Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software

Hugo Lefeuvre[†], Vlad-Andrei Bădoiu[▽], Yi Chien[‡], Felipe Huici[∞], Nathan Dautenhahn[‡], Pierre Olivier[†]
[†]*The University of Manchester,* [▽]*University Politehnica of Bucharest,* [‡]*Rice University,* [∞]*Unikraft.io*

***Abstract*—Least-privilege separation decomposes applications into compartments limited to accessing only what they need. When compartmentalizing existing software, many approaches neglect securing the new inter-compartment interfaces, although what used to be a function call from/to a trusted component is now potentially a targeted attack from a malicious compartment. This results in an entire class of security bugs: Compartment Interface Vulnerabilities (CIVs).**

This paper provides an in-depth study of CIVs. We taxonomize these issues and show that they affect all known compartmentalization approaches. We propose ConfFuzz, an in-memory fuzzer specialized to detect CIVs at possible compartment boundaries. We apply ConfFuzz to a set of 25 popular applications and 36 possible compartment APIs, to uncover a wide data-set of 629 vulnerabilities. We systematically study these issues, and extract numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them. We stress the critical importance of CIVs in compartmentalization approaches, demonstrating an attack to extract isolated keys in OpenSSL and uncovering a decade-old vulnerability in sudo. We show, among others, that not all interfaces are affected in the same way, that API size is uncorrelated with CIV prevalence, and that addressing interface vulnerabilities goes beyond writing simple checks. We conclude the paper with guidelines for CIV-aware compartment interface design, and appeal for more research towards systematic CIV detection and mitigation.

I. Introduction

The principle of least privilege has guided the design of safe computer systems for over half a century by ensuring that each unit of trust in a system can access only what it truly needs to fulfill its duties: in this way, system designers can proactively defend against unknown vulnerabilities [65]. Software compartmentalization is a prime example where unsafe, untrusted, or high-risk components are isolated to reduce the damage they would cause should they be compromised [50].

Recent years have seen the appearance of an increasingly large number of new isolation mechanisms [10], [4], [3], [65], [53], [45] that enable fine-grained compartmentalization. This resulted in compartmentalization works targeting finer and finer granularities, such as libraries [67], [60], [19], [42], [53], [35], [5], [51], [29], [2], modules [22], [2], [52], files [2], and even functions/blocks of code [16], [64], [57], [1]. In that context, major attention was dedicated to compartmentalizing existing code, since rewriting software from scratch to work in a compartmentalized manner is costly and complex [16]. With recent developments on compiler-based compartmentalization, frameworks offer to apply isolation at arbitrary interfaces for a low to non-existent porting cost [67], [5], [35], [1].

Unfortunately, breaking down applications into compartments means that control and data dependencies through shared interfaces create new classes of vulnerabilities [61]: in order to provide safe compartmentalization, it is not only necessary to ensure spatial memory isolation but also to design interfaces with distrust in mind. For example, objects passed through APIs can be corrupted to launch confused deputy attacks [39], [21], data structures can be manipulated to control execution or leak data through Iago attacks [8], [11], called components can modify return values or indirectly access shared data structures to launch new forms of exploit, etc.

Even though interface-related vulnerabilities (denoted *Compartment-Interface Vulnerabilities* / CIVs in this paper) were previously identified to various extents in the literature [39], [8], [21], [61], almost all modern compartmentalization frameworks [67], [60], [19], [53], [35], [25], [45], [5], [51], [57], [30], [29], [1] neglect the problem of securing interfaces, and rather focus on transparent and lightweight spatial separation. Since CIVs are already problematic for interfaces hardened from the ground up (e.g., the system call API [28], [8]) with well-defined trust-models (kernel/user), their impact on safety is likely to be even greater when considering arbitrary interfaces and trust models that materialize when compartmentalizing existing software that was not designed with the assumption of hostile internal threats. Worse still, the complexity of safeguarding interfaces increases as more fine-grain components are targeted.

Beyond this lack of consideration, CIVs remain misunderstood; we ask the following research questions: *how widespread are CIVs when compartmentalizing unmodified applications? What are the API design patterns leading to them? What is the concrete impact of CIVs on the safety guarantees brought by compartmentalization, and what is the complexity of addressing them?* In order to achieve CIV mitigations that are generic and principled, we stress the need to formalize and quantify the problem.

This paper provides an in-depth study of CIVs. We taxonomize CIVs into a coherent framework, and systematize existing efforts to address them, highlighting categories that need attention in future research. In order to study existing CIVs in real-world scenarios, we propose ConfFuzz, an in-memory fuzzer specialized to detect CIVs at possible compartment boundaries. ConfFuzz automatically explores the complexity of compartment interfaces by exposing data dependencies leading to vulnerabilities. Contrary to existing fuzzers, that inject malformed data in a single direction (e.g., a library),

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24117
www.ndss-symposium.org

ConfFuzz can show the degree to which data flowing through an interface can be manipulated to harm either direction of a cross-compartment call. We apply ConfFuzz to a corpus of 39 compartmentalization scenarios, many of which previously proposed as use-cases of 12 existing research and industry frameworks. We uncover a wide data-set of 629 potential vulnerabilities[1]. We systematically study these issues, extracting numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them.

At the highest level, our results confirm how important the problem of CIVs should be to compartmentalization research: in many cases, CIVs seriously reduce or even fully negate all benefits of compartmentalization, and that even when the interface is extremely simple: we demonstrate an attack to extract isolated keys in OpenSSL, a common application of compartmentalization, and a decade-old vulnerability in sudo's authentication API. Beyond this, we note the following high-level insights: 1) CIVs are present in almost all existing interfaces, but at significantly varying degrees: for instance module APIs are much more vulnerable than library interfaces, and some interfaces are entirely CIV-free; 2) the complexity of objects crossing the interfaces imports more than the size of the API itself, and most of an API's CIVs can often be tracked down to a handful of objects; 3) fixing CIVs goes further than writing a few checks, and often requires reworking interfaces and partially redesigning existing software. We conclude with an appeal for more research towards systematic CIV detection and mitigation, hoping that this study can encourage future works to consider the issue of interfaces.

To sum up, this paper makes the following contributions:

- A systematization and taxonomy of CIVs and existing efforts to address them (§III).
- ConfFuzz, an in-memory fuzzer that automatically detects CIVs in existing software at arbitrary interfaces (§IV).
- A systematic study of the CIVs found by ConfFuzz applied to 39 real-world application compartmentalization scenarios, backing insights with concrete data (§V).
- A series of interface design guidelines intended to ease the development/adaptation of new/existing interfaces with compartmentalization in mind (§VI).

II. Motivation

The problem of secure interface design is not new [18]. The Linux system call interface, for example, is the result of years of organic evolution towards a strong boundary that preserves the integrity of the kernel in the presence of untrusted applications. Alas, designing strong interfaces in an adversarial context is notably hard: interface-related vulnerabilities are still regularly reported against the system call interface [28], [26], [23], even after decades of hardening. The task is even harder when assuming mutual distrust; the system call API, to take the same example, is notably weak at protecting the application from the kernel [8], [11], and requires extensive shim interfaces [43], [7], [46], [14], [13] to sanitize untrusted inputs and outputs.

Modern compartmentalization frameworks enable users to easily enforce spatial and temporal separation between components of existing software. Typically, code is either *sandboxed*, where a software component prone to subversion is restricted from accessing the rest of the system (e.g., image processing libraries), *safeboxed*, where sensitive data is only accessible to a component while maintaining high privilege (e.g., libssl), or *separated* into mutually distrusting subsystems [27]. Depending on domain crossing frequencies, compartmentalization promises good vulnerability containment at a modest cost [65].

[1]We open-sourced code and data-set: https://conffuzz.github.io

```
// ImageMagick callback exposed to libghostscript
static int MagickDLLCall GhostscriptDelegateMessage(
  void *handle, const char *message, int length) {

  /* CIV: unchecked dereference/usage of
   * sandbox-provided pointer/bounds information */
  (void) memcpy(*handle, message, (size_t) length);
  (*handle)[length] ='\0';
} /* ... abbreviated / simplified ... */
```

Listing 1: ImageMagick callback lets libghostscript perform arbitrary writes outside the sandbox.

Unfortunately, as shown by previous works [21], [61], [42] and highlighted in this paper, simply isolating software components is not enough: if cross-compartment interfaces have not been designed as trust boundaries (e.g., when compartmentalizing existing software), a wide range of *Compartment-Interface Vulnerabilities* (CIVs) arise. Reasoning about the safety of an interface is complex due to data dependencies exposed through the use of that interface by actors that previously belonged to a single trust domain, but under compartmentalization distrust each other. That complexity increases with that of interface-crossing data flows. CIVs arise when developers would like to avoid trust in a component, and encompass traditional confused deputies [18], Iago vulnerabilities [8], or Dereferences Under Influence (DUIs) [21]. We define CIVs more formally in §III.

Take the example of library sandboxing in ImageMagick as done by the Compiler-Assisted Library Isolation (Cali) [5] framework. Here, libghostscript is sandboxed because it is notoriously prone to high-impact vulnerabilities. Cali automatically sandboxes the library by applying compiler-based techniques to detect data shared between the application and the library, and place them in a shared memory region, before running application and library in separate processes. When the application needs to execute a function of the library, Cali performs the function call in the library compartment. Whenever the library needs to execute an application callback, Cali executes the callback in the application compartment.

This approach might seem sufficient: it makes it harder for attackers to escape the sandboxing of libghostscript. In practice, however, as shown in Listing 1, ImageMagick exposes a callback to libghostscript that allows the untrusted library to perform arbitrary writes in the application's compartment as often as it wants and at any time, negating spatial isolation entirely. This vulnerability, identified by our tool ConfFuzz, is caused by ImageMagick (victim compartment) dereferencing sandbox-provided pointers (`handle` and `message`) and bounds information (`length`) without sufficient checking.

Even though CIVs particularly affect new fine-grain compartmentalization frameworks such as Cali, they are not a specificity of these frameworks; as we show in §V, CIVs also affect long-standing, production-grade sandboxing approaches such as the worker/master separation in Nginx.

Clearly, while a strong compartmentalization framework capable of reliably enforcing spatial and temporal isolation is necessary, it is insufficient to offer tangible security benefits: software must also be adapted to fit distrust scenarios by vetting information that crosses compartment interfaces.

In the remainder of this paper, we propose the first systematization and taxonomy of CIVs and existing defenses, introduce ConfFuzz, a tool to automatically detect CIVs at potential compartmentalization boundaries, and use it to provide an in-depth study of real-world CIVs found with ConfFuzz.

III. Compartment-Interface Vulnerabilities

In this section, we provide the first definition and taxonomy of CIVs, along with a systematic review of existing defenses and their shortcomings. We define three main classes of CIVs, subdivided in a total of 8 subclasses. We relate each subclass with existing mitigations and discuss their limits, summarized in Table I. Here we use the term *corrupted* to refer to data voluntarily malformed by a malicious compartment.

A *Compartment-Interface Vulnerability* (CIV) is an instance of the general confused deputy [18] problem, where a compartment is its own deputy, and fails to adequately vet the use of the interface it exposes to other compartments, as well as its usage of other compartments' interfaces.

Malicious compartments can leverage a CIV to mount data and control-based attacks, confusing a victim compartment into leaking and altering its private data and addresses, executing code, etc. Many CIVs arise due to incorrect or missing checks and sanitization of data flowing through the interface; our taxonomy provides a comprehensive list of causes. Type confusion [17], DUIs [21], and Iago [8] are all part of the CIV spectrum. For instance, Iago vulnerabilities are CIVs at the system-call boundary, and DUIs match DC1 and DC2 (§III-B).

A. Cross-Compartment Data Leakage (DL)

a) DL1: Exposure of Addresses: A victim compartment may leak compartment-internal memory addresses, allowing an attacker, among others, to break ASLR in the victim and locate critical objects. In Listing 1, address leaks may help libghostscript to know where to point `handle` to. DL1 can stem from interface-crossing uninitialized data structures/fields and compiler-added padding [44], as well as data over-sharing between compartments. RLBox [42] proposes to address DL1 with pointer swizzling [66], ensuring that interface-crossing pointers can only address the sandbox. Generalized to arbitrary compartmentalization scenarios (e.g., by forcing interface-crossing pointers to address shared regions), this solves leaks due to oversharing, but does not address leaks due to uninitialized memory or padding. A near-complete protection can be achieved by combining RLBox with uninitialized memory use detectors such as MSan [56]. RLBox is not generic, as it requires strong types which are not available in all languages (e.g., C), and non-trivial manual refactoring that forbids certain C/C++ idioms. More generic but weaker protection can be achieved with ASLR hardening techniques [36], [24].

b) DL2: Exposure of Compartment-Confidential Data: A victim compartment may leak compartment-confidential data to a malicious compartment. The impact depends on the nature of the leakage; typical targets include cryptographic secrets, or user data. Leaks stem from over-sharing, as well as uninitialized shared objects containing data from previous allocations. SOAAP [16] proposes to address DL2 with manually annotated sensitive data, leveraging data-flow analysis to guarantee that annotated objects never cross trust boundaries. This approach does not prevent leakages due to uninitialized memory/padding, and is notoriously prone to human error: past attempts at compartmentalizing OpenSSL failed because of misidentification of private data [49]. Similarly to DL1, more complete protection can be achieved by combining SOAAP with uninitialized memory use detectors.

B. Cross-Compartment Data Corruption (DC)

a) DC1: Dereference of Corrupted Pointer: A victim compartment may dereference a pointer corrupted by a malicious compartment. The impact spans that of all classical spatial vulnerabilities: malicious actors may gain read, write, or execute capabilities in the victim's context, or cause Denial of Service (DoS). In Listing 1, corrupted pointers `handle` and `message` grant the untrusted compartment full write permissions. DC1 vectors include shared objects, cross-compartment function call arguments and return values. RLBox [42] proposes to address DC1 with pointer swizzling [66], with the same limitations as in DL1. Hardware memory capabilities such as CHERI [65] address DC1 by making it impossible to forge pointers. Although promising, this technology is still at a research prototype stage [3]. Its protection is not generic, requiring porting/annotations to fully address DC1, leaving certain C/C++ idioms unsupported. Generally, pointer authentication techniques like ARM PA [33] can also address DC1, but may require porting in a compartmentalized context.

b) DC2: Usage of Corrupted Indexing Information: A victim compartment may use indexing information (size, offset, index) corrupted by a malicious compartment. The impact includes DoS, and that of buffer overflows, and underflows, depending on the context. Typical vectors are, similarly to DC1, shared data, cross-compartment function call arguments, and return values. RLBox [42] proposes to employ compiler-based techniques to force experts to write checks prior accessing interface-crossing data (and in particular indexing information). This ensures that humans will sanitize the API, but does not offer correctness guarantees. Hardware memory capabilities [65] address DC2 by offering bounds safety for C/C++, with the limitations mentioned in DC1. Generally, bounds-checking techniques [59] can address DC2.

c) DC3: Usage of Corrupted Object: A victim compartment may use an object corrupted by a malicious compartment. Examples include corrupted strings lacking `NULL` termination or including arbitrary format string parameters, corrupted OS/libc constructs such as `FILE*`, corrupted integers causing numeric errors [41], etc. Corrupted objects may be control or non-control data [9]. DC3 impact includes, in addition to DoS, information leaks, or exposing read, write, or execute primitives. Vectors are the same as DC1 and DC2. The fundamental difficulty to address DC3 is that the validity of an interface-crossing object is entirely dictated by the

TABLE I: Compartmentalization mechanisms and frameworks that consider certain CIV classes. A ● indicates that a CIV class is fully addressed; a ◐ indicates a *partially* addressed CIV class; a ○ means fully vulnerable. An asterisk * indicates that the fix is not generic: the method makes assumptions about the source code being compartmentalized, or the use-case.

Mitigation Approach \ CIV Class	*DL1/Exp.Addr*	*DL2/Exp.Dat*	*DC1/Corr.Pt*	*DC2/Corr.Ind*	*DC3/Corr.Obj*	*TV1/API.Ord*	*TV2/Corr.Sync*	*TV3/Race*
TYPE-BASED CHECKS: RLBOX [42]	◐*	○	●*	◐	◐*	◐*	○	○
HARDWARE CAPABILITIES: CHERI [65]	○	○	●*	●*	◐*	○	○	○
UNDEFINED MEM. SANITIZERS [56]	◐	◐	○	○	○	○	○	○
BOUNDS-CHECKING TECHNIQUES [59]	○	○	○	●*	◐*	○	○	○
ASLR-GUARD [36], FG-ASLR [24]	◐	○	○	○	○	○	○	○
ANNOT. + DF-ANALYSIS: SOAAP [16]	○	◐	○	○	○	○	○	○
POINTER AUTHENTICATION [40], [33]	○	○	●*	○	○	○	○	○
API SEM. SANITIZATION: APISAN [68]	○	○	○	○	○	◐	○	○
TOCTTOU PROTECTION: MIDAS [6]	○	○	○	○	○	○	○	●*

semantics of the API and of its users. The difficulty to extract this information systematically is a well-known problem [68]. RLBox [42] partially addresses DC3 with automatic validity checking and copy, for the types that allow it (e.g., strings), and forces manual sanitization for others, with the drawbacks mentioned for DC1. Even for types that allow automatic sanitization such as C-style strings, checks remains partial (checking `NULL`-termination, but not format string parameters). Hardware memory capabilities [65] address part of the *symptoms* of DC3 with full spatial memory safety, but cannot offer complete protection: not all control and data attacks that could be mounted on DC3 rely on spatial memory safety vulnerabilities.

C. Cross-Compartment Temporal Violations (TV)

a) TV1: Expectation of API Usage Ordering: A victim compartment may expose functions (or callbacks) to other compartments and assume call ordering, without enforcing it. For example, a compartment may expose two functions `init()` and `work()`, expecting `init()` → `work()`. Malicious compartments may call `work()` first. The immediate impact can be any form of undefined behavior in the victim, spatial or temporal depending on the context, such as DoS, uninitialized pointer dereferences, use-after-frees, synchronization bugs, etc. RLBox [42] proposes tooling to enforce callback ordering, but still requires manual detection and patching of TV1. This poses further difficulties when a compartment can be concurrently queried. This could be coupled with API semantics inference techniques such as APISan [68] to lighten the manual effort, but these techniques remain incomplete. Generally, there is a need for more research in identifying and enforcing compartment API usage ordering.

b) TV2: Usage of Corrupted Synchronization Primitive: A victim compartment may use corrupted synchronization primitives (e.g., mutexes, locks). The impact includes, beyond DoS (deadlock), that of any race-condition which could be leveraged to mount control-based attacks. TV2 vulnerabilities are a special case of DC3 where the corrupted object is a synchronization primitive. Existing frameworks do not offer solutions to this class of vulnerabilities: addressing TV2 is particularly challenging, as it requires redesigning the way distrusting compartments cooperate in multithreaded contexts.

c) TV3: Shared-Memory Time-of-Check-to-Time-of-Use: A victim compartment may check corrupted data in the shared memory. This may allow a malicious compartment to corrupt the value after the check and before the use, making the check useless. TV3 may lead to any previously mentioned CIV impact. TV3 vectors are shared objects with double fetches. Existing mitigations include forcing the copy of objects to a private region before checking, as done by RLBox [42] (with the genericity limitations mentioned in DL1), or forbidding concurrent modification altogether as done by Midas [6] (that targets only kernel-space).

D. Summary: CIV Protections are in their Infancy

No existing compartmentalization framework tackles all CIV classes. Techniques that can address a subset of CIVs only offer partial and/or non-generic solutions. Even the most comprehensive system, RLBox [42], relies extensively on manual checking with no guarantees of check correctness. It is likely that combining all the techniques required to achieve state-of-the-art protection would result in an impractical performance overhead that contradicts the initial motivation of using lightweight isolation technologies for fine-grain compartmentalization. This observation motivates our study assessing the safety and complexity impact of CIVs.

IV. CONFFUZZ: EXPLORING CIVS WITH FUZZING

A. Assumptions and Threat Model

We assume an application decomposed into compartments that are mutually distrusting. Compartments are defined as protected subsystems (as proposed by Lampson [27]), with private code, heap, and stack. Compartments communicate through interfaces. If a pointer is passed through an interface, the object it references is shared between the two compartments. A protection mechanism enforces spatial isolation: code cannot access private data or code from other compartments. The compartmentalization framework enforces cross-compartment control-flow integrity: one compartment can only call explicit entry points exposed by other compartments. These assumptions fit the vast majority of modern frameworks [67], [60], [19], [53], [35], [25], [45], [5], [51], [30], [29], [1].

We assume a completely compromised compartment which we refer to as the *malicious* compartment: an attacker can execute arbitrary code in its context. Compartments communicate through interfaces that respect the semantics of function calls, with variable degrees of sanitization. The malicious compartment attempts to misuse these interfaces to attack another compartment called the *victim*. The interface can be abused in either direction, according to the caller/callee role of the malicious/victim compartments:

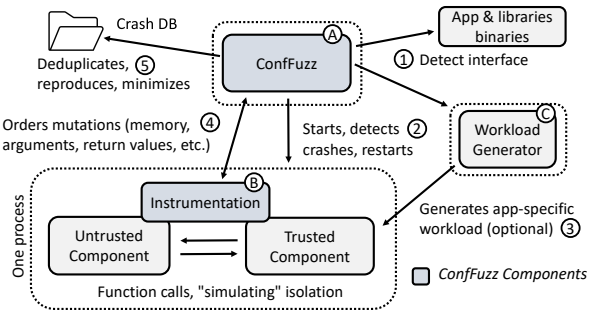

Fig. 1: ConfFuzz architecture diagram.

a) Safebox: As the caller, the malicious compartment can abuse an interface *exposed by the victim* (callee). Vectors of corruption are function call arguments, data in shared memory, and return values of callbacks invoked by the victim to be executed in the context of the malicious compartment. This corresponds to a *safebox* scenario, in which a trusted subsystem (e.g., libssl) is protected from the rest of the system.

b) Sandbox: As the callee, the malicious compartment can abuse an interface *invoked by the victim* (caller). Here, corruption vectors are return values, data in shared memory, and parameters of callbacks invoked by the malicious compartment to be executed in the victim's context. This is a *sandbox* scenario, in which an untrusted component (e.g., a 3rd-party library) is prevented from accessing the rest of the system.

B. Overview

To assess the impact of neglecting interface safety, we propose to fuzz monolithic (non-compartmentalized) software at possible arbitrary compartment boundaries, and analyze the set of CIVs we uncover. ConfFuzz is an *in-memory* fuzzer [58]: it instruments software targets to hook into arbitrary interfaces, such as libraries, modules, functions, etc.

Because of their systematic nature, approaches based on static analysis may seem enticing to explore interface-related issues. However, related works leveraging such techniques fail to scale to more than simple programs [21]. Hence, we take a pragmatic approach and rely on fuzzing. Our goal is further different from existing in-memory/in-process/library/API fuzzers [55], [58] because our tool needs to fuzz both ways (safebox/sandbox). Hence, we develop ConfFuzz from scratch.

a) Two-Components Approach: Each run of ConfFuzz considers two communicating software components: a malicious, and a victim one. ConfFuzz simulates attacks towards the victim by automatically altering data crossing the interface between them; we call this *interface data altering*. For that, ConfFuzz hooks into the interface and fuzzes in both directions (sandbox/safebox), altering function call arguments, shared data, and return values for direct interface calls and callbacks.

b) Architecture Overview: As shown in Figure 1, ConfFuzz is composed of a self-contained fuzzing monitor (Ⓐ), and dynamic binary instrumentation (DBI, Ⓑ). This instrumentation, which sits between the malicious component and the victim in the application's process, leverages the Intel Pin [37] DBI framework. Using Pin lets us apply ConfFuzz to software with a low engineering cost, and hook at arbitrary interfaces unlike other approaches e.g., `LD_PRELOAD`.

First, ConfFuzz automatically identifies the interface between application and compartment components by analyzing debug information in the corresponding binaries (①). When starting the application (②), the fuzzing monitor dynamically injects instrumentation wrappers at the detected compartment interface. Following this, it optionally starts a workload generator (Ⓒ) to stimulate the application (③). At runtime, at each API call, the fuzzing logic in the monitor determines a set of alterations to perform, possibly through mutation of an existing set, and performs the alterations via the instrumentation (④).

The application runs with Address Sanitizer [54] (ASan) as bug detector. When ASan reports a crash in the victim, ConfFuzz deduplicates it based on the stack trace. If the bug is not known, ConfFuzz reproduces it, before minimizing the set of alterations performed to obtain the nucleus of alterations that trigger the bug (⑤). ConfFuzz is implemented in 4.5K LoC of C++, Bash and Python. The following subsections present ConfFuzz's fuzzing process steps in greater details.

C. Interface Detection and Instrumentation

ConfFuzz automatically handles the detection of the signature of a given target API. The tool gathers a list of API symbols: functions and callbacks used by the victim and malicious components to communicate with each other, along with, for each of these, the number of arguments, the type and size of each argument, and the type and size of the return values if appropriate. We compile the target application with debug symbols, allowing ConfFuzz to use DWARF metadata to retrieve interface and type information. The tool can automatically infer the list of functions composing the interface exposed by shared libraries, and the user can provide that list manually when targeting arbitrary interfaces. Most functions are instrumented when the application starts. Concerning callbacks, ConfFuzz automatically detects them at runtime by scanning API call parameters for function pointers, and instrumenting the identified functions on the fly. ConfFuzz automatically infers what data is shared between the malicious and victim components, considering that all buffers referenced by pointers crossing the API are shared data.

Each symbol, including API elements and callbacks, is instrumented at entry and exit. At each of these events, the instrumentation checks for reentrance to protect against API calls performed by the compartmentalized component itself, notifies the fuzzing monitor with information about the function being executed and arguments/return value information, and allows the monitor to perform alterations: depending on the type of symbol and the fuzzing direction (safe/sandbox), altering argument values, return value, altering shared memory, etc. The instrumentation is kept as simple as possible and all the fuzzing logic runs in the monitor. Instrumentation and monitor communicate via a well-defined protocol using pipes.

D. Workload Generation and Coverage

ConfFuzz passively sits at internal API boundaries in an application, and users must determine an application-specific configuration and input workload that exercise the API. Finding a good set of inputs with high coverage is a problem shared across most fuzzers [48], [69]. In the data set considered in this paper, the time to understand the configuration system of

TABLE II: *ConfFuzz data altering strategies for each CIV class*. Each class of data alterations done by ConfFuzz is targeted at a particular type of CIV, as shown in this table.

CIV Class	Corresponding Data Alteration Strategy
DC1	*Alteration of pointer types* to invalid values (zero page, arbitrary unmapped areas).
DC2	*Alteration of integer types*: increments/decrements to trigger over/underflows, replacement to known limits such as `INT_MAX` (possibly at offsets) to trigger numeric errors, replacement with random values.
DC3	*Alteration of non-pointer types & of pointer targets*: decrements/increments at varying offsets in the object, replacement of bytes at various offsets in the object. *Replacement of pointers to the same type* (replay), and *Replacement of pointers to different types* (type confusion) to trigger more complex DC3 flaws.
TV1	*Non execution of API functions* for partial TV1 detection.
TV2	*Alteration of mutex/lock types* in shared memory (allows partial detection of TV2).
TV3, DL1-2	Not targeted.

an application and find an appropriate workload went from a few minutes to a few hours for a graduate student. We also explored the use of OSS-Fuzz [47] to generate workloads for the application, but found that hand-tuned workloads are generally better at precisely targeting the internal APIs that we intend to fuzz, which is critical for this study. Nevertheless, OSS-Fuzz should be considered in cases where the manual effort to create workloads should be minimized.

ConfFuzz is not coverage-guided. However, to provide an indication of how comprehensive the fuzzing of an API is with a given configuration and workload, ConfFuzz measures *API coverage*: the number of target API functions reached. This metric can be compared to the target API's size to understand the coverage of a given configuration and workload.

E. Interface Data Altering and Fuzzing Strategy

At each API crossing, the instrumentation notifies the monitor, which may proceed to alter interface data over the entire attack surface exposed to the malicious component. When fuzzing in sandbox mode, the malicious component may alter return values and callback arguments. In safebox mode, function call arguments and callback return values may be altered. In both cases, the malicious component may also alter data shared between the malicious and victim components.

ConfFuzz probabilistically decides whether or not to alter data at an API crossing. In order to avoid revealing only shallow bugs that systematically crash in early fuzzing stages, we use a dynamic probability adaptation threshold: at first, the threshold is at 0, i.e., ConfFuzz alters data aggressively at all crossings. When the number of new crashes becomes scarce, ConfFuzz increments the threshold to find crashes further in the API usage. Concretely, based on a counter incremented at each API crossing, crossings that come before the threshold is reached see their data altered with a lower probability. This allows ConfFuzz to find stateful crashes as well.

ConfFuzz alters values in two ways: applying increments/decrements, and replacing the value altogether. ConfFuzz uses type information to drive alterations. For pointer values, ConfFuzz may perform replacements with other pointers of the same type, of different types, at varying offsets, at the zero page, on the heap, stack, data, text sections, etc. For integer values, ConfFuzz may perform replacements with known limits (e.g., `INT_MAX`). A detailed description of data alterations performed by ConfFuzz is provided in Table II. While fuzzing, the fuzzer enriches an alteration corpus with values gathered during previous alterations. Values from the corpus are reused, possibly mutated, with a given probability.

F. Crash Processing and Bug Analysis

a) Crash Sanitization: Upon a crash, ConfFuzz compares the ASan stack trace with its database of known crashes. If the crash is a duplicate, no further analysis is performed, but information about the new occurrence is logged. ConfFuzz then checks whether the new crash is a false positive. False positives arise when altered objects are sent back to the malicious component by the victim. In such cases the malicious component corrupts itself, yielding an invalid bug.

In order to detect false positives, ConfFuzz walks down the stack trace until it finds an entry referencing code belonging to a component. If that component is the one considered as malicious for this fuzzing run, the crash is considered a false positive. Even though such false positives are not valid crashes, ConfFuzz still attempts to minimize them, as this allows to detect non-viable data alterations that can be avoided later in order to minimize time wasted on false positives.

b) Reproduction and Minimization: After sanitization, ConfFuzz systematically attempts to reproduce crashes. Unfortunately, not all crashes are reproducible, as some might be due to particular non-deterministic factors such as scheduling effects, reliance by the application on random values/changing external inputs, etc. A non-reproducible crash cannot be further processed automatically by ConfFuzz. Still, information regarding such crashes are valuable for the analysis and are logged for manual inspection. On the other hand, if the crash can be reproduced, the monitor gradually minimizes it.

As part of the minimization step, ConfFuzz tries to understand the minimum set of alteration steps required to trigger a given bug. ConfFuzz gradually goes through each alteration performed in reverse order (since the last alterations performed tend to be the most likely to trigger the crash), and determines whether the alteration is sufficient to trigger the crash, necessary to trigger the crash (without it the crash cannot be reproduced, but it is not sufficient by itself), or superfluous. This results in a minimal set of attack primitives that the malicious component can perform to trigger the bug.

c) Impact Analysis of Crashes: After processing crashes, ConfFuzz performs initial triage. It harvests ASan-provided information: whether the crash is due to an illegal read, write or execution, allocator corruption, or to a `NULL` dereference, along with faulty addresses. Then, for R/W/X crashes, ConfFuzz tries to determine whether the vulnerability is arbitrary: for each alteration in the minimized steps, ConfFuzz mutates the altered value with increments/decrements, trying to reproduce the crash. If the crash is reproducible and the faulty address varies accordingly to the increment/decrement, the vulnerability is considered arbitrary.

TABLE III: Bugs found for sandbox and safebox Trust Models (TM, fuzzing directions). *Refs.* links to studies that implemented an equivalent scenario. *Victims* gives the number of individual software components (application code, libraries, modules) that the malicious component managed to crash. *API Coverage* represents workload coverage: *callers* denotes the number of components calling the API, and *coverage* denotes how many functions of the fuzzed API are hit at runtime. *Impact* describes the type of bug: R/W/X fault, `NULL` dereference, or improper calls to the allocator (e.g. calling `malloc` with a negative value).

TM	Application	Compartment API	References	Crashes		Victims	API Coverage		Impact (of which arbitrary)				
				Raw	*Dedup.*		*Callers*	*Coverage*	*Read*	*Write*	*Exec*	*Alloc*	*Null*
Sandbox	HTTPd	libmarkdown	[42]	192	13	3	1	100% (4/4)	10 (8)	7 (7)	0 (0)	1	4
		mod_markdown		381	71	5	1	100% (1/1)	62 (52)	17 (14)	2 (1)	0	30
	aspell	libaspell		278	8	1	1	34% (48/141)	7 (7)	7 (7)	2 (1)	0	3
	bind9	libxml2 (write API)		0	0	0	1	86% (13/15)	0 (0)	0 (0)	0 (0)	0	0
	bzip2	libbz2	[67], [5]	16	5	1	1	62% (5/8)	5 (2)	1 (0)	0 (0)	0	0
	cURL	libnghttp2		61	7	2	1	50% (18/36)	3 (3)	5 (5)	0 (0)	1	3
	exif	libexif		400	7	1	1	10% (13/129)	3 (3)	0 (0)	0 (0)	0	5
	FFmpeg	libavcodec		316	20	3	4	31% (19/60)	13 (12)	12 (12)	0 (0)	3	7
		libavfilter		51	1	1	2	12% (2/16)	1 (1)	0 (0)	0 (0)	0	1
		libavformat		217	9	2	3	52% (10/19)	8 (7)	1 (1)	0 (0)	0	7
	file	libmagic		150	5	1	1	63% (7/11)	5 (2)	1 (1)	0 (0)	0	4
	git	libcurl	[22]	13	4	2	1	90% (18/20)	2 (2)	2 (2)	0 (0)	1	1
		libpcre		81	2	1	1	44% (8/18)	2 (2)	0 (0)	0 (0)	2	0
	Inkscape	libpng	[67]	66	3	1	1	46% (14/30)	2 (1)	2 (2)	0 (0)	0	1
		libpoppler	[16]	81	4	2	1	100% (9/9)	4 (3)	4 (4)	0 (0)	0	2
	libxml2-tests	libxml2 (write API)		0	0	0	1	100% (47/47)	0 (0)	0 (0)	0 (0)	0	0
	lighttpd	mod_deflate		117	26	2	1	100% (6/6)	16 (11)	5 (0)	1 (1)	2	9
	Image Magick	libghostscript	[5]	67	14	2	1	100% (11/11)	4 (2)	1 (1)	0 (0)	3	9
		libpng	[67]	778	44	1	2	22% (17/77)	2 (2)	9 (9)	2 (0)	2	39
		libtiff	[67]	197	14	2	1	30% (13/43)	3 (3)	6 (6)	0 (0)	0	13
	Nginx	libpcre		144	10	1	1	93% (14/15)	8 (7)	3 (3)	0 (0)	6	2
		mod_geoip	[52]	276	25	2	1	35% (5/14)	21 (17)	4 (1)	1 (1)	1	10
	Okular	libmarkdown	[42]	64	5	3	1	100% (4/4)	3 (1)	0 (0)	0 (0)	1	2
		libpoppler	[16]	195	9	1	1	6% (24/379)	8 (6)	7 (7)	0 (0)	1	4
	Redis	mod_redisbloom		389	23	1	1	42% (8/19)	18 (13)	6 (4)	0 (0)	0	13
		mod_redisearch		381	21	1	1	54% (18/33)	15 (14)	14 (11)	0 (0)	0	12
	rsync	libpopt		167	8	1	1	90% (9/10)	4 (3)	2 (0)	0 (0)	0	5
	squid	libxml2		226	12	1	1	70% (7/10)	9 (5)	3 (3)	4 (1)	0	4
	su	libaudit		0	0	0	1	66% (2/3)	0 (0)	0 (0)	0 (0)	0	0
	Wireshark	libpcap		162	8	2	1	50% (20/40)	8 (3)	5 (5)	0 (0)	0	4
		libzlib		42	1	1	1	85% (6/7)	0 (0)	0 (0)	0 (0)	0	1
	Total:			5508	379	47	38	N/A	246 (192)	124 (105)	12 (5)	24	195
Safebox	cURL	libssl	[5]	198	27	1	1	25% (14/56)	18 (10)	5 (4)	1 (1)	0	17
	GPA	libgpgme		174	9	1	1	4% (3/72)	7 (2)	0 (0)	0 (0)	0	6
	GPG	libgcrypt	[5]	4221	105	1	1	15% (15/95)	64 (60)	4 (0)	0 (0)	77	20
	Memcached	internal_hashtable	[45]	4037	16	1	1	50% (6/12)	10 (5)	2 (0)	0 (0)	1	6
	Nginx	internal_libssl-keys	[45], [60], [15], [34]	599	46	1	1	50% (2/4)	32 (1)	28 (0)	0 (0)	0	22
		libssl	[5], [1], [22], [51]	346	39	2	1	11% (11/96)	16 (13)	19 (13)	2 (1)	0	26
	sudo	internal_auth-api		191	5	1	1	100% (5/5)	5 (4)	0 (0)	0 (0)	0	4
		libapparmor		97	3	1	1	100% (2/2)	2 (2)	2 (0)	0 (0)	0	2
	Total:			9863	250	9	8	N/A	154 (97)	60 (17)	3 (2)	78	103

V. A Large-Scale Study of Real-World CIVs

In this section, we use ConfFuzz to gather a large dataset of real-world CIVs, from which we extract insights on a set of research questions: (Q1) what is the number of CIVs at legacy, unported APIs?, (Q2) what patterns lead to CIVs and are all APIs similarly affected by CIVs?, (Q3) what is the complexity to address these CIVs when compartmentalizing?, and (Q4) what is the range of severity of the CIVs we uncover, i.e. without a fix, what can attackers do? Table III shows our results. §V-A and §V-B give an overview of the methodology and results. The following sections provide in-depth analysis.

A. Methodology

a) Choice of Scenarios: Our corpus is composed of 25 applications and 36 library/module/function APIs, totaling 39 real-world sandbox and safebox scenarios. We choose scenarios that are meaningful security-wise, e.g., sandboxing image processing libraries because they are higher-risk, or safeboxing functions manipulating SSL keys because they are sensitive. Motivated by (Q2), we choose scenarios that encompass a diversity of API types (libraries, pluggable modules, internal APIs), and usages (image processing, text parsing, logging, key management, etc). We focus on highly popular applications. Several of these scenarios (16/39, see table) have been presented as use cases in 12 different compartmentalization frameworks, only 2 of which [42], [16] providing protection against certain CIV classes (discussed in §III). The smaller number of safebox scenarios reflects that, in general, libraries/modules tend to perform more untrusted operations that one might want to sandbox (e.g., request processing, complex data parsing) than trusted operations that one might want to safebox (e.g., cryptographic operations).

b) Comparison with Related Works: As discussed in §IV-B, existing fuzzers are unfit by design to the investigation of CIVs, so we present no baseline. A relevant work in the domain of static analysis is DUI Detector [21], which could be used to detect classes DC1 and DC2 presented in §III-B. Unfortunately, its authors were unable to provide us with its source code. We therefore provide a textual comparison in §VII. We evaluated the cost of DBI (Pin) in ConfFuzz on a representative set of applications (Nginx, Redis, file, xmltest),

and found it to be $\leq$65%, thanks to Pin's probe (JIT-less) instrumentation mode. These numbers match official data [12]. The overhead could be further reduced with compiler-inserted instrumentation, which we leave for future works.

B. Overview of the Data Set

Overall, ConfFuzz found 629 unique bugs deduplicated from 15,371 crashes. ConfFuzz uncovered bugs of 4 different classes as listed in §III: DC1-3, and TV2. In the *sandbox* mode, where ConfFuzz fuzzes from the (untrusted) component towards components calling it (e.g., application code, other libraries/modules), we found 379 CIVs. For 3 scenarios, ConfFuzz did not find any CIVs. We discuss these scenarios in §V-D. In the *safebox* mode, the fuzzer found 250 unique crashes. There are significant differences with the sandbox results impact distribution, which we analyze in §V-D. For both modes, the summed number of bugs of all impact types is larger than the total bug count (999 versus 629) because 224 bugs (1/3rd of the bugs) have more than one type of impact.

a) API End-Point Coverage: We observe unequal interface coverage across scenarios. Some scenarios such as HTTPd with libmarkdown show full API coverage, while others such as bzip2 with libbz2 feature poorer coverage. Generally, low coverage is due to ConfFuzz fuzzing a single software configuration or CLI option. In the case of bzip2 for example, ConfFuzz fuzzes decompression (`-d`), but not compression or archive testing (`-z`/`-t`). These entry points of libbz2 are left unexplored, and the user is notified. This is a common problem across fuzzers, addressable by varying configurations [48], [69], and fuzzing with more workloads, which we consider out of scope. Nevertheless, despite its simple exploration approach, ConfFuzz shows good ability to find relevant bugs. For example, in HTTPd with libmarkdown, a scenario from RLBox [42], ConfFuzz correctly discovered all CIVs addressed by RLBox, asserted the criticality of the bugs, and even one more due to library version differences.

C. Prevalence of CIVs

At the highest level, the results confirm our expectations: *CIVs are widespread among unmodified applications*. Indeed, all but 3 scenarios present CIVs. Looking into greater details, however, disparities appear: libraries present widely varying CIV numbers, ranging 0-105 for a single scenario. Disparities become even clearer when looking at the ratio of vulnerable over covered API elements, ranging 0%-100%, as shown in Figure 2. This observation is not a consequence of coverage disparities; we find that the number of functions covered and the number of CIVs found are uncorrelated ($|r| < 0.09$). Similarly, there is no correlation between the size of compartment APIs and the number of CIVs found ($|r| < 0.1$). This observation further materializes when considering APIs that do comparable tasks, e.g., libbz2 and libzlib (compression), or libpng and libtiff in ImageMagick (graphics). In both cases, CIV counts vary by 3-5x for nearly identical coverage. This is most startling in ImageMagick where the workload (image format conversion) is identical in both cases, but the number of CIVs jumps from 14 to 44. These observations hint that the vulnerability of interfaces to CIVs is a factor of *individual patterns and structural properties* rather than of size or functionalities. We study these patterns in the next section.

TABLE IV: Low-level CIV patterns from API-crossing types.

Type Class	Low-Level CIV Patterns Involving this Type Class
Integer Types	sizes and bounds (DC2), error codes and return values (DC3, Pattern 4), and generally control-flow manipulations (DC3)
Custom Structs	state data (DC3, Pattern 1), mutexes (TV2, Pattern 1), non-control data (DC3)
C-style Strings	version/error strings (Pattern 4), serialized data (all DC1&3)
void buffers*	exposed allocator arguments (DC1-3, Pattern 5), and generally opaque data structures (DC1&3, rare in this dataset)

Insight: smaller APIs do not imply less CIVs in unmodified applications: API size and number of CIVs are uncorrelated.

D. Patterns Leading to the Presence/Absence of CIVs

Zooming in, the types/data structures triggering crashes are integer types, custom classes and structures, C-style strings, and raw `void*` buffers flowing through compartment interfaces. As shown in Figure 3, integer types and custom objects are the main culprits, being involved in 85% of CIVs, while only 33% of CIVs are due to C-style strings and raw buffers. Table IV summarizes the low-level CIV patterns observed for each type. The following paragraphs give insights from a selection of these patterns and higher-level observations.

a) Pattern 1 – Modularity & Exposure of Internal State: Module APIs are some of the most vulnerable interfaces in the study; HTTPd, Nginx, Redis, and lighttpd modules rank 2nd-10th in CIV count, clearly above average. HTTPd, particularly, ranks 2nd with 71 unique CIVs, even though its module API has a single function and entry point. Impact-wise, modules represent more than half of read CIVs, and 1/3rd of write and execute CIVs, even though they only represent 5/39 of our scenarios. In short, the most modular interfaces of the dataset get *more bugs* and *worse bugs* on average.

We track down these observations to one common pattern across module APIs: *exposing the application's internals to maximize performance and flexibility*. Unlike library APIs, module APIs are designed to accommodate *generic needs with good performance*. These needs stand at odds with the requirements of compartmentalization: in order to achieve this, the application must expose its internals to the module, resulting in significantly less encapsulation than with traditional libraries. Take Apache modules as an example: HTTPd modules can register hooks into core operations of the server (e.g., configuration, name translation, request processing), and are systematically passed a request structure `request_rec`. This structure is highly complex, with over 75 fields, of which over 60% of pointers, many of them referencing other complex structures: memory pools, connection data, server data, and even synchronization structures (resulting in TV2 CIVs). These shared structures are not only very hard to sanitize, they lead to oversharing when compartmentalizing because most modules do not need access to all of them. These observations become even more clear comparing the results of the two Apache scenarios: mod_markdown and libmarkdown. Here, the HTTPd Markdown module uses libmarkdown under the hood, thus we can isolate at the module boundary, or at the library boundary to obtain comparable guarantees. As expected, isolating at the library boundary yields fewer crashes than isolating at the module boundary (5.5x less). Despite a larger number of

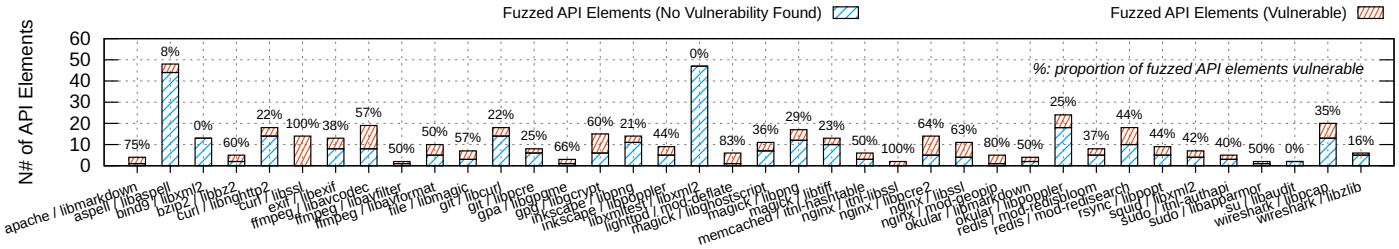

Fig. 2: Proportion of covered vulnerable endpoints versus covered endpoints for each scenario (see Table III for coverage).

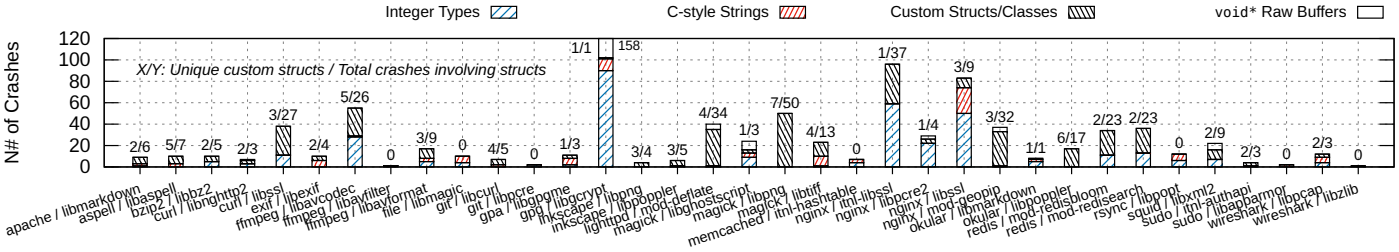

Fig. 3: High-level type classes involved in CIVs for each scenario.

entry points, the attack surface exposed to the library is far smaller than that of the module API due to encapsulation; the library does not have knowledge or access to Redis' internals. We make similar observations for Nginx, Redis, and lighttpd. These observations are not trivial: multiple studies propose privilege separation of modules [22], [52] without considering this problem, and other studies [31], [51], [29] build on the assumption of modularity being fundamentally good for compartmentalization. We stress that this is not always the case, and that more research is need to achieve fast, generic, and compartmentalizable modular APIs.

Insight: modularity $\neq$ low compartmentalization complexity: boundaries exposing internal state are hard to protect.

b) Pattern 2 – CIV-Resilient Input-Only APIs: Several scenarios are entirely CIV-free: libxml2 write API, and su with libaudit. In the case of libxml2, even API tests with full coverage of all 47 endpoints do not yield a single CIV. We manually confirmed that no CIVs are possible at these APIs, reducing these observations to one common pattern: *designing the API like a write/input-only endpoint*. For instance, Bind9 uses libxml2 to store statistics onto the filesystem in XML format; it strictly forwards data to libxml2, which formats and stores it. The situation is similar in su with libaudit, used as a logging facility. su passes logs to libaudit, which processes them. There is no feedback loop and no data flow from the library to the application, and thus no CIVs. Note that this does not apply to squid, which uses a different API of the libxml2 family. While this restrictive pattern cannot suit generic sandboxed API needs, it shows that there are naturally robust APIs for privilege separation, and remains applicable to several other scenarios such as (de-)compression or image processing. We expand more on CIV-resilient APIs in §VI.

Insight: "input-only" APIs are CIV-resilient by design, and can be found is several scenarios.

c) Pattern 3 – Corrupted Data Forwarding: In sandbox scenarios, the number of victim components (individual application/libraries/modules that the untrusted component managed to crash), as shown in Table III, is >1 for 1/3rd of the cases. This is surprising, since only 4/39 scenarios have more than one caller component — 3 of them being FFmpeg. We tracked down these observations to one common pattern, where a trusted component T_1 receives corrupted input from an untrusted component U_1, and forwards it to another trusted component T_2. Component T_2 thus receives corrupted data from *trusted* component T_1. Take the example of curl, which features this CIV pattern in Listing 2. Here, curl's callback `send_callback` (T_1) receives corrupted input `mem` and `length` from libnghttpd2 (U_1), and transmits it to libssl (T_2) via `send_underlying`, resulting in a libssl crash. The untrusted HTTP parsing library thus manages to attack libssl even though the two libraries do not directly communicate. Intuitively, this pattern motivates to check as early as possible to avoid unsanitized data spreading. Unfortunately it is not always possible to perform checks at the untrusted boundary: T_1 is the recipient of the data and may not have knowledge of the semantics of the corrupted object. Thus, checks must be thought with the full system in mind: *it is not because a library only interfaces with safe or trusted components that it won't receive untrusted inputs*. This poses the question of "when to check, and when not to", to ensure that no check is missed, while limiting the amount of unnecessary vetting.

```
ssize_t send_callback(nghttp2_session *h2,
    uint8_t *mem, size_t length, ...) {
  /* (...), send_underlying = libssl callback */
  written = ((Curl_send*)c->send_underlying)(data,
      FIRSTSOCKET, mem, length, &result);
} /* (...) simplified */
```

Listing 2: Fall-through CIV in libssl, with curl and libnghttpd2 isolated: curl transparently forwards corrupted data to libssl.

Insight: the attack surface of a component exceeds interactions with directly reachable untrusted components; targeted attacks to non-adjacent components are realistic and checks must be thought with the full system in mind.

```
static char * ngx_http_block(ngx_conf_t *cf, ...) {
  /* (...) simplified */
  if (module->postconfiguration(cf) != NGX_OK)
    /* cf used in caller after return: */
    return NGX_CONF_ERROR; ①

  if (ngx_http_variables_init_vars(cf) != NGX_OK) ②
    return NGX_CONF_ERROR;
} /* (...) many operations involving cf here */
```

Listing 3: Multi-alteration vulnerability with Nginx modules.

d) Pattern 4 – Error-Path CIVs: About 42% of the dataset's crashes require more than one alteration to manifest. In these cases, part of the alterations aim at diverting the control flow to reach a vulnerable location. We manually studied these multistep vulnerabilities and found that, in sandbox scenarios, many present a common pattern where the vulnerability is located in an error-handling path. In this case, ConfFuzz diverts the control flow to the error path (e.g., via a non-zero return value), so that the application reaches a location that makes use of another corrupted value. Take the example of Nginx with the GeoIP module isolated, as depicted in Listing 3. Nginx calls the module's post-configuration callback, passing it a pointer to its shared configuration object `cf`. Assume GeoIP corrupts `cf`. If the call succeeds (default behavior in GeoIP), Nginx proceeds with other operations making use of the configuration object, particularly at ②, triggering a crash. Unfortunately, crashes at this stage hide potential for corruption beyond ② or in the caller. Exploiting error-path CIVs, ConfFuzz found that by corrupting `cf` and returning an error value from `postconfiguration`, further crashes could be uncovered in `ngx_http_block`'s caller, after the return statement ①, avoiding ②. Beyond showing that ConfFuzz can easily uncover multistep crashes, this strengthens Pattern 3's observations: checks should be done as early as possible, ideally before any control flow operation. CIV checks should preempt other functional checks like error code handling to avoid attack surface multiplication.

e) Pattern 5 – Allocator Exposition: The dataset features 102 allocator corruption bugs affecting a total of 14 scenarios, with 77 bugs coming from GPG with libgcrypt. We investigated this CIV class and reduced it to two related patterns, where (1) corrupted data flows reach parameters of allocators calls, or (2) trusted components expose untrusted components with a direct window to their allocator. In the first case we observe numerous DC1-corrupted pointers reaching `free()`, DC2-corrupted integers reaching `malloc()`, as well as cases within the libc due to DC3-corrupted `FILE*` pointers. The second case appears in GPG with libgcrypt, and almost systematically in sandbox scenarios with module APIs (HTTPd, Nginx). Here, the application presents the sandboxed component with a malloc-like API that allows it to allocate objects with the application's memory allocator. This results in numerous DC1-2 CIVs and explains the peak in `void*` related bugs for GPG in Figure 3. Allocator exposition is achieved with the intention to achieve higher performance (custom memory allocator), for introspection reasons (statistics, or record log information), or for correctness reasons (allocation/freeing is spread over both components). Vulnerabilities arising from this pattern are high impact: they allow malicious compartments to trigger arbitrary use-after-frees, heap Feng Shui [63], and other allocator exploitation techniques. Since this pattern also defeats compartment heap separation, it is likely that DL2 CIVs will arise too. In all cases, allocator exposition is very difficult to get right in compartmentalized contexts, and should be avoided where possible. Removing allocator exposition can be straightforward if performed for performance reasons only, but may require significant redesign of the API if done for introspection or correctness reasons.

Insight: cross-compartment memory management is hard & leads to exploitable CIVs. Fixes may imply API redesign.

E. Security Impact of CIVs

At the highest level, our impact analysis confirms that CIVs are particularly critical from a security standpoint. We find that more than 75% of scenarios studied present at least one write vulnerability, and all safebox scenarios present at least one arbitrary read, which defeats attempts to protect secrets. We present an overview of impact results from Table III, before focusing in depth on a selection of bugs from the data set.

As visible in Table III, the distribution of impact is clearly in favor of read vulnerabilities, `NULL` dereferences, followed by write, allocator corruption, and code execution. This follows the distribution of typical patterns in applications. Most commonly, a victim component reads data referenced by a pointer provided by a malicious component (resulting in read vulnerabilities), or use a corrupted integer within a check (e.g. a success code) before accessing internal data (resulting in `NULL` dereferences if the application reads unallocated/uninitialized data with the call returning a success code). Less commonly, a victim write to a pointer provided by a malicious component (write impact). Memory allocator corruption bugs most commonly happen via pattern 5 of §V-D, or when size parameters flow from an interface to an allocation site. The least common impact is execute, typically resulting from the victim executing a callback passed by a malicious component.

We find that more than 70% of read and 66% of write vulnerabilities are arbitrary, as well as half of execute CIVs. Thus, in the absence of countermeasures, if a subverted component can perform illegal R/W/X operations outside its compartment through APIs, it is likely to be able to do so at any address. Further, even though the proportion of execute impact is low (8/39 scenarios), it is probable that attackers will be able to mount attacks with arbitrary R/W CIVs to reach code execution. Next, we illustrate these observations with an analysis of concrete bugs from the dataset.

1) Case Study: OpenSSL Key Extraction: OpenSSL is a popular compartmentalization target, being both high-risk (high-complexity, shipped in network-facing applications) and sensitive (holding secrets). We count at least 8 studies safeboxing it [15], [51], [16], [5], [1], [22], [34], [60] with attempts going beyond that of academic research [49]. Safeboxing approaches typically either (I1) compartmentalize OpenSSL in full and isolate at the libssl API, or (I2) compartmentalize at the libcrypto internal API of key-interacting primitives. I2 is viewed as more robust because of the reduced TCB and the ability to tackle intra-libssl bugs such as Heartbleed [34].

```
// CIV 1: option setting API leads to arbitrary R/W
ulong SSL_CTX_set_options(SSL_CTX *ctx, ulong op) {
  return ctx->options |= op;
}

// CIV 2: cross-API object SSL_CTX with function
// pointers leads to arbitrary execution
SSL *SSL_new(SSL_CTX *ctx) {
  /* ... */
  s->method = ctx->method;
  /* ... */
  if (!s->method->ssl_new(s)) // arbitrary execution
    goto err;
} /* ... */
```

Listing 4: Two libssl CIVs leading to arbitrary read, write, and execute impact. Both functions are exposed to the application.

```
void aesni_ecb_encrypt(const uchar *in, uchar *out,
  size_t length, const AES_KEY *key, int enc);
```

Listing 5: Prototype of internal encryption API from (I2), vulnerable to key extraction CIVs.

We applied ConfFuzz to both interfaces. *In all cases, we are able to extract keys out of the safebox leveraging a single CIV uncovered by our fuzzer*; we present three of them.

Listing 4 illustrates two CIVs found for I1, where libssl is safeboxed as a whole. The first CIV affects libssl's primitives to set/get SSL options, part of the official libssl API. Here, an untrusted caller compartment can control `ctx` and `op`, enabling for arbitrary read/write via the bitmask. In the second CIV, libssl executes callbacks provided by an untrusted object of type `SSL_CTX`, one of the three key vulnerable structures highlighted by Figure 3. These callbacks are very common across the libssl API and result in arbitrary code execution. Both vulnerabilities can easily be leveraged to extract the key.

Listing 5 illustrates a CIV found for I2, where keys are isolated at the internal interface. In this encryption primitive, callers control the `in` and `out` pointers, along with the key location `key`. By pointing `in` to the key and pointing `key` to a known value, attackers can either cryptanalyze the key out of `out`, or use the decryption function to extract the key.

There are systematic problems that make robust safeboxing at the libssl API difficult. This large API makes heavy use of state structs such as `SSL*` or `SSL_CTX*`. Sanitizing such structures is hard; it is likely that, even provided countermeasures from §III, the result will approach that of a rewrite of OpenSSL. Safeboxing at I2 is less complex but still requires redesign: encryption and decryption primitives must be made stateful to store the location of valid keys, checking that input and output buffers do not overlap key locations. Key creation and loading must also be carefully validated.

Insight: CIVs make it simple to extract SSL keys from an unmodified API safebox. Robust SSL key safeboxing requires redesign of the key API into a stateful entity.

2) Case Study: Sudo, Impact Beyond CIVs: Sudo is a strong target for compartmentalization, being high-risk (>100K LoC, many features) and sensitive (exploits lead to system privilege escalation). We considered several scenarios, one of them safeboxing the authentication API, which manages

```
int sudo_passwd_verify(struct passwd *pw, char *pass,
  sudo_auth *auth, struct sudo_conv_callback *cb) {
  /* ... abbreviated ... */
  sav = pass[8]; // read CIV
  pass[8] = '\0'; // write CIV
} /* ... abbreviated ... */
```

Listing 6: sudo CIV and CVE-2022-43995 manifesting when passed password with length below 8 Bytes.

password verification. Here, ConfFuzz found 5 CIVs, with read and `NULL` dereference impact. Investigating them, we realized that one CIV, shown in Listing 6, actually features R/W impact, and is reachable from user external input, i.e., it is also a vulnerability in non-compartmentalized contexts. This decade old issue manifests when users enter small passwords and was assigned CVE-2022-43995 after we reported it.

3) Case Study: Nginx Master/Worker Interface CIV: We studied the applicability of ConfFuzz to other compartmentalization models such as the Nginx master/worker manual separation. Here we assume that a worker has been compromised (e.g., from the network), and attempts to escalate to master privilege level. In this model we found a decade-old CIV that allows a worker to trigger memory corruption in the master[2]. The vulnerability affects a reliability feature of Nginx: when a worker crashes, the master forcibly unlocks shared memory mutexes hold by the worker to prevent deadlock. A malicious worker may corrupt the mutex before crashing itself to force the master to dereference a crafted pointer. This particular CIV is low impact due to control constraints in the mutex unlocking routine – bytes will only be overwritten if they match the worker's PID. Nevertheless, CIVs at such interfaces present a real risk: less constrained bugs are realistic and may pose a real privilege escalation threat.

Insight: CIVs also affect production-grade software and may be leveraged to mount privilege escalation attacks.

F. Conclusions

We stressed that CIVs widely affect unmodified software, but in varying proportions (Q1). Factors are structural; we elaborated on them with 5 central patterns and insights (Q2). We illustrated that API redesign will be necessary in many cases to achieve robust least-privilege enforcement (Q3). Finally, we showed that CIVs are impactful, exploitable, and elaborated with case studies on popular compartmentalization targets (Q4). Drawing from this, we discuss how to design interfaces that are by conception more CIV-resilient in §VI.

VI. (Re-) Designing Interfaces for Distrust

We showed that interfaces are not equally affected by CIVs because of interface design patterns. Next, based on previous sections, we discuss interface patterns that *reduce* compartmentalization complexity, and how to leverage them to design strong compartment boundaries, or refactor existing ones. These patterns do not eliminate the need for CIV countermeasures as detailed in §III; in their absence, these patterns reduce the number of CIVs, and in the presence of countermeasures, these patterns help palliate their limitations.

[2]https://github.com/conffuzz/conffuzz-ndss-data/blob/main/docs/nginx.md

When refactoring, many of the items listed below require major software redesign. We believe it is a necessary price to pay to obtain firm safety guarantees from compartmentalization.

1) Resources (memory, handles) must be clearly segregated: Memory ownership must be clearly defined, with each component responsible for allocating and freeing memory in their region: components must not rely on an another component's memory allocator (see Pattern 5, §V-D). Similarly, system resource handles (or handles to any third-party-managed resources) must not be shared. Take the example of `FILE*`: when shared, it is hard to determine who should release the handle and when, requiring complex, ad-hoc, and error-prone virtualization [5], [22], [42]. Instead, components should acquire and release their own handle: e.g., for `FILE*`, components should exchange file paths and call `open()` on their own.

2) Copy API-crossing objects: Shared objects must be systematically copied to avoid TV3 CIVs: it is very hard to safely use objects that can be concurrently modified by malicious compartments. More generally, concurrent usage of objects across compartment boundaries should be avoided as well, as it introduces the need for cross-compartment synchronization, which in turn opens for TV2 CIVs.

3) Simplify API-crossing objects: Compartment interfaces must not expose data that cannot be safely checked. This includes state information, which is hard to protect in the general case (Pattern 1, §V-D). In such cases, a layer of indirection can be added so that the object is not accessed directly, but through a set of primitives that can assert the safety of individual operations. If this is not possible, the interface is probably not a good compartmentalization boundary in the first place.

4) Trusted-components allocates: When a trusted component is passed a pointer to a buffer allocated by another component, it needs to either trust that component, or verify the pointer (which only privileged monitors can perform as it requires knowledge of the memory layout). Take the example of C-style strings: if a sandboxed callee allocates and returns through an API a string pointer, a trusted caller needs to verify the pointer's validity and the `NULL`-termination of the string. This problem can be eliminated by applying a *trusted-component allocates* policy, i.e., *caller-allocates* in sandbox scenarios, and *callee-allocates* in safebox scenarios. If the trusted component allocated the string buffer, it knows the maximum size of the string and can safely check for `NULL`-termination. In the case of mutual distrust, the involvement of a privileged monitor is necessary.

5) Trusted interface functions must be thread-safe: When a trusted compartment C_t exposes a function f_t (API function for safeboxes, callbacks for sandboxes) to an untrusted compartment C_u, C_t enables C_u to interfere with its control flow at any time. For example, C_u may interleave multiple calls to f_t and other API functions to trigger TV1 CIVs in C_t. Even when calls to C_t functions are serialized, these may perform callbacks back to C_u that will allow it to interleave other calls to C_t (a behavior that we observed with ImageMagick and libpng). Thus, trusted compartment functions must be designed *thread-safe* to support any concurrent calling. Alternatively, trusted interface calls should be strictly serialized and run to completion (no callbacks), a rather restrictive model.

6) Trusted interface functions must define & enforce ordering requirements: Similarly, if trusted interface elements $f_1 \cdots f_n$ have ordering requirements, then these must be clearly stated and enforced to further tackle TV1 CIVs. This may require safeboxed libraries to become stateful, where they previously relied on invoking undefined behavior if the caller did not respect ordering. When asynchronous behavior is suitable, event-loop-based designs may allow interface designers to shift as much control-flow leverage as possible out of the hand of attacker by processing the core of the callback in the main loop, in a way that is consistent with other external inputs (similarly to signal processing).

7) No sharing of uninitialized data: API-crossing uninitialized data must be systematically zero-ed to avoid DL1-2 CIVs (§III-A). Even sharing of properly checked objects can be unsafe if they have not been zero-ed at initialization, since compiler-added padding might remain uninitialized. Where applicable, zeroing should be compiler-enforced.

8) CIV checks first: As soon as one allows untrusted data to propagate unchecked through a compartment, it becomes hard to ensure that all checks are properly performed down the line (Pattern 4, §V-D), and encourages duplication of non-trivial checks, maximizing the likelihood of errors in present and future versions of the software. Worse, untrusted data might not even be used within but simply flow through a compartment to be used in another one, which might have variable trust assumptions on the compartment feeding it data (Pattern 3, §V-D). Copy and checks should therefore be performed on the data as soon as possible after crossing the API, and preempt all other functional checks.

VII. Related Works

a) Finding API Vulnerabilities: DUI detector [21] leverages static binary analysis, symbolic execution and dynamic taint analysis to detect pointer dereferences made by a security domain under the influence of another through an interface. Due to performance and scalability issues (emulation and symbolic execution), such approaches are hard to scale to large programs, large interfaces, large numbers of programs, or, as is the case here, high bug counts. Further, compiler-based static approaches are unsuited to scenarios like OpenSSL, where safeboxed components are implemented in pure assembly files. Other studies such as Van Bulck et al. [61], focusing on Trusted Execution Environment (TEE) runtimes, take a manual approach to identify interface vulnerabilities. Being manual, such approaches are limited in scope. In-memory fuzzing allows us to be faster and more scalable than static and manual approaches, enabling for a larger-scale CIV study. Fuzzing yields a subset of all CIVs present at an interface, which is a suitable limitation in our case.

Classical system call fuzzing [62] searches for kernel vulnerabilities at the system call API. This corresponds to *one specific safebox scenario* where the compartment API is the system call API. ConfFuzz is much more general, targeting arbitrary sandbox/safebox scenarios at arbitrary APIs. Similarly to system call fuzzing, Emilia [11] fuzzes for Iago [8] vulnerabilities, hooking at system calls and altering their return values to simulate a malicious kernel, corresponding to a sandbox model. Here too, ConfFuzz is much more general as it

1) can hook into arbitrary interfaces; 2) supports bidirectional fuzzing (sandbox/safebox); and 3) fuzzes the *full compartment attack surface* (callbacks, return values, shared data, function arguments) – whereas Emilia only fuzzes return values.

b) Finding API Misuses: APISan [68] studies existing software to infer semantic usage information for a given API (e.g., semantic relation on arguments/functions). Using that information, it searches for deviations to detect possible API misuses at the source code level. Such an approach is not suited to detect CIVs. First, CIVs can be present even when the semantics of an API are respected in the code: at runtime, a malicious compartment with code execution abilities can manipulate the program's execution in a way that does not respect the API semantics. Second, even if API semantics could be fully enforced at runtime, most CIVs would remain undetected because the semantics of unmodified APIs are generally unsuited to distrust, as we show in the paper. Nevertheless, as we highlight in §III, APISan's ability to infer API semantics may be leveraged to determine enforcement policies, provided a large enough set of API usage samples (usually available for popular APIs).

c) In-Memory Fuzzing: Unlike conventional fuzzing approaches that inject malformed data through a program's input channels (e.g., network), in-memory fuzzing [58] moves the fuzzer within the target using process instrumentation techniques. ConfFuzz is an in-memory fuzzer specialized for CIV fuzzing. ConfFuzz mainly differs from existing in-memory fuzzers [58], [55], [38] in that it 1) fuzzes in both ways (sandbox and safebox) – whereas existing in-memory fuzzers mostly correspond to safebox fuzzing, and 2) targets a different attack surface, the *compartment* attack surface – which, unlike usual in-memory fuzzers, also includes callbacks, return values, etc. To our knowledge, we are the first to use in-memory fuzzing with the goal of studying CIVs in unmodified software.

d) Interface-Aware Compartmentalization Frameworks: Compartmentalization frameworks provide a variable degree of support for protecting security domain interfaces. The vast majority of modern compartmentalization frameworks [67], [60], [19], [53], [35], [25], [45], [5], [51], [30], [29], [1] do not achieve more than basic ABI-level interface sanitization at security domain crossing, such as switching the stack and clearing registers. Combined with the fact that most also rely on relatively coarse-grain shared memory-based communication for performance reasons, this opens up a wide range of CIVs and was one of our motivations to develop ConfFuzz.

RLBox is a sandboxing framework for untrusted C++ software components. RLBox sanitizes sandbox data flow in a partially automated way: using static analysis and C++ type information, the framework can add certain checks automatically. When not possible, RLBox outputs compiler errors to require human intervention. Similarly, SOAAP [16] relies on code annotations and employs static analysis to flag possible data leaks. Both approaches are prone to human error due to manual effort. The CHERI [65] hardware memory capability model promises strong and efficient compartmentalization by extending RISC ISAs with capability instructions. Certain CHERI features (e.g., unforgeable pointers/capabilities, byte-level memory sharing) eliminate or mitigate some classes of CIVs. Nevertheless, CHERI is still a prototype [3]. We discuss the benefits and limitations of all three systems in §III.

Several TEE runtimes have been proposed [43], [7], [46], [14], [13] to transparently shield enclaves from the outside world by maintaining a secure interface. However, as demonstrated by several studies [61], [11] this cannot eliminate all CIVs, motivating fuzzers such as Emilia [11] and ConfFuzz. Before TEEs, sandboxing frameworks protecting applications from a malicious OS such as InkTag [20] and MiniBox [32] attempted to prevent Iago attacks by vetting/managing the memory mappings requests made by the protected program.

VIII. Conclusion

Breaking down monolithic software into compartments without reasoning about newly created interfaces leads to Compartment-Interface Vulnerabilities. This paper presented an in-depth study of CIVs. We proposed ConfFuzz, an in-memory fuzzing approach to investigate CIVs and their impact in compartmentalized software. Applying it to 25 applications and 36 libraries, we uncovered a large data-set of 629 CIVs from which we extracted numerous insights on the prevalence of CIVs, their causes, impact, and the complexity to address them: we confirmed how important CIVs should be to compartmentalization research, and highlighted how API design patterns influence their prevalence and severity. We concluded by stressing that addressing these problems is more complex than simply writing a few checks, proposed guidance on compartmentalization-aware interface design and adaptation, and motivated for more research towards systematic CIV detection and mitigation. We open-sourced code and data: https://conffuzz.github.io.

Acknowledgements

We thank the anonymous reviewers for their insights. We are also grateful to David Chisnall and Istvan Haller for their insightful feedback. This work was partly funded by a studentship from NEC Labs Europe, a Microsoft Research PhD Fellowship, the UK's EPSRC grants EP/V012134/1 (UniFaaS), EP/V000225/1 (SCorCH), the EPSRC/Innovate UK grant EP/X015610/1 (FlexCap), the EU H2020 grant agreements 871793 (ACCORDION) and 758815 (CORNET), and the NSF CNS #2008867, #2146537, and ONR N00014-22-1-2057 grants.

References

[1] Ioannis Agadakos, Manuel Egele, and William Robertson. Polytope: Practical Memory Access Control for C++ Applications. https://doi.org/10.48550/arXiv.2201.08461, 2022.

[2] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. CompartOS: CHERI Compartmentalization for Embedded Systems. https://arxiv.org/abs/2206.02852, 2022.

[3] ARM Ltd. ARM Morello Program. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello. Online; accessed June 25, 2020.

[4] ARM Ltd. Building a Secure System using TrustZone Technology. https://developer.arm.com/documentation/genc009492/c, 2009. Online; accessed Jan 24, 2021.

[5] Markus Bauer and Christian Rossow. Cali: Compiler assisted library isolation. In *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (ASIA CCS'21)*. Association for Computing Machinery, 2021.

[6] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel TOCTTOU protection. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22. USENIX Association, 2022.

[7] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, ATC'17, pages 645–658. USENIX Association, 2017.

[8] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13. Association for Computing Machinery, 2013.

[9] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, USENIX Security'05, 2005.

[10] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86), 2016. https://eprint.iacr.org/2016/086.

[11] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching iago in legacy code. In *Proceedings of 29th Network and Distributed System Security*, NDSS'22, 2022.

[12] Tevi Devor and Sion Berkowits. Pin: Intel's dynamic binary instrumentation engine – Pin CGO 2013 tutorial, 2013. https://www.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf.

[13] Fortanix. Fortanix enclave development platform. https://edp.fortanix.com/, 2022. Online; accessed Dec, 23 2022.

[14] Google. Asylo: An open and flexible framework for enclave applications. https://asylo.dev/, 2022. Online; accessed Dec, 23 2022.

[15] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. A Hardware-Software co-design for efficient Intra-Enclave isolation. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22, pages 3129–3145. USENIX Association, 2022.

[16] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15. Association for Computing Machinery, 2015.

[17] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. TypeSan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 517–528, 2016.

[18] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[19] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, ATC'19. USENIX Association, 2019.

[20] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems*, ASPLOS'13, pages 265–278, 2013.

[21] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Proceedings of the 20th European Symposium on Research in Computer Security*, ESORICS'15, pages 312–331. Springer International Publishing, 2015.

[22] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The endokernel: Fast, secure, and programmable subprocess virtualization. https://doi.org/10.48550/arXiv.2108.03705, 2021.

[23] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security'14, pages 957–972. USENIX Association, 2014.

[24] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC'06, pages 339–348, 2006.

[25] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the 12th European Conference on Computer Systems*, EuroSys'17. Association for Computing Machinery, 2017.

[26] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *Proceedings of the 2020 IEEE European Symposium on Security and Privacy*, EuroS&P'20, pages 309–321, 2020.

[27] Butler W. Lampson. Protection. *ACM SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[28] Michael Larabel. Linux picks up fix for latest "confused deputy" weakness going back to 2.6.12 kernel, 2021. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Confused-Deputy-2.6.12.

[29] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: Towards Flexible OS Isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'22, 2022.

[30] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Stefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. FlexOS: Making OS Isolation Flexible. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, HotOS'21. Association for Computing Machinery, 2021.

[31] Jialin Li, Samantha Miller, Danyang Zhuo, Ang Chen, Jon Howell, and Thomas Anderson. An incremental path towards a safer OS kernel. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems*, HotOS'21. Association for Computing Machinery, 2021.

[32] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX annual technical conference*, ATC'14, pages 409–420, 2014.

[33] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19, pages 177–194. USENIX Association, 2019.

[34] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 49–64. USENIX Association, 2016.

[35] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, CCS'17. Association for Computing Machinery, 2017.

[36] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 280–291. Association for Computing Machinery, 2015.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200. Association for Computing Machinery, 2005.

[38] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[39] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP'11. Association for Computing Machinery, 2011.

[40] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings*

of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pages 941–951. Association for Computing Machinery, 2015.

[41] MITRE, Common Weakness Enumeration. CWE 189: Numeric errors, 2022. https://cwe.mitre.org/data/definitions/189.html.

[42] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20. USENIX Association, 2020.

[43] OE SDK Contributors. Open enclave sdk web page. https://openenclave.io/sdk/, 2022. Online; accessed Dec, 23 2022.

[44] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kMVX: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, pages 559–572, 2019.

[45] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedins of the 2019 USENIX Annual Technical Conference*, ATC'19. USENIX Association, 2019.

[46] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. https://doi.org/10.48550/arXiv.1908.1114, 2019.

[47] OSS-Fuzz Project. OSS-Fuzz. https://google.github.io/oss-fuzz/, 2022. Online; accessed Dec, 23 2022.

[48] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX Security'14, pages 861–875. USENIX Association, 2014.

[49] Don Reisinger. Akamai heartbleed patch not a fix after all. https://www.cnet.com/news/privacy/akamai-heartbleed-patch-not-a-fix-after-all/, 2014. Online; accessed Dec, 23 2022.

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

[51] Vasily A. Sartakov, Lluis Vilanova, and Peter Pietzuch. CubicleOS: A library OS with software componentisation for practical isolation. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21. Association for Computing Machinery, 2021.

[52] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *Proceedings of the 31st USENIX Security Symposium*, USENIX Security'22. USENIX Association, 2022.

[53] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for RISC-V and x86. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security'20. USENIX Association, 2020.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC'12, pages 309–318, 2012.

[55] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Proceedings of the 2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[56] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'15, pages 46–55. IEEE, 2015.

[57] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'20. Association for Computing Machinery, 2020.

[58] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[59] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, S&P'13, 2013.

[60] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, USENIX Security'19. USENIX Association, 2019.

[61] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1741–1758. Association for Computing Machinery, 2019.

[62] Dmitry Vyukov. Syzkaller: adventures in continuous coverage-guided kernel fuzzing. BlueHat IL, https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/syzkaller%20Adventures%20in%20Continuous%20Coverage-guided%20Kernel%20Fuzzing.pdf, 2020. Online; accessed Dec, 23 2022.

[63] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. In *Proceedings of the 30th USENIX Security Symposium*, USENIX Security'21, pages 1647–1664. USENIX Association, 2021.

[64] Nicholas Wanninger, Joshua Bowden, Kirtankumar Shetty, and Kyle Hale. Isolating functions at the hardware limit with virtines. In *Proceedings of the 17th European Conference on Computer Systems*, EuroSys'22. Association for Computing Machinery, 2022.

[65] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, S&P'15. IEEE, 2015.

[66] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *SIGARCH Comput. Archit. News*, 19(4):6–13, 1991.

[67] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Proceedings of the 17th European Symposium on Research in Computer Security*, pages 859–876. Springer Berlin Heidelberg, 2012.

[68] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API usages through semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium*, USENIX Security'16, pages 363–378. USENIX Association, 2016.

[69] Zenong Zhang, George Klees, Eric Wang, Michael Hicks, and Shiyi Wei. Registered report: Fuzzing configurations of program options. In *Proceedings of the 1st International Fuzzing Workshop*, 2022.