriptors, SOAAP tracks explicit flows along define-use chains. Analyses are *sandbox sensitive*, meaning that data-flow propagation within one sandbox is distinct from that of another, even if they contain overlapping code. This enables us to model the different security domains of proposed compartmentalizations.

One of the challenges in analyzing C/C++ programs is that function pointers and polymorphism mean that the targets of a function call can be unclear. For static analysis to be safe, it must approximate all possible behaviors of the program, which means knowing which functions will be called. One approach is to assume that any plausible function could be invoked (i.e., a function pointer could refer to any function that has its address taken, or any instance method in the case of C++ – both assuming type compatibility). However, this can be overly conservative, and lead to imprecise analysis results.

SOAAP infers function-pointer targets by tracking assignments, and also allows the programmer to explicitly annotate callees. For C++, it builds a static class hierarchy using static type information obtained from Clang's AST and vtable metadata present in the LLVM IR. SOAAP then performs Class Hierarchy Analysis (CHA) [10] to calculate the set of potential targets of a virtual method call, i.e., by finding all method definitions in the class hierarchy rooted at the receiver object's static type.

3.2.3 Performance simulation

SOAAP instruments an application at relevant points based on the hypothesis. For a persistent sandbox, we fork a single process when the `__soaap_create_persistent_sandbox` annotation is reached. For ephemeral sandboxes, we fork and terminate a process at the start and end of the entry point function. There are other models that could be used, such as a zygote process, but we leave these for future work.

SOAAP emulates communication between compartments by sending RPC messages (implemented as synchronous messages sent via a UNIX pipe). Overhead is measured using `clock_gettime`. The performance simulation is an estimate: it will necessarily contain errors.

One source of error relates to overhead. SOAAP currently models sandbox creation as a constant cost, but true compartmentalization costs are non-linear in the number of sandboxes as contention triggers TLB and cache misses.

A second source of error is the varying cost of IPC. SOAAP simulates IPC cost by sending a simulated payload through a real IPC channel and measuring the round-trip time. This measures the cost of data transmission, but may not accurately capture scheduler interaction. When a process sends a synchronous message to another, an ideal scheduler would immediately switch to the receiving process and then switch back on reply. This policy was explicit in the Spring [15] microkernel's Doors mechanism, which was later ported to Solaris, but is not guaranteed by most UNIX IPC mechanisms; latency will increase considerably if other processes are scheduled between IPC send and reply.

The performance emulation in SOAAP is intended as a prototyping tool, giving a rough overview of whether a particular compartmentalization strategy will incur huge overheads, rather than as a modeling tool to determine exactly what the overheads will be on a specific system.

4. EVALUATION

The purpose of SOAAP is to allow developers to reason about (and evaluate) current and potential future compartmentalizations of software. We demonstrate three aspects of SOAAP: that a diverse array of practical compartmentalizations (and compartmentalization technologies) map into the SOAAP model; that a relatively sparse and maintainable set of annotations produce immediately useful results when a SOAAP analysis is applied; and that SOAAP performs and scales sufficiently well so as to be a practical addition to a contemporary software development environment, whether as part of live builds or a continuous integration setup. We therefore consider a series of practical compartmentalization case studies that explore both fresh compartmentalizations (of `fetch` and KDE's Okular), as well as existing compartmentalizations (of OpenSSH and Chromium).

We ran our benchmarks on a server with a 4-core Intel Xeon E5-1620 3.6GHz CPU, 64GB of RAM, and a 500GB SSD running FreeBSD/amd64 10.1-RELEASE-p5 [26]. We disabled hyper-threading and used four cores (passing `-j4` to `make` and `ninja`) when building.

4.1 Fetch: design-space exploration

FreeBSD's `fetch` application is an ideal candidate for compartmentalization. It performs risky operations such as URL/HTTP header parsing and implements TLS via OpenSSL. It also runs with full user privileges, often the system superuser. Finally, `fetch` may hold sensitive data, such as credentials for file servers and proxy servers. With many potential vulnerabilities and mutually untrusting parties, `fetch` encapsulates much of the design-space tradeoff of monolithic applications such as mail readers, web browsers, and office suites. We use SOAAP to explore two points in the design space space: compartmentalizing URL parsing and executing all network code in a single compartment. Using SOAAP, we validated our hypotheses and simulated performance, then implemented the actual compartmentalizations and compared their behavior with SOAAP's predictions.

4.1.1 URL parsing

URL parsing is a small fraction of `fetch`'s code, but it is a well-known source of vulnerabilities (e.g., CVE-2010-1663, CVE-2010-2179 and CVE-2012-0641). This makes it an excellent target for incremental, greedy compartmentalization. We expressed this hypothesis by annotating the function `fetchParseURL()` with `__soaap_sandbox_persistent` and the `url` struct's password field with `__soaap_classify`.

An initial run of SOAAP warns us about global variable accesses. Checking and annotating these global variables as being accessible to the URL parser, SOAAP suggests we should propagate the value of the global variable `fetchLastErrCode` back to the privileged parent, as well as the value of `fetchDebug` to the sandbox. SOAAP also warns about sharing sensitive state between sandbox invocations:

```
* Variable "u" (fetch.c:345) contains
* sensitive information, but is not
* erased before leaving the sandbox.
* Reusing the sandbox could allow
* another request to access  this data.
```

This compartmentalization is fairly straightforward, as the sandbox does not require any rights to execute. We must only ensure that state is propagated between the sandbox and privileged parent, and to scrub sensitive data before leaving the sandbox. An implementer would also have to add cross-domain interactions, including marshalling the URL string and unmarshalling the returned struct.

We simulate performance and compare it with actual compartmentalized performance in Section 4.1.3.

4.1.2 Networking

We also explored the threat model of a malicious server exploiting HTTP header parsing or OpenSSL to gain elevated privileges. This can be mitigated by executing network code in a persistent sandbox. Again, we wish to ensure that sensitive data cannot leak out of the sandbox.

In addition to data dependencies, SOAAP warns us about system calls that are performed within the sandbox but which would not be allowed under Capsicum by default:

```
* Sandbox "net" performs system call
* "connect" but it is not allowed to based
* on current sandboxing restrictions
```

Capsicum does not allow a sandbox to call `connect()`, nor can it be delegated as a capability right. The programmer must proxy such calls to the privileged parent or else reconsider sandbox boundaries. If we rerun SOAAP specifying `chroot` as the sandboxing technique, this warning is not produced. This reflects a tension between security and code complexity: a less invasive change also gives less security.

SOAAP also warns of potential private data leaks through `extern` functions. The first of these leaks is unintentional and the second is intentional:

```
* Sandboxed method "fetchParseURL"
* executing in sandboxes: [net] may leak
* private data through the extern function
* "fprintf"


* Sandboxed method "fetch_writev"
* executing in sandboxes: [net] may leak
* private data through the extern function
* "SSL_write"
```

The first is an illustration of a common but infrequently-considered source of leaks [29]. The second warning can be silenced by annotating the intentional flow of data with a declassify-data annotation (`__soaap_declassify`).

4.1.3 Evaluating performance simulation

We evaluate SOAAP's performance simulation by comparing it with actual compartmentalized performance. Figure 3 shows a graph with this comparison.

We manually compartmentalized `fetch` according to our hypotheses, using Capsicum as the sandboxing platform. We fork a persistent sandbox at the start of `main()` and use UNIX IPC to send data between the privileged parent and the sandboxes. For the `net` sandbox, we proxy the privileged calls identified by SOAAP: `connect()`, `fopen()`, `stat()`, `utimes()`, `mkstemps()`, `rename()`, `symlink()` and `unlink()`.

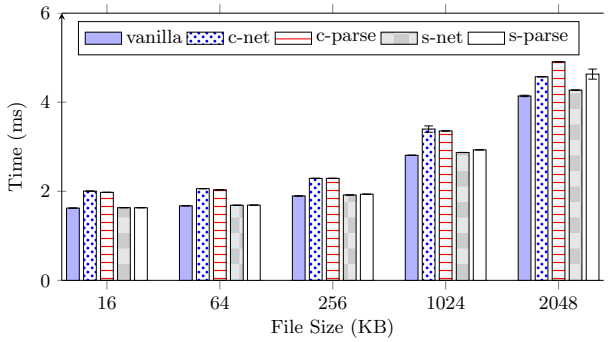


Figure 3: Overhead of compartmentalization vs. file size.

We tested our compartmentalization by using the `ktrace` tool to check for benign attempts at violating the sandbox policy. To our surprise, we saw violations that were not detected by an early version of SOAAP:

```
CALL  open(0x801c29100,0<O_RDONLY>,...)
CAP   restricted VFS lookup
RET   open -1 errno 94 Not permitted in
 capability mode
```

These violations result from calls to `getservbyname()`, a function that returns a service entry for a given name and protocol, and `timegm()`, a function that converts a time value to UTC. The implementation of these functions on FreeBSD read the network services file and UTC time zone file respectively on first call. This is a subtle failure, as any call before sandbox creation to functions that rely on network services or the time zone will cause `libc` to cache copies of them, making subsequent calls safe even after sandbox entry.

Other implementations of `libc` make `timegm()` safe to call at any time. For example, Solaris `libc` implements locale support via shared libraries, with the default C locale and UTC time zone information compiled into `libc` by default. To address this issue, we extended the platform description with information about sandbox-safe functions.

In Figure 3, compartmentalized versions have the prefix c- and SOAAP simulations are prefixed with s-. The graph shows that SOAAP's simulations are 0.4–0.6 ms less than the actual performance. This could be due to not simulating the compartmentalized code in a separate process and thus incurring less context-switching overhead. One surprising result is that the `c-net` compartmentalization is faster than `c-parse` when downloading files of size 2048 KB, despite performing significantly more IPC. Using manual source code instrumentation as well as FreeBSD's `dtrace`, we have been able to determine that this performance anomaly is disk-IO–related and not associated with either the sandboxing mechanism or the decomposition strategies implemented. SOAAP's simulation, executed on the same system, replicated this anomaly. This unexpected outcome shows that performance emulation, rather than static model-based prediction, is able to capture non-deterministic performance effects caused by hardware or the OS.

4.2 Okular: complex compartmentalization

The fetch case study demonstrated the SOAAP tool in a simple application. We now investigate using SOAAP to add compartmentalization to something much larger: Okular, the document viewer of the KDE desktop environment. Okular is around 80KLOC (plus 4MLOC from external libraries that were also analyzed).

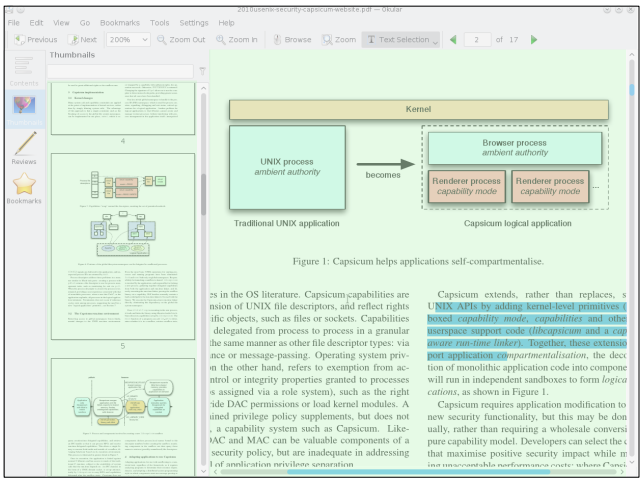


Figure 4: Okular compartmentalization: grey code executes with ambient authority; green code is sandboxed.

Okular has a modular structure with rendering plugins ("renderers") for each supported document format. We decided to use this plugin interface to draw the compartmentalization boundary, which allows us to support both sandboxed and unsandboxed plugins. Okular currently has 16 different renderers and we do not wish to have to add sandboxing support to all of them simultaneously. This highlights one of the key requirements for a tool such as SOAAP: aiding *incremental compartmentalization*. For this case study we focused on the commonly-used PDF renderer. This renderer is implemented using the `libpoppler` library, which has had 38 known vulnerabilities [35] in the last 10 years. Of these, 19 are code execution vulnerabilities.

In the case of `fetch`, we were able to annotate a single function as the sandbox entry point. This was not possible here because Okular dynamically loads renderer plugins at run time – which equates to creating a single C++ object that implements the `Okular::Generator` interface. Our proposed sandbox scope would be to include every member function of that class and the base classes. As this was not easily expressible with the existing SOAAP annotations, we introduced a new `__soaap_sandboxed_class(name)` annotation to annotate the `PDFGenerator` class.

One interesting issue found by SOAAP was that the KDE translation system may lazily load a shared library when translating messages into certain languages. The `dlopen()` call will fail when in the sandbox, however as the library is not required for most languages, this error is unlikely to be observed when using dynamic analysis or testing. This problem of lazy initialization was also encountered when compartmentalizing `fetch` (see Section 4.1.3) for the calls to `getservbyname()` and `timegm()`.

These findings, as well as the broken `rhosts` support in OpenSSH (see Section 4.3), show that static analysis with SOAAP can find compartmentalization errors that otherwise might go unnoticed for years. However, being able to find compartmentalization bugs caused by lazy initialization can come at the cost of an increase in false positives. When analyzing Okular, SOAAP showed 9 false positives due to lazy initialization that performs I/O but which had already been executed before entering the sandbox. There was also one such finding that was a legitimate issue. The PDF renderer would lazily attempt to load a configuration file from within the sandbox. These issues demonstrate that lazy initialization is a real anti-pattern when compartmentalizing applications as it must be ensured that all the initialization code has run before the sandbox is entered.

By moving `libpoppler` into a sandbox we have not only removed code with a history of vulnerabilities from the TCB but also reduced the total amount of code that runs at full privileges by over 180,000 lines – which is more than the whole Okular source tree with 80,000 lines.

We have now submitted our changes, created with the benefit of SOAAP, to the upstream Okular project for review. The changes amount to fewer than 2,500 lines. Of these 2,500 lines, 1,800 are adding new classes required for proxying the calls to `Okular::Generator` functions to the sandbox running in a separate process. Almost all other changes are related to moving classes from `.cpp` files to `.h` files so that the serialization code can access the required fields. The only changes to the existing source were 30 lines to enable loading of sandboxed renderers and 100 lines to the `PDFGenerator` class. Figure 4 provides a visualization of the implemented compartmentalization. Green indicates code within a sandboxed compartment and grey is ambient code.

Compared to the ad-hoc nature of the OpenSSH privilege separation which introduced changes across many files, the use of object oriented interfaces makes it possible to have almost all code related to compartmentalization in one location (in this case the proxy class that implements the required interface). This makes it much easier to reason about the correctness of the compartmentalization than in a large C program without a modular design.

4.3 OpenSSH: long-term maintenance

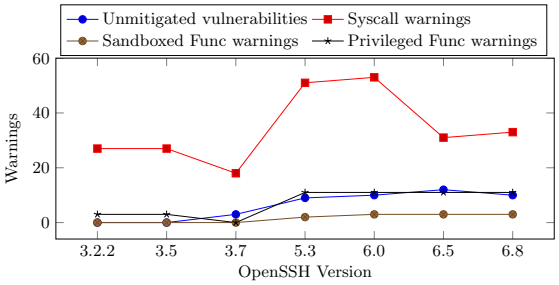
OpenSSH [36] has had privilege separation since 2002, with gradual refinements over time, making it an excellent case study for long-term compartmentalization maintenance using SOAAP. Of particular concern is the risk of silent security regression: previously mitigated vulnerabilities might become unmitigated, confidentiality goals might no longer be met, or performance overhead might become intolerable. A recent example [2] is a bug discovered in NetBSD's pkgsrc patch for `sshd` that mistakenly disabled privilege separation on all platforms except Interix. The intention was the opposite but this error was only detected nine years later!

OpenSSH uses a two-stage sandboxing strategy: a completely unprivileged process prior to authentication called `preauth` followed by a process running with the rights of the authenticated user, called `postauth` [36]. Table 5a shows several releases of OpenSSH, from the latest version (6.8) back to the first privilege-separated version (3.2.2). For each version, we annotated the sandboxing boundaries, past vulnerabilities, confidentiality requirements, cross-domain communication, assertions of functions that should only run privileged and those that should only run sandboxed.

We observed that OpenSSH's compartmentalization has not changed significantly over time, supporting our supposition that such maintenance is difficult in practice. We also discovered functionality that had silently regressed due to compartmentalization. For example, by applying SOAAP to OpenSSH v3.2.2 we discovered that the `preauth` sandbox was performing multiple `stat()` calls on `rhosts` files inside `auth_rhosts2_raw()`. As this sandbox runs in a `chroot` environment, these accesses would always fail, breaking `RhostsAuthentication` when compartmentalization was en-

Ver.	Year	LOC	Functions	Annotations			
			P/S/P&S	CDC	Assert	Vuln.	All
3.2.2	2002	43.6K	197/454/84	80	64	23	192
3.5	2002	45.4K	200/454/83	88	67	34	211
3.7	2003	48.7K	212/463/90	113	73	34	247
5.3	2009	63.8K	367/667/182	117	75	58	287
6.0	2012	73.9K	433/691/196	117	76	65	298
6.5	2014	82.6K	578/764/251	117	76	68	308
6.8	2015	96.7K	617/880/276	117	74	65	300

(a)



(b)

Figure 5: (a) Table showing OpenSSH code changes between versions. P: privileged; S: sandboxed; CDC: cross-domain communication. (b) Graph showing the number and types of SOAAP warnings across OpenSSH versions.

abled. This was never fixed, but `rhosts`-based authentication was already a vestigial feature by the time OpenSSH came into widespread use and it was removed in v3.7.

We were able to carry our the initial annotations forward with only trivial merge conflicts. However, shown in Table 5a, SOAAP reports that the ratio between total code and code executing in sandboxes has decreased from 69% to 59% — the TCB has been getting larger. One major change is the use of multiple sandboxing platforms such as `seccomp-bpf` (added in 6.0) and Capsicum (added in 6.5).

We also examined the change in SOAAP's warnings across OpenSSH versions, shown in Figure 5b. We looked at unmitigated vulnerabilities, system call warnings and functions that execute outside their intended domains. System call warnings tell us which system calls are not permitted by a given sandboxing platform. SOAAP correctly warned about `stat()` calls, and these warnings disappeared when `rhosts` support was removed in 3.7. We also see the warnings drop when Capsicum support was added in 6.5. Along with the rise in privileged code, SOAAP shows us that there is a corresponding rise in the number of unmitigated vulnerabilities.

SOAAP also gave insights into how the execution of code annotated as being privileged or sandbox-only has changed over time. Sandbox-only code (likely to be riskier) executes almost only in sandboxes whereas privileged code is increasingly being executed in sandboxes. If the security policy for the sandbox is not correctly defined, the sandbox could perform operations it is not supposed to. Conversely, if the policy is correct then privileged operations will fail.

4.4 Chromium: analysis scalability

We tested SOAAP's scalability by applying it to the Chromium web browser [37], a large, complex, compartmentalized application with a history of security vulnerabilities and a variety of vulnerability mitigation techniques. We used SOAAP's risk annotation and data-flow analysis capabilities to investigate the separation of "risky" code and data from each other. We were able to observe at least one instance in which risky code could be exposed to risky data, potentially leading to critical security vulnerabilities as have been previously reported in released versions of Chromium.

4.4.1 Risky code

The Chromium web browser, which is the open source variant of Google Chrome, has complex security requirements and incorporates code from many sources. In addition to core browser code written by Google and community developers, Chromium incorporates code from vendors and open-source libraries with varying security track records. For instance, the WebKit rendering framework has suffered over 200 publicly-reported security vulnerabilities since it was spun off from the KHTML framework [1], and through a dedicated fuzzing effort, Google security developers found over one thousand security vulnerabilities in the FFmpeg media libraries in the span of a year [17]. Code that has previously suffered from multiple vulnerabilities often deals with the interpretation of complex data and is therefore an excellent candidate to be compartmentalized and isolated from privilege and ambient authority (see Section 2.3).

SOAAP facilitates reasoning about the exposure of risky code by allowing developers to annotate previously-vulnerable code. We identified risky code in Chromium by searching Chromium's historical bug database for security vulnerabilities in extant code. Due to constant code churn, many historic vulnerabilities applied to code that no longer exists. However, we did find 21 historic Chromium vulnerabilities that have been patched and made public but applied to extant code in the WebKit rendering engine, Skia graphics library, FFmpeg media transcoder, v8 JavaScript engine and internal HTTP or URL handling code. From these vulnerability descriptions, we annotated ten previously-vulnerable functions (`__soaap_vuln_fn`) and 18 locations within other functions that were previously vulnerable (`__soaap_vuln_pt`). These annotations are a machine-readable formalization of a common practice in the Chromium code base: referencing bug IDs from comments in source code. We found 1,803 instances of such references within C++ comments, some of which we were able to formalize with SOAAP annotations:

```
__soaap_vuln_pt("Cr issue #425035");
CHECK(HasBeenSetUp()); // crbug.com/425035
```

Annotating the locations of previous vulnerabilities allows SOAAP to statically enumerate the call graphs of risky code. Chromium is designed to execute risky code inside compartments, using platform-specific compartmentalization technologies. To analyze the effectiveness of Chromium's sandboxing, we needed to expose the details of the compartmentalization scheme to SOAAP. We re-implemented the sandboxing model using Capsicum [42] and found it to be an excellent fit for the compartmentalization models of both SOAAP and Chromium: the Linux compartmentalization requires 13,821 lines of code and Windows requires 28,239, but the Capsicum model requires only 406 lines of source code and two sets of SOAAP annotations. The first was a pair of annotations (`__soaap_sandboxed_region_start` and `__soaap_sandboxed_region_end`) within `RendererMain` to expose the extent of the sandbox. The second was

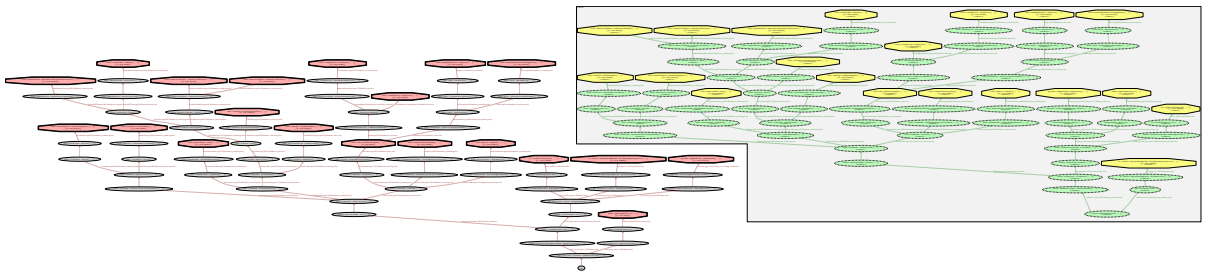


Figure 6: Simplified SOAAP-generated call graphs of unmitigated (red) and mitigated (yellow) vulnerabilities in Chromium.

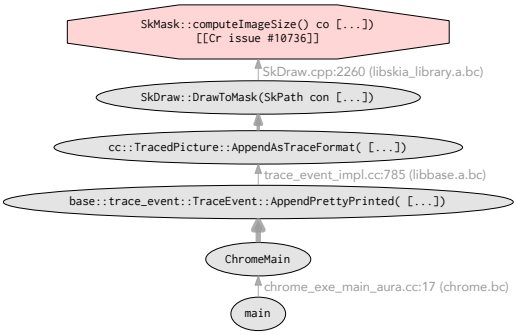


Figure 7: A simplified call graph produced by SOAAP that demonstrates how a previously-vulnerable function can be called from `main` in an unsandboxed (i.e., privileged) context.

a `__soaap_limit_fd_syscall` annotation to describe how Chromium limits a sandbox's authority on particular file descriptors according to the Capsicum model.

After annotating Chromium to describe both its risky code and its compartmentalization strategy, SOAAP was able to statically check which risky functions could be executed outside of a sandbox. The results, shown in Figure 6, are surprising at first glance: almost all of the risky functions are executed in both the privileged browser process and the least-privileged renderer process. For example, debug tracing in the privileged browser process can call a previously-vulnerable image parsing function within Skia, as shown in Figure 7. However, this static call graph does not tell the whole story of code and data risk.

A reliance on static call graphs can lead to large numbers of false-positive results. Figure 7 does show that the previously-vulnerable function `SkMask::computeImageSize()` might execute outside of the sandboxed renderer process, but if it only computes the sizes of browser chrome images distributed by Google, there is a much lower risk than if it is exposed to content from arbitrary web pages. In order to better understand the real risk of running risky *code*, we added risky *data* annotations and combined the above static call graphs with SOAAP's information flow analysis.

4.4.2 *Risky data*

In addition to annotating risky (previously-vulnerable) code, we also annotated risky data: data that is obtained from the network and which may therefore be dictated by an attacker. We annotated Chromium's `net::URLRequest::Read` method, telling SOAAP to treat the read buffer as private to the renderer sandbox:

```
bool Read(__soaap_private("renderer")
          IOBuffer* buf,
          int max_bytes, int* bytes_read);
```

We then enabled SOAAP's information flow tracking to determine all of the locations where this network-originated data was accessed outside of a sandbox. SOAAP detected 1.88 M such accesses, a number too large to manually sift for false positives. We are able to cope with this scale of problem in two ways. First, our SOAAP graph manipulation tools are able to compute useful operations on combined call/data flow graphs, such as intersections to a certain depth and leaf node filtering. For example, we can trace back a fixed number of calls or data flows from a vulnerability or private data access to compute a set of ancestor nodes, then only include that node in the new intersection graph if it shares at least one ancestor with the ancestors of another graph. This allows us to reduce the graph of all private accesses to the graph of private accesses (risky data) that occur in some proximity to a previously-vulnerable function (risky code).

The result of intersecting the vulnerability and private-access graphs is still too large for manual exploration. Such exploration was important when iteratively investigating locations for network data annotation (`net::HttpStreamParser`, `net::HttpNetworkTransaction`, `net::URLRequest`, etc.). Therefore, for initial data exploration, SOAAP is able to treat private accesses probabilistically. We found that randomly selecting 0.01% of private access allowed us to perform exploratory data analysis and rapidly identify suspicious data flows. We then filtered large graphs by leaf nodes (vulnerabilities or private accesses) to produce the simple call and data-flow graphs shown in Figures 8 and 10 (see Section A).

Using SOAAP's static call graphs of previously-vulnerable code, information flow propagation analysis, sandbox annotations, graph intersection and filtering, we were able to identify network-tainted data (risky data) flowing to previously-vulnerable functions (risky code). For example, Figure 8 depicts a highly elided call and data flow graph that shows resources loaded by URL and presented to MIME type detection code that has suffered security vulnerabilities in the past. This is precisely the type of interaction that application authors might prefer to occur within a least-privileged compartment; SOAAP makes it visible to the authors so they can take appropriate mitigation steps.

After intersection and filtering SOAAP graphs, we still encountered many false positives caused by Chromium's varied network use. For instance, Chromium's HTTP and URL handling code is used both to retrieve data from the network and to manage transport between its privileged and unprivileged components. We saw many data flows from, e.g., `net::HttpStreamParser` to code in the `remoting::protocol` namespace, and then on to previously-vulnerable WebKit code. We view this as a false-positive result: it is part of a

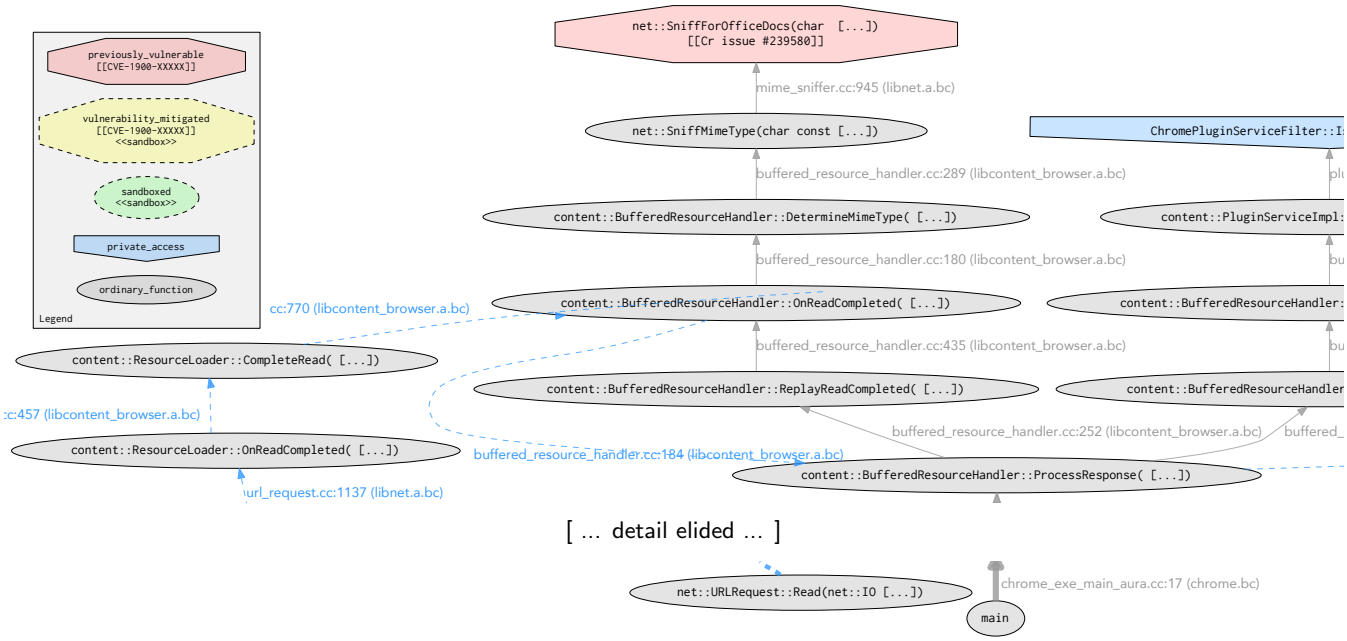


Figure 8: An example of data flowing from the network to code that has previously been vulnerable to exploitation. In this example, which has been cropped by 38 function calls and nine relevant data flows, data flows from a `URLRequest` to a `BlobURLRequestJob` to a `BufferedResourceHandler`, where it is inspected to determine its MIME type. `net::SniffForOfficeDocs` was annotated with `__soaap_vuln_pt` due to a previous critical security vulnerability (see `http://crbug.com/239580`).

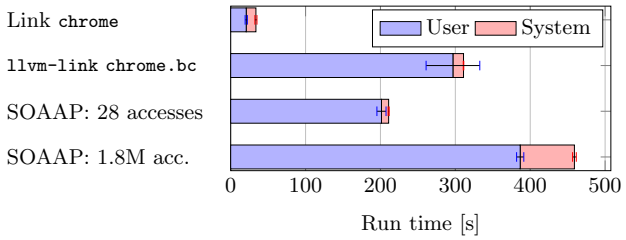


Figure 9: SOAAP's run time is similar to that of linking the Chromium bitcode using `llvm-link`. With a restricted set of private-access annotations, leading to 28 detected private accesses, it is 0.68 that of `llvm-link`. With the annotation used in Section 4.4.2, which causes 1.8 M private access warnings, the run time is 1.48 times that of `llvm-link`.

carefully-managed interaction between browser components for the purpose of enabling compartmentalization.

Such false positives could be managed by introducing more selective annotations, e.g., on `content::ResourceLoader` rather than the general-purpose `net::URLRequest`, but our goal in this work was to explore as many variations of data flowing to risky code as possible. That is, we had very little tolerance for false negatives, so we erred on the side of false positives and developed tools to cope with them.

4.4.3 *Performance*

As a whole-program analysis, SOAAP's run-time cost when targeting Chromium is comparable to other whole-program LLVM-based tasks. Figure 9 compares the run time of four whole-program compilation and analysis tasks: linking Chromium's object files and static archives into an executable ELF binary, linking its individual LLVM IR object files into a complete bitcode file with `llvm-link`, running SOAAP with some private data accesses and running SOAAP with the annotations described in Section 4.4.1.

The following measurements were performed on a system running FreeBSD 10.1-RELEASE-p16, with a ZFS ARC (adaptive replacement cache) of 20–60 GiB (out of a total of 128 GiB of system RAM) and no memory pressure: the entire working set was able to fit into the ARC rather than be retrieved from disk on-demand. Our system has dual Xeon E5-2667v3 CPUs with 32 logical processors, but SOAAP does not currently exploit their parallelism.

Figure 9 shows, as we would expect, that linking Chromium as a 985 MiB LLVM bitcode file is slower than linking a 141 MiB ELF executable. LLVM IR is a higher-level representation with explicit types and other information that is not included in machine code. SOAAP's run-time cost is either just over or just under this LLVM linking time, depending on the complexity of the data flows that SOAAP must track. Loading this 985 MiB bitcode file into memory required over 10 GiB of memory, approximately the same amount of memory required by `llvm-link`. Processing the class hierarchy, finding sandboxes and adding information about previous vulnerabilities pushed memory usage to just over 12 GiB, and adding the details of 1.5 M private accesses increased memory usage to just over 14 GiB. The difference in run time between the two SOAAP cases in Figure 9 is due to the private accesses that were detected, according to the `_soaap_private` annotations that were applied. The outcome of SOAAP's analysis was a 978 MiB JSON file that we initially explored with `jq` [12] and finally processed with our SOAAP graph manipulation tools.

Combining SOAAP's sandbox, vulnerability and data flow annotations allowed us to detect the interaction of risky code and risky data outside of least-privileged compartments, creating risk for Chromium's users. These complex analyses can be used to immediately inform application authors of potential risks and inform their decisions about where to invest compartmentalization effort. Even on a large, complex

Program	Version	LOC	Running time (secs)					
			IR	Null	Vuln.	Correct	Info	All
fetch	r252375	5K	1.17	0.05	0.05	0.06	0.06	0.06
OpenSSH	3.2.2	44K	3.8	0.3	0.3	0.3	0.3	0.3
OpenSSH	6.8	96K	8.7	1.15	1.15	1.15	1.15	1.15
Okular	e03e6f	4.2 M	2970	14.28	14.66	14.73	14.74	15.37
Chromium	42	13.1 M	7300	195	261	2156	1028	3048

Table 2: The running times of various SOAAP modes of execution for each of our case studies.

piece of software such as Chromium, SOAAP was able to produce these data in eight minutes, or approximately the time required to link Chromium as LLVM bitcode.

4.5 SOAAP tool performance

Table 2 shows the time taken to run SOAAP on each of the case studies in three different modes. These modes are selected to correspond to different stages in the development cycle for compartmentalized applications:

Vulnerability analysis checks which of the annotated vulnerabilities would be mitigated within sandboxes. This mode is used for the first cut at describing a sandbox, to check that dangerous code is not run with ambient authority.

Sandbox correctness checks whether each of the sandboxes tries to access resources (global variables, system calls, and so on) without being explicitly granted capabilities for them. This mode is used to check that all of the communication between sandboxes is explicit. Once a program passes this mode, all points that need modifying are annotated.

Information flow analysis is the final check, which ensures that sensitive information is not leaked between sandboxes. This is the final step before implementing a chosen compartmentalization strategy, which checks that the security goals are met. This also includes the same checks as the vulnerability analysis and would be run as part of a continuous integration build to check for sandboxing regressions.

We distinguish between the cost to compile to LLVM IR and the cost to perform the initial analysis setup of loading the IR and building a call graph. We represent these as the two modes **IR Compilation** and **Null analysis**.

We also focus on the Chromium case study, as the one with the longest running time, to provide more detail on where the time is being spent. Table 2 shows a breakdown of the SOAAP run according to the time spent in different phases of the analysis, generated from a profiling build of SOAAP. Note that there is some potential for optimization here, for example 10% of the total time is spent on formatting JSON output and 35% on reading in the LLVM IR. These can both be improved, for example by constructing the JSON concurrently with the main thread and by demand-loading the LLVM IR from separate files using some of the infrastructure for more efficient link-time optimization support in LLVM that is currently under development at Google [39].

5. FUTURE WORK

SOAAP has proven to be a useful tool for analyzing and compartmentalizing programs in our environment. A natural next step would be to investigate fully automated compartmentalization – which requires significantly more work than a simple analysis tool: RPC stubs must be generated, and current false positives and negatives in analysis will become actual bugs in generated code. Another valuable exploration would be wider deployment of SOAAP into the software development community; with increasing adoption of the Clang/LLVM suite in the FreeBSD community, and that community's heavy use of the Clang static analyzer as part of regular tinderboxing, it is easy to imagine a "tinder-sandbox" that regularly reviews sandboxing tradeoffs and checks that past and new vulnerabilities are properly mitigated. This notion of integrating SOAAP into a continuous integration cycle, as is done with other types of static analysis (e.g., Coverity Prevent) and testing seems promising.

6. RELATED WORK

The principle of least privilege and other security goals (such as protecting integrity, confidentiality, and availability) were enumerated in Saltzer and Schroeder's 1975 article, *The Protection of Information in Computer Systems*. Karger's 1987 article on Trojan horse mitigation [19] lays the conceptual groundwork for privilege separation, and later application compartmentalization, which became mainstream techniques applied to system-level applications in the early 2000s, with Provos's work on OpenSSH [36], and Kilpatrick's Privman [20]. Application compartmentalization is applied to user-level applications by Gong et al. in Java [13], Reis et al. in the Chromium web browser [37], and by Watson et al. in Capsicum [42] – where the focus is on intra-application security concerns rather than system privileges. All of these projects have reported difficulty in applying compartmentalization to C-language applications; we draw particularly on our own experience in developing Capsicum. The interfaces between separated components have also been studied for their security implications – in particular, those of cryptographic security APIs [4], whose lessons are also applicable to general-purpose compartmentalization. The analysis of Beurdouche et al. [6] suggests the need for further compartmentalization of the client and server sides of OpenSSL and JSSE to mitigate surprising compositional vulnerabilities.

Brumley and Song's Privtrans [9] and Bittau et al.'s Wedge [7] explore techniques for assisting programmers in identifying and exploiting compartmentalization opportunities. Privtrans takes a code-oriented view, focusing on dividing operation between privileged and unprivileged processes through program annotation, whereas Wedge relies on programmer-provided memory type information. Harris et al.'s secure programming by parity games [16] reasons about the defense characteristics of Capsicum compartmentalization, representing policies as automata.

SOAAP benefits from the object-capability philosophy explored in HYDRA [23] and rigorously extended by Miller [28]. It is also comparable to Miller et al.'s or Mettler et al.'s imposition of object-capability semantics on Java in DarpaBrowser [40] or Joe-E [27] – albeit in the uncon-

strained execution environment of UNIX processes rather than a type-safe and already object-oriented language, and so requiring the imposition of an object-oriented structure. SOAAP incorporates decentralized information flow-control techniques [34], past vulnerability information, and source-code risk analysis, allowing object-capability boundaries to align with critical security constructs in the application and its libraries. Rather than implementing separation policies, SOAAP is an exploration tool for programmers who may not fully understand the code that they are separating, recognizing the complexity of large-scale applications.

Software decomposition for robustness (and sometimes security) has a long history, seeing early exploration in microkernel designs [24], but performance concerns have discouraged widespread use. Growing compartmentalization reflects two changes: improvement in hardware performance, and a pressing need for vulnerability mitigation. Klein et al.'s SeL4 microkernel [21] offers a contemporary take on microkernel design, achieving both performance and high assurance as a compartmentalization substrate. Hardware virtualization extensions can support more efficient separation via hypervisors [33], and extensions to the process model such as Belay et al.'s Dune [5]. Research architectures such as CHERI [43] may support vastly more scalable compartmentalization in the future. SOAAP's model and analysis spans a broad range of compartmentalization substrates, and can assist developers in making safe and effective use of new facilities as they become mainstream.

7. CONCLUSION

Application compartmentalization is a key vulnerability mitigation technique that addresses a broad range of known (and unknown) attack vectors. Despite increasingly widespread deployment, it is also an approach that is rife with subtlety: we have demonstrated that developers are challenged by the need to develop and maintain compartmentalizations that adequately trade off correctness, security, complexity, and performance. SOAAP's conceptual framework, and practical tools, have assisted us in analyzing current compartmentalizations, and develop new ones, with greater understanding and confidence. SOAAP is scalable to multi-million LoC code bases, and we are hopeful that this approach will facilitate greater deployment of compartmentalization as a strong mitigation feature.

8. AVAILABILITY

For further details, refer to an extended version of this paper [14]. Our prototype is available as open source:

`https://www.cl.cam.ac.uk/research/security/ctsrd/soaap/`

9. ACKNOWLEDGMENTS

We thank Ross Anderson, David Drysdale, Úlfar Erlingsson, Somesh Jha, Steve Hand, William Harris, Angelos Keromytis, Bob Laddaga, Anil Madhavapeddy, Simon Moore, Steven Murdoch, Robert Norton, Thomas Reps, Hassen Saidi, Howie Shrobe, Arun Thomas, Munraj Vadera, Sam Weber, and Jonathan Woodruff, as well as our anonymous reviewers, for their feedback and assistance. We gratefully acknowledge Google, Inc., for its support via a Focused Research Award. This work is also part of the CTSRD, MRC2, and CADETS projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237, FA8750-11-C-0249, and FA8650-15-C-7558. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. We also acknowledge the Isaac Newton Trust, the EPSRC REMS Programme Grant [EP/K008528/1], NSERC Discovery Grant RGPIN/06048-2015, and Thales E-Security.

10. REFERENCES

[1] Apple WebKit Vulnerability Statistics. `http://www.cvedetails.com/product/10007/Apple-Webkit.html`.

[2] CVS commit: pkgsrc/security/openssh, Aug. 2015. `http://mail-index.netbsd.org/pkgsrc-changes/2015/08/14/msg128305.html`.

[3] Anderson, J. Computer security technology planning study. Tech. Rep. ESD-TR-73-51, U.S. Air Force Electronic Systems Division, October 1972.

[4] Anderson, R., Bond, M., Clulow, J., and Skorobogatov, S. Cryptographic processors-a survey. *Proceedings of the IEEE 94*, 2 (Feb 2006), 357–369.

[5] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: safe user-level access to privileged CPU features. In *Proceedings of the 10th Conference on Operating Systems Design and Implementation* (2012), USENIX.

[6] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., and Zinzindohoue, J. K. A Messy State of the Union: Taming the Composite State Machines of TLS. In *Proceedings of the IEEE Symposium on Security and Privacy* (2015).

[7] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX.

[8] Boebert, W., and Kain, R. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth DoD/NBS Computer Security Initiative Conference* (1985).

[9] Brumley, D., and Song, D. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium* (2004), USENIX.

[10] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming* (1995), ECOOP '95, Springer-Verlag.

[11] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. *Commun. ACM 9*, 3 (1966), 143–155.

[12] Dolan, S., Williams, N., Tolnay, D., Lapresta, S., Langford, W., and Gordan, A. `jq`: a lightweight and flexible command-line JSON processor. `http://stedolan.github.io/jq/`.

[13] Gong, L., Mueller, M., Prafullchandra, H., and Schemers, R. Going beyond the sandbox: An

overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the Symposium on Internet Technologies and Systems* (1997), USENIX.

[14] Gudka, K., Watson, R. N. M., Anderson, J., Chisnall, D., Davis, B., Laurie, B., Madhavapeddy, A., Marinos, I., Murdoch, S. J., Neumann, P. G., and Richardson, A. Clean Application Compartmentalization with SOAAP (extended version). Tech. Rep. UCAM-CL-TR-873, University of Cambridge Computer Laboratory, Sept. 2015.

[15] Hamilton, G., and Kougiouris, P. The Spring Nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer Usenix Conference* (1993), USENIX.

[16] Harris, W. R., Farley, B., Jha, S., and Reps, T. Secure Programming as a Parity Game. Tech. Rep. 1694, University of Wisconsin Madison, July 2011.

[17] Jurczyk, M., and Coldwind, G. FFmpeg and a thousand fixes. Google Online Security Blog, January 2014. `http://googleonlinesecurity.blogspot.com/2014/01/ffmpeg-and-thousand-fixes.html`.

[18] Kamp, P., and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference* (2000).

[19] Karger, P. A. Limiting the damage potential of discretionary trojan horses. In *Proceedings of the IEEE Symposium on Security and Privacy* (1987), IEEE.

[20] Kilpatrick, D. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference* (2003), USENIX.

[21] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an operating-system kernel. *Commun. ACM 53* (June 2009), 107–115.

[22] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (2004), CGO '04, IEEE.

[23] Levin, R., Cohen, E., Corwin, W., Pollack, F., and Wulf, W. Policy/mechanism separation in Hydra. In *SOSP '75: Proceedings of the fifth ACM Symposium on Operating Systems Principles* (1975).

[24] Lipner, S. B., Wulf, W. A., Schell, R. R., Popek, G. J., Neumann, P. G., Weissman, C., and Linden, T. A. Security kernels. In *AFIPS '74: Proceedings of the May 6-10, 1974, National Computer Conference and Exposition* (1974), ACM.

[25] Loscocco, P., and Smalley, S. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the USENIX Annual Technical Conference* (2001), USENIX.

[26] McKusick, M. K., Neville-Neil, G. V., and Watson, R. N. M. *The Design and Implementation of the FreeBSD Operating System*. Pearson, 2014.

[27] Mettler, A., Wagner, D., and Close, T. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the Network and Distributed System Security Symposium* (2010), NDSS'10.

[28] Miller, M. S. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006.

[29] MITRE. CVE-2012-0652.

[30] MITRE. CVE-2014-0160.

[31] MITRE. CVE-2014-4877.

[32] MITRE. CVE-2014-6271.

[33] Murray, D. G., and Hand, S. Privilege separation made easy: trusting small libraries not big processes. In *Proceedings of the 1st European Workshop on System Security* (2008), EUROSEC '08, ACM.

[34] Myers, A. C., and Liskov, B. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev. 31* (October 1997), 129–142.

[35] NIST. libpoppler CVEs. `https://web.nvd.nist.gov/view/vuln/search-results?cpe_vendor=cpe:/:freedesktop&cpe_product=cpe:/::poppler`.

[36] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003), USENIX.

[37] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems* (2009), ACM.

[38] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (September 1975), 1278–1308.

[39] Teresa Johnson, X. D. L. ThinLTO: A Fine-Grained Demand-Driven Infrastructure. In *EuroLLVM* (2015).

[40] Wagner, D., and Tribble, D. A security analysis of the combex darpabrowser architecture, March 2002.

[41] Watson, R. N. M. A decade of OS access-control extensibility. *Commun. ACM 56*, 2 (Feb. 2013).

[42] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (2010), USENIX.

[43] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015).

APPENDIX

A. ADDITIONAL GRAPH DETAIL

Figure 10 provides a larger example of a call and data flow graph emitted by SOAAP tools than could be included in Section 4.4. This particular graph is an example of a false-positive result that was obtained by annotating a low-level networking method that is used by several Chromium subsystems, not just communication with remote hosts.

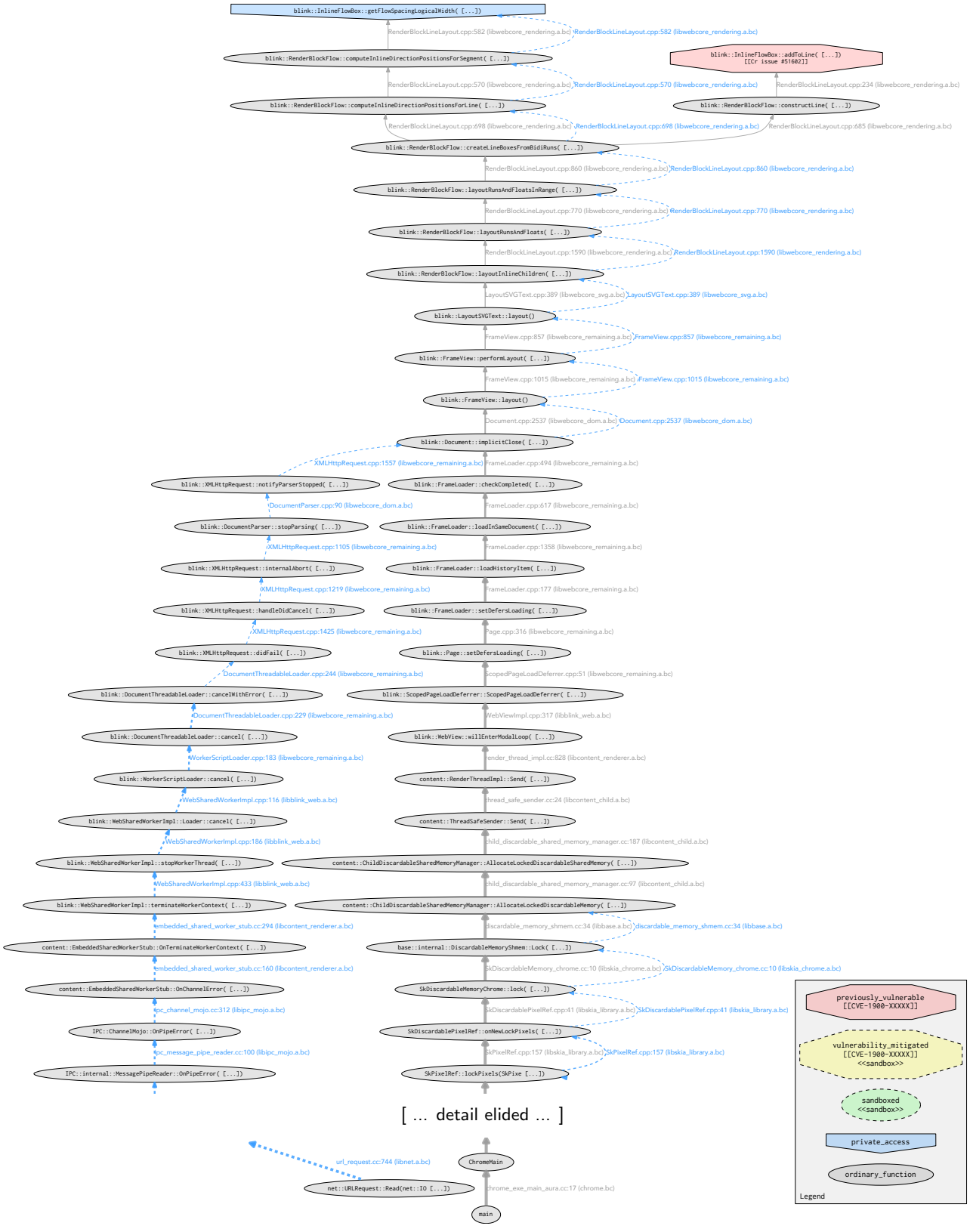


Figure 10: An example of a false positive result (cropped to fit: we have elided nine calls and eight flows). Here, data flows from `URLRequest::Read` towards previously vulnerable code, which we deem to be "risky" code. However, these data flows go through IPC interfaces that are used for communicating between the browser and renderer processes. Rather than exposing privileged code to arbitrary data, this data flow passes highly constrained data for the purpose of enabling compartmentalization.