Whole-Program Privilege and Compartmentalization Analysis with the Object-Encapsulation Model

Yudi Yang
Rice University
yudi.yang@rice.edu

Weijie Huang
Rice University
weijie.huang@rice.edu

Kelly Kaoudis
Trail of Bits
kelly.kaoudis@trailofbits.com

Nathan Dautenhahn
Rice University
ndd@rice.edu

***Abstract*—We present the object-encapsulation model, a low-level program representation and analysis framework that exposes and quantifies privilege within a program. Successfully compartmentalizing an application today requires significant expertise, but is an attractive goal as it reduces connectability of attack vectors in exploit chains. The object-encapsulation model enables understanding how a program can best be compartmentalized without requiring deep knowledge of program internals. We translate a program to a new representation, the *Program Capability Graph* (PCG), mapping each operation to the code and data objects it may access. We aggregate PCG elements into *encapsulated-object* groups. The resulting *encapsulated-objects PCG* enables measuring program interconnectedness and encapsulated-object privileges in order to explore and compare compartmentalization strategies. Our deep dive of parsers reveals they are well encapsulated, requiring access to an average of 545/4902 callable interfaces and 1201/29198 external objects. This means the parsers we evaluate *can* be easily compartmentalized, applying the encapsulated-objects PCG and our analysis to facilitate automatic or manual trust boundary placement. Overall, the object-encapsulation model provides an essential element to language-level analysis of least-privilege in complex systems to aid codebase understanding and refactoring.**

***Index Terms*—program analysis, least privilege, object models**

I. Introduction

Complexity and economics force applications to integrate many components and diverse stakeholders' data without sufficient boundaries, leading to *permiscuity*: an over-permissioning scenario where components have access to information they should not. An attacker who can compromise a component gains access to everything inside that component. Even with compartmentalization in place, components still require interface-level access to code and data *belonging-to* other components and dependencies. Patch Tuesday (and Exploit Wednesday) may exist in perpetuity as vendors continue to release incremental and insufficient patches for a neverending stream of supply chain attacks, ransomware, and any number of individual vulnerabilities [4].

The common notion of privilege within programs arises in the context of compartmentalization, privilege separation, or techniques to explore privilege. In the first two, a program is decomposed into separate compartments that each have access to a set of language objects and code interfaces. A few key efforts have explored semi-automated transformations [1, 2, 5–7], where the developer annotates the program and a compiler identifies data and code and separates. Unfortunately, they do not define privilege so that it can be extracted and quantified holistically in a codebase, and only consider specialized compartmentalization strategies that sandbox a region of buggy code or isolate some data to a small subset of the program.

We hypothesize that while many of today's codebases neglect isolation, their developers had to manage increasing complexity with modularity; and that *soft* modularity, even in languages that cannot enforce it, naturally emerges in a program's source structure and dynamic behavior. The objective of the *Program Capability Graph* (PCG) and the complexity and security analyses we present in this work is to expose and measure this emergent modularity, enabling future automated least-privilege program transformations, akin to a compiler's ability to optimize even when program complexity requires specialized knowledge exceeding individual developer capacity. The PCG represents an entire program in a language-agnostic, human-readable fashion. Analysing the PCG to measure program interconnectedness and *encapsulated-object* (aggregations of operations mapped to code and data following lexical scoping rules) privileges requires neither substantial security expertise nor deep context of the underlying codebase. The novel object-encapsulation model overall enables fully representing and analysing privilege relationships automatically even in large codebases.

First, we represent every low-level program operation as a PCG node. Since we produce the PCG via static analysis, it represents every possible control flow and access within a program and captures the upper bound of privileges required for the program to operate as designed. Second, we assign each program-level operation or object as belonging-to or operating on-behalf-of a specific authority context (encapsulated-object). Partitioning the PCG into encapsulated-objects represents a unique program compartmentalization strategy. Encapsulated-object authorities can be a simple lexical scope, like the file where the code is located as in our evaluation, or dynamic scopes such as the syscall context. Third, we compute the compartmentalization's suitability per encapsulated-object. Finally we present an interface for labeling program objects as *sensitive*, labeling program operations as *suspicious*, and calculating the overall exposure of sensitive objects in an encapsulated-objects PCG.

The privilege-set of an encapsulated-object E consists of the accessible code and data objects within it, and the objects belonging-to neighbouring encapsulated-objects that E can access. We consider E's percentage of exposed internal

state (public objects) its privilege upper bound. Additionally, the percentage of external objects that *E* can access within *other* encapsulated-objects represents the upper bound of all non-local privilege *E* may require. Together, these quantify *E*'s privilege within the system as a whole. Generally, the lower the percentage of public objects within an encapsulated-object, the more suitable the trust boundary partitioning that encapsulated-object from the rest of the PCG.

In this work, we demonstrate the potential of our model and analysis tooling by characterizing the security and interconnectedness of programs to evaluate how they can best be compartmentalized. Our core contributions include:

- A novel object-oriented representation of privilege within a program, the object-encapsulation model.
- The Program Capability Graph (PCG), which enables measuring program interconnectedness and encapsulated-object privileges.
- A study of parsers for complex and commonly exploited input formats, which finds many such parsers are well encapsulated and can therefore be easily compartmentalized, applying the encapsulated-objects PCG and our analysis to facilitate automatic or manual trust boundary placement.
- A detailed analysis of NGINX demonstrating that an object-oriented PCG representation of C-based systems leads to useful compartmentalization insights, *e.g.,* that more than 50% of the objects in NGINX are only internally used in the file where they are defined.
- A detailed *sensitive-object* analysis that labels syscalls as sensitive and evaluates their accessibility throughout a given program under diverse compartmentalizations.

While our representations are automatically extracted, the resulting graphs and the metrics we derive from them are human-readable to facilitate understanding and validation. Beyond these concrete demonstrations, we believe the object-encapsulation model enables analysis for many purposes: enhancing program understanding by finding sharing anti-patterns, understanding how code interacts, measuring how much authority any given component *should* have, as well as assessing program complexity and aiding in refactoring.

II. Background

Our primary objective is to present a general purpose representation and interface for analyzing reachability within a system. Specifically, we aim to characterize the complexity of privileges at the granularity of individual code and data objects within a system and in the future enable automated least-privilege program transformation. The primary challenge is to do this without expert annotations with a concrete representation that can be audited and understood by humans.

Program-Mandering (PM) encodes privilege metrics to explore optimal points at which to compartmentalize [9]. Unfortunately, despite providing one of the first procedural methods for representing and using a privilege metric, PM lacks the ability to be used for custom analysis and cannot be applied to programs in different languages.

μSCOPE introduced a capmap representation of privilege as a low-level access graph to enable dynamic privilege tracing and analysis [13]. While μSCOPE applied a low-level definition of privilege similar to that of the PCG, the capmap is derived dynamically and can only describe the required *lower* bound of privilege in a codebase. In contrast (and complementary) to the capmap, our object-encapsulation model enables exploring the *upper* bound on privilege in a codebase.

We primarily seek to enable whole-program analysis of the upper bound of required component privilege in large existing codebases. Several existing intermediate representations and object models attempt to do this [11]. While these object models can prescriptively define complexity, our goal of analysing legacy systems requires a more holistic and *descriptive* way to measure privilege across many components and dependencies.

Developers have long been able to construct and query a program dependence graph (PDG) [3] to answer the question of what other program components can access a particular component in the form of graph reachability. However, this approach to measuring privilege does not scale to granular analysis conducted holistically over large systems. The algorithmic complexity of evaluating interconnectedness and reachability over a PDG for *every* low-level operation and object in a program becomes rapidly infeasible with codebase size and does not scale to the size of systems we analyse.

Pidgin and other recent work on authority contexts represent and explore program privilege over a PDG [8, 12]. However, these run into the intrinsic complexity limitations of analysis over the PDG and either work only for explicit languages that encode them, or merely enable the ability to inspect specific privileges. They cannot provide whole-program privilege metrics for upper-bound complexity analysis. The PDG includes information our analysis does not use, which accordingly would require additional memory to store. Our PCG, in contrast, is tailored to security-oriented privilege analysis over large codebases.

III. The Object-Encapsulation Model

Our model represents privilege throughout a program in terms of low-level operations, objects, and how they interact. The goodness-of-fit of encapsulation, *i.e.,* the relative number of private objects and interfaces given the encapsulation partitioning applied over the PCG, indicates how well the encapsulation strategy in question realizes the principle of least privilege.

Our approach borrows from the object-capability model of computation [11], where a program comprises instances of code (as computation) and data that are aggregated into encapsulated-objects. Access rights are granted to an instance if the data *belongs-to* the object or is obtained through the program's reference graph. Since the objective of this paper is to analyze privilege without assuming an explicit definition for it, we limit the scope of this design to what is necessary to infer privilege-based relations. This leads to an intermediate representation that captures privilege as capabilities to perform

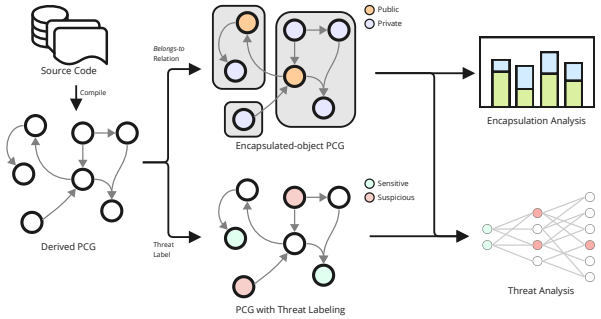


Fig. 1: The phases of PCG construction and analysis. Code objects are mapped to nodes in the PCG that perform low-level operations (read, write, call, return, alloc, free) on data object nodes. Nodes are aggregated into encapsulated-objects, which are then used to measure the degree to which privilege is reduced under that encapsulation. Threat modeling labels nodes as *sensitive* (data) or *suspicious* (operation) and measures exposure under a given encapsulation.

specific operations on objects and a method for labeling each of these by the authority context to which it belongs. Figure 1 shows the design and generation process of each major phase of the model and approach.

A. The Program Capability Graph

Without explicit developer annotations our approach must be descriptive, meaning derived bottom-up from readily available elements within the program itself, rather than prescriptively expressed top-down. An operation is a distinct instance of computation that can be labeled as *belonging-to* a lexical scope or as operating *on-behalf-of* some dynamic scope and which performs some type of access on an object. An object is a unit of program storage as defined by the language or runtime. An encapsulated-object is a grouping of objects (code and data) under a single authority context.

The PCG includes *primitive* operations intended to represent any type access control rights. Objects are created by computation that `alloc` the object (either statically or dynamically) and are destroyed by computation that `frees` the objects. Objects can receive `read` or `write` operations which bestow privileges to the computational context performing the operations. Objects can additionally receive a `call` or `return` which change the overall authority context along with the current scope.

The PCG nodes consist of all the code and data objects of the program. The code objects include functions and function pointers. The data objects include global variables and local variables. Since parameters are stack objects just as local variables. We also consider parameters as a special type of local variables, with an inherit accessibility through function calls. In addition, function pointers are considered both code objects (when the program performs indirect calls) and data objects (when the program accesses the address of the pointer, such as assignments.) The PCG edges determine the privilege-set of all the functions. The edges are incident from code objects to all objects, representing that the function accesses those objects through reads, writes, pointer dereferences and aliasing, or function calls (including both direct calls and indirect calls). There are edges from a function to all of its local variables including parameters, edges from a function to global variables it accesses (with annotation determining read access or write access,) and edges from a function to other functions that it calls. Therefore, the PCG is a map of accessibility from code objects to code and data objects and the maximum privilege-set required by the function.

B. Authority Context Labels for Encapsulated Objects

Our objective with the PCG is to provide a low-level model for building other analysis. The question now becomes that how do we define the belongs-to or on-behalf-of relations, i.e., how could we partition the PCG reasonably without sufficient codebase expertise. In fact, there is not a single "correct" way to partition the objects: we could partition them lexically by files; we could use a grouping by directory; we could use dynamic on-behalf-of runtime information; and we could manually construct the relation. We would like to ensure that the encapsulation model is programmable, language-independent, and security researchers can explore their own relations.

The *Authority Context* is a label on the PCG nodes (objects) that represents a unique encapsulated-object. The labeling of each object in the PCG determines which encapsulated-object it belongs-to. This labeling automatically satisfies the properties of a belongs-to relation: it is many-to-one, and it covers every code and data object.

An **encapsulated-objects PCG** is a PCG with labels assigned to each object. For example, a file-based lexical-scoped encapsulation has each object label assigned to be the file it is defined within. We have designed and listed several encapsulations in Section IV.

C. Encapsulation Analysis

With the encapsulated-objects PCG, we are able to quantify the privilege-set (PS) of each encapsulated-object and evaluate the degree to which the encapsulation provides information hiding and thus reduces permiscuity. The objective of encapsulation analysis is to measure the degree to which a given encapsulated-objects PCG encapsulates objects within a program. The value is that well encapsulated systems can be compartmentalized at those boundaries and thus lead to improvement in whole program least-privilege. In this section we present metrics for providing whole program analysis of a given encapsulated-objects PCG and describe how and why those metrics characterize meaningful least-privilege properties.

1) The Privilege Set Metric: The privilege-set (PS) is the set of all objects that a given encapsulated-object may access and which is given directly by both the PCG and encapsulated-objects PCG. We quantify privilege as the number of objects accessed ($|PS|$). If the object-encapsulation model analysis is field-sensitive then this number can also capture the complexity of the objects being shared as well. To

make meaningful analysis of the metrics we examine the ratios of internal:external object access from an encapsulated-object and the ratio of private:public objects in the encapsulated-object.

a) The PS-From Ratio: We can characterize the overall privilege for a given encapsulated-object as the PS-From Ratio (PSFR) metric, which characterizes the total number of each of the following classes of objects relative to the total number of object in the program: local data, local code, external data, external code, inaccessible data, and inaccessible code. The value of this metric is that it allows us to investigate the privilege footprint from a given encapsulated-object and includes the inaccessible objects so we can estimate how localized the privileges of a given component are. The PSFR is characterized as a percent of total and thus the ratio of the six classes of objects relative to the total objects sum to 1.

Since the objects include objects passed as arguments to function calls, the object number is more representative than function number. Thus, we rank encapsulated-objects with highest percentage of external objects as the largest, and the object with lowest percentage of accessed external objects as the smallest external set.

A encapsulated-object with higher rank (higher external object references) suggests that it is worse according to the Privilege Set metric, as they are exposed to more external data that a security problem within an encapsulated-object is easier to trigger violations in another encapsulated-object. We should ensure objects with higher ranks should be hardened as they have more access privilege.

b) The External Access Ratio Metric: We can separate the objects and functions in a software system in a different perspective. Some of them are used only internal to the encapsulated object, *i.e.,* private objects. In contrast, objects and functions that are used inter-encapsulation are called "Public Objects" and "Public Functions". The External Access Ratio (EAR) captures the total ratio of externally accessible objects relative to the total number of objects in the program. Thus, for better encapsulation and demonstrating less sharing complexity this ratio should be lower. A lower sharing complexity and ratio means that specifying policies is simpler, transforming the system is easier, performance might be better (depending on hot code paths), and simpler to reason about security implications if a compromise occurs.

The exact implementation of both metrics and the encapsulated-objects PCG are described in Section VI.

D. Threat Modeling and Analysis Interface

While encapsulation analysis gives us useful exploration and measurement of properties we intuitively map to least-privilege, they fall short of demonstrating real security gains. The last element of the object-encapsulation model is an interface for expressing and measuring the real threat under a given encapsulated-objects PCG to sensitive data objects from suspicious operations. In this exploration, a developer labels sensitive objects, code interfaces or data objects, from the PCG and optionally labels suspicious operations and then launches a series of analysis. Whole program analysis, measures the degree to which the sensitive objects are accessible to all encapsulated-objects, and specific analysis determine whether access may occur between any suspicious operations. The overall benefit of this analysis is that it enables custom methods that can automatically apply across any system if the objects are common to runtimes, such as the syscalls.

1) The Access Distance Metric: The access distance metric, measures how many and which hops must an attacker take to get to a sensitive object given a specific encapsulated-objects PCG. If an attacker can gain control of an encapsulated-object, it can be used to attack sensitive objects. The closer the encapsulated-object is to the sensitive object, the more likely it will be exploit and used to attack the sensitive object. This security property can be formulated as the access distance, i.e. the shortest distance between the subject and some sensitive object.

2) The Exposure Reduction Metric: For each object labeled as sensitive, we define its exposure as the number of encapsulated-objects that may access it. If there were no protection mechanism, the exposure of a sensitive object would be the number of total number of encapsulated-objects in the program because any memory corruption could allow an attacker to access the sensitive object regardless of the code dependency. Intuitively, reducing the exposure of sensitive objects leads to better security properties, and it can achieved by applying some encapsulation techniques. In this case, the exposure is reduced to the number of encapsulated-objects that could directly access the sensitive object. To quantify the gain of security for a sensitive object, we measure the reduction as the ratio of exposures after and before encapsulation. By analyzing the exposure reduction of each sensitive object, we can grasp a concrete attack view of an encapsulation's security properties.

IV. Exploring Encapsulations

The interface for constructing the encapsulated-objects PCG enables first of its kind analysis based on exploring encapsulation boundaries and measuring how well they minimize public access to objects. In this section we present three methods for labeling encapsulated-objects.

A. File-Based Lexically Scoped Encapsulation

In file-based scoping, each object is automatically assigned to an encapsulated-object based on where the line of code allocating the object resides. By assigning objects according to this belongs-to rule, we can evaluate the degree to which a codebase follows this pattern and whether or not it provides meaningful encapsulation boundaries for minimizing privilege. Heap objects are labeled based on the file of the allocation site, treating the allocation site as a defacto type (even for `void *` objects). All objects defined in header and implementation files with matching base names are labeled with the basename of the file. Objects defined in header files without associated implementation files are labeled as belonging-to the header file encapsulated-object.

B. Manual Encapsulation

To show off a directed encapsulation, we manually assign encapsulated-objects based on labeling a set of source files. For example, we could devise a parser encapsulation by labeling all parser files and then using the lexically scoped model to assign all associated objects to the parser encapsulated-object. We apply this to explore and compare parsers in Section VII-B.

C. Dynamic Syscall Encapsulation

In contrast to the lexically scoped belongs-to assignment we can also explore dynamically scoped on-behalf-of assignments. We dynamically trace the PCG operations in the Linux Kernel and label each operation with the active system call context. This means that all operations and objects are labeled by the syscall context at the time of the operation. Just as the file-based lexical scopes create encapsulated-objects under the belongs-to relation, so too does the dynamic syscall context assignment under the on-behalf-of relation. Accesses from non-allocating syscalls make the accessed object public and therefore expose privilege across that dynamic context boundary.

V. Exploring Threat Models

To demonstrate attack specific analysis, we model threats by labeling syscall interfaces as sensitive and measure their accessibility under the file-based belongs-to assignment. Syscalls are a common attack target. They enable an attacker to progress to other parts of a system using the privileges of the exploited process. By labeling syscalls we enable automated analysis of the widely targeted syscall attack surface common in all Posix processes, meaning it can be reused beyond a single application threat analysis. For example, a common security policy is to debloat or apply seccomp filters to reduce the use of syscalls for further propagating an attack. With our syscall labels and analysis methodology, these evaluations could be automated, systematically measuring any such mitigation while enabling comparison to alternatives.

VI. PCG Construction

We present the design and implementation of translating C to the PCG. While we demonstrate for C, the PCG can be derived from any source language including via binary instrumentation (as demonstrated with our on-behalf-of syscall contexts). We use LLVM to build the PCG [10]. The analysis translates a program from its LLVM bitcode to PCG, exported in JSON format.

a) Deriving Code Objects: A node is created in the PCG for each function declaration.

b) Data Object Extraction: A node is created for each of three types of objects: **Global Objects** are defined in the global scope of the program and can be obtained by iterating through the LLVM globals list; **Local Objects** are defined by debug information attached to LLVM instructions (`llvm.dbg.declare` instructions) and obtained by iterating over the instructions; **Function Parameter Objects and Pointer Values** are located through the `llvm.dbg.declare` instructions.

TABLE I: Read / Write Relation for LLVM Instructions and Operations

Instruction	Read / Write
LoadInst	read
StoreInst	write
GetElementPtrInst	Depends on forward uses
CallInst	read
CallBrInst	read
ICmpInst	read
AtomicRMWInst	write
AtomicCmpXchgInst	write
ReturnInst	read
SelectInst	read
PtrToIntInst	read
PtrToIntOpr	read
GEPOperator	Depends on forward uses
BitCastOperator	read
PHINode	read
Constant	N/A

c) Authority-Context Labeling: We label each object with the absolute file path based on the debug information. The local and parameter variables belong to the same encapsulated-objects as their parent functions.

d) Privilege Set Extraction: Operations are labeled by the full path and source line of the code the occur in. An edge is created in the PCG when an operation may access a given object, which occurs in the following scenarios:

- **Functions Access Global Variables**: We iterate through every use of global variable and mark it as either a read or a write. We map the relation to separate LLVM instruction types and operator types in Table I. Specifically, when we meet a GEP instruction or operator, we need to check the uses of the GEP: if those uses contain a store instruction, then the GEP instruction or operator is an write operation; otherwise, it is an read operation.
- **Functions Access Local Variables**: Local variables including parameters are accessible to a function if they are defined within the function, or they are indirectly used (aliased). The indirection will be discussed later.
- **Object Aliasing**: If an object is in the alias-set of a data object, it is accessible from the other data object. We extract the alias set using the SVF alias analysis [14]. A function is accessible to every object which is in the alias set of the object it is accessible to.
- **Function Calls**: Direct and indirect function calls means the caller gain the accessibility of callee. If there are multiple callee from the indirect calls from the alias analysis, all callee are accessible from the caller to model the maximum privilege-set.

A. Metric Evaluation

The evaluation program is written in Python. It takes in the Program Capability Graph in JSON format as input, which is directly passed from the PCG construction program, and outputs the evaluation results in CSV format.

a) The Accessibility Algorithm: We calculate the accessibility of each encapsulated-object in the subject program. It is calculated by the following steps:

1) For each function, extract the required code and data objects.
2) Deduce the set of external encapsulated-objects required for each function.
3) Calculate each object's field count.
4) For each encapsulated-object, union all of the sets above, and calculate the required capabilities in terms of three metrics: **Encapsulated Object Count**, **Object and Function Count**, and **Field Count**.

b) The PS-From Ratio Algorithm: We identify the parser encapsulated-object by manually inspection and matching files with parser or non-parser encapsulated-object.

We calculate the PS-From Ratio of the subject program by the following:

1) **Local Data**: All global variables and local variables (including parameters) declared in the parser encapsulated-object.
2) **External Data**: All accesses to global variables and all variables in the alias set of local objects declared in the non-parser encapsulated-object.
3) **Inaccessible Data**: Calculated through subtracting local data and external data from total data objects (all global variables and local variables).
4) **Local Code**: All functions declared in the parser encapsulated-object.
5) **External Code**: All functions declared in the non-parser encapsulated-object called by functions declared in the parser encapsulated-object. Includes both direct and indirect calls.
6) **Inaccessible Code**: Calculated through subtracting local code and external code from all code objects declared in the subject program.

c) The External Access Ratio Algorithm: We calculate the EAR of the subject program by the following steps:

1) For each function, if they are called by functions belongs to other encapsulated-objects, we add the function to the *public function* set. Otherwise, we add the function to the *private function* set.
2) For each global variable, if in the PCG, there is a reference by any function or its alias set contains any variable belongs to other encapsulated-objects, we add the global variable to the *public global variable* set. Otherwise, we add the global variable to the *private global variable* set.
3) For each local variable. If it is a parameter or its alias set contains any variable belongs to other encapsulated-objects, we add it to the *public local variable* set. Otherwise, we add it to the *private local variable* set.

d) The Exposure Algorithm: For a given encapsulated-objects PCG, we first label all sensitive objects, then do a tiny modification on the encapsulated-objects PCG so that the sensitive objects and the rest of the encapsulated-objects are regarded as vertices. Since the modified graph is still directed, we can obtain the exposure of these objects trivially by calculating the in-degree of the corresponding vertices.

TABLE II: Evaluated Programs and Libraries

Parsing Libraries	Software System
libroxml	Nginx
jansson	CPython
facil.io	PHP
libxml2	MuPDF
json-c	gimp

TABLE III: PCG Generation Runtime

Target Program	Generation Runtime
libroxml	0.854s
jansson	0.824s
facil.io	8.290s
libxml2	2m58.852s
json-c	0.884s
Nginx	6m3.503s
CPython	119m24.766s
PHP	180m44.243s
MuPDF	60m43.136s
gimp	1m26.717s

VII. Evaluation

With the object-encapsulation model, we would be able to create the encapsulated-objects PCG for every software written in C with a suitable Authority Context Label. We apply the object-encapsulation model to explore the upper-bound privileges required for the programs listed in Table II using the Lexical Encapsulation Model compared to the privileges they have without encapsulation.

We measure the runtime for PCG generation program, the result is shown in Table III. The runtime is still significant due to the inclusion of SVF alias analysis during generation.

The accessibility statistics result is shown in Table IV. It provides statistics on the average number of encapsulated-objects, objects, and scalars accessible and accessed per each encapsulated-object. In the metric, the object counts include code objects (number of functions), global variables, and local variables including parameter objects. The scalar counts expand each compound data structure (structs and pointer to structs in specific) by its correlated field numbers recursively. For example, a struct with 3 fields and each field is a struct with 2 fields, the scalar count is $3 * 2 = 6$. For recursively defined data structures, we solve the recursive issue by only counting the top-level pointer fields. In subsequent pointer fields, we only count the pointer as one scalar. The Percentage Reducible row calculates how many unnecessary accessibility of encapsulated-objects can be removed by applying the Lexical Encapsulation Model.

For example, Nginx has 129 encapsulated-objects. As a software system written in C, Nginx currently has its total count of capabilities equals $129 \times 128 = 16512$. The object-encapsulation model suggests that only $129 \times 61 = 7869$ of the 16512 capabilities are directly required. Thus, in an object-encapsulation model, we can remove $16512 - 7869 = 8643$ capabilities, which is 52.60%. This means that if we apply automatic transformation on Nginx based on the encapsulated-objects PCG, we would reduce the jump targets and exposed data to 47.40% of the original program. Thus, if the Nginx

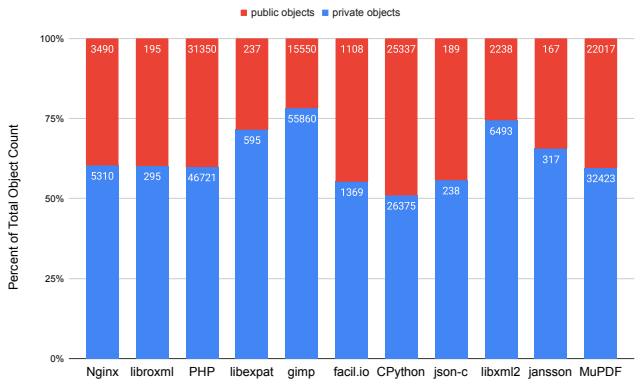


Fig. 2: EAR Analysis: showing public vs. private data objects for all encapsulated-objects. More than 50% of data objects are private.

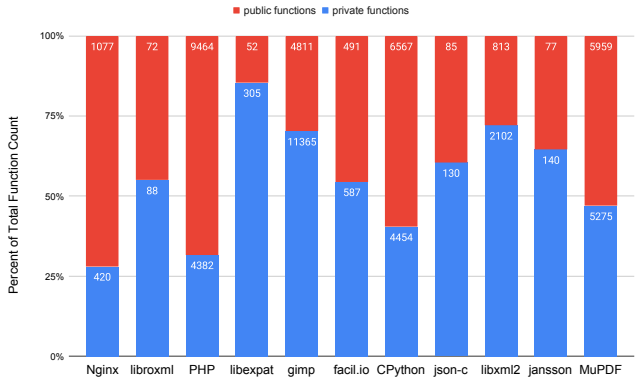


Fig. 3: EAR Analysis: showing public vs. private code objects for all encapsulated-objects.

program exposes an ROP attack or a buffer overflow, it could only directly returns to or modify an average of 47.40% of the functions and data.

In general, we observe that more than 50% of the encapsulated-object capabilities are reducible. This is a tremendous step given that we are only using fully automatic transformation without requiring an expert developer, compiler engineer, or security engineer. Another result from Table IV is that the larger the project is (as respective to file numbers and lines of code,) the more percentage of capabilities can be automatically reduced. This is because the required capabilities is linearly correlated to the size of the project, while the total capability number is a quadratic correlation. This suggests that the object-encapsulation model is useful in trimming unused capabilities, especially in large software systems.

A. EAR Analysis

Figure 2 and 3 show the EAR for the systems and libraries we analysis.

In a software system with more private objects and private functions, the system would be better encapsulated by exposing fewer interfaces and global variables. For example, in the EAR graph for objects, more than 50% of total objects are private for all programs, indicating that compartmentalization can be made for the programs and we automatically have at least 50% objects invisible to other encapsulated-objects. In Nginx, almost 75% functions are publicly used in other encapsulated-objects. This means that if we provide a sanitation for those interface functions, the sanitation would be abundantly used and might affect the performance.

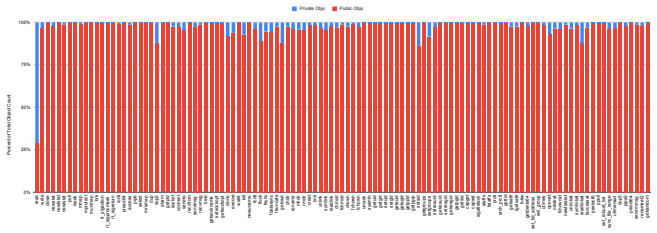


Fig. 4: EAR Analysis for dynamic objects System calls alone are a bad context for encapsulation.

1) EAR Analysis with Dynamic Syscall Encapsulation: Figure 4 show the EAR for dynamic system call encapsulations for data objects, while the code objects exhibit a similar pattern that most objects captured in the dynamic analysis are public objects.

We observe from the result that most of the objects from system calls are public, and that system calls are not suitable targets for choosing encapsulation boundaries. The result explains that using the EAR Analysis, we can evaluate encapsulation models and determine whether the encapsulation boundary is suitable.

B. Parser Evaluation with PSFR

We can use the PSFR metric to analyze different parsers and provide comparisons on parsing libraries. For the parser evaluation, we require a modified Lexical Encapsulation Model: in the Parser Model, we manually select files in the Lexical Encapsulation Model that contain parsing functionality to form, together, a "parser" encapsulated-object; all other files form the "external" encapsulated-object. The analysis focus on the availability of external encapsulated-object to the parser encapsulated-object.

Figure 5 shows the normalized parser compartmentalization statistics for the software systems and libraries we have analyzed. In each program, the parser compartment comprises one or more files annotated as the parser. Figure 5 depicts the PSFR.

The PSFR analysis shows that most parsers only has access to a limited amount of external code and data objects. This gives us an idea that the parser is a suitable encapsulation boundary to be used for compartmentalization.

C. Threat Analysis

We adopt the metric discussed in Section III-D to analyze the systems listed in Table II. We label the system interfaces

TABLE IV: Accessibility for an Average Encapsulated-Object with Object-Encapsulation Model

System	# of Files	CLOC (in C)	Encobjs.*	Objects*	Scalars*	% Reducible
libxml2	43	215,796	22	176	31,292	48.12%
jansson	14	7,529	4	15	607	70.88%
libroxml	12	7,205	3	10	1,199	74.24%
libexpat	8	25,452	2	13	1,259	76.79%
facil.io	30	22,602	6	39	851	78.28%
json-c	14	8,501	2	11	127	82.97%
Nginx	129	138,467	61	484	96,464	52.60%
CPython	262	530,504	106	6,327	394,867	59.53%
MuPDF	542	860,824	175	1,045	62,128	67.62%
PHP	479	1,162,182	144	2,167	236,744	69.82%
gimp	1,103	894,939	8	21	138	99.29%

* Accessiblity per encapsulated-object in Average

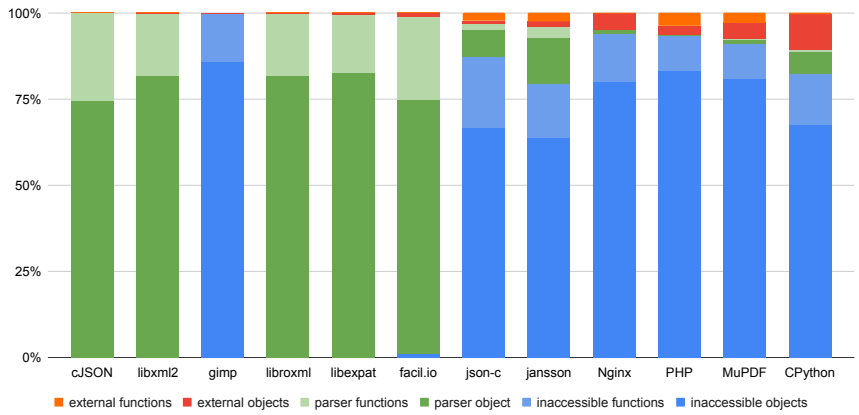


Fig. 5: Comparative analysis of parsers using PSFR.

(i.e. glibc wrappers for system calls) that are associated with process execution, file system access, and memory management as sensitive. Some system calls have several variants (e.g. execve, execv, execveat, etc.), and we group them together as one system interface (like exec*).

Table V shows the exposure reduction of the programs. The number of encapsulated-objects vary among the programs, from 16 to 4086. However, only a few of them are directly related to the system interfaces. This shows that given file-based encapsulation, we could mitigate most threats to sensitive syscall objects.

Most privilege-escalation attacks focus on leveraging execve for loading arbitrary code. In this experiment, we analyze the access distances of encapsulated-objects to execve in Nginx, which are shown in Figure 6. The red vertex on the left is execve itself, and the blue ones are the encapsulated-objects grouped by the access distance (ascending from left to right). Although there are 111 encapsulated-objects that may indirectly rely on sensitive interfaces, only one of them (src/os/unix/ngx_process.c) may invokes it directly. With file-based encapsulation policies, we could disallow any malicious calls to execve except that file, and hence mitigate most privilege-escalation threats.

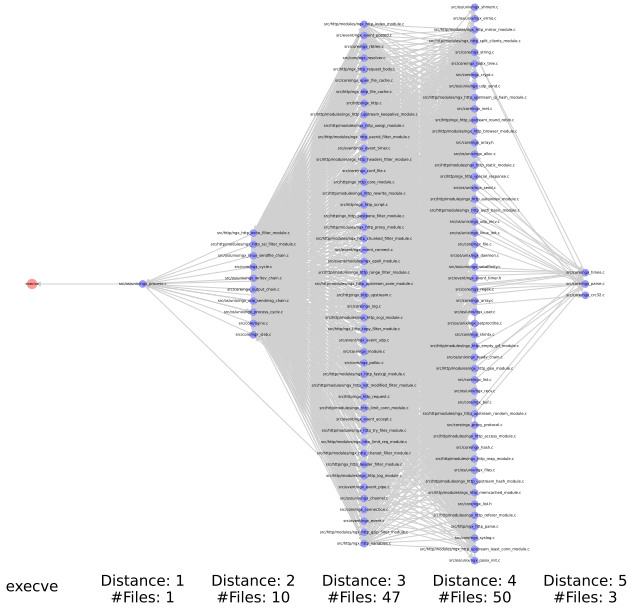


Fig. 6: Access Distance of encapsulated-objects to execve in Nginx, where execve is also encapsulated

TABLE V: Exposure Reduction of System Interfaces (After / Before Encapsulation)

System	System Interfaces										
	system	fork	exec*	popen	open*	read*	write*	ioctl	dup	mmap	mprotect
libexpat	0 / 16	0 / 16	0 / 16	0 / 16	1 / 16	1 / 16	0 / 16	0 / 16	0 / 16	0 / 16	0 / 16
libroxml	0 / 40	0 / 40	0 / 40	0 / 40	2 / 40	1 / 40	1 / 40	0 / 40	1 / 40	0 / 40	0 / 40
jansson	0 / 37	0 / 37	0 / 37	0 / 37	3 / 37	2 / 37	1 / 37	0 / 37	0 / 37	0 / 37	0 / 37
json-c	0 / 56	0 / 56	0 / 56	0 / 56	2 / 56	2 / 56	1 / 56	0 / 56	0 / 56	0 / 56	0 / 56
facil.io	0 / 102	1 / 102	0 / 102	0 / 102	4 / 102	5 / 102	4 / 102	0 / 102	0 / 102	1 / 102	0 / 102
libxml2	0 / 141	0 / 141	0 / 141	0 / 141	3 / 141	3 / 141	4 / 141	0 / 141	1 / 141	0 / 141	0 / 141
Nginx	0 / 260	2 / 260	1 / 260	0 / 260	10 / 260	4 / 260	4 / 260	4 / 260	2 / 260	2 / 260	0 / 260
CPython	1 / 600	1 / 600	1 / 600	0 / 600	4 / 600	6 / 600	4 / 600	2 / 600	1 / 600	2 / 600	1 / 600
MuPDF	1 / 613	0 / 613	0 / 613	0 / 613	8 / 613	8 / 613	5 / 613	0 / 613	0 / 613	0 / 613	0 / 613
PHP	0 / 1056	2 / 1056	1 / 1056	4 / 1056	16 / 1056	8 / 1056	7 / 1056	0 / 1056	2 / 1056	5 / 1056	1 / 1056
gimp	0 / 4086	1 / 4086	1 / 4086	0 / 4086	4 / 4086	4 / 4086	1 / 4086	0 / 4086	0 / 4086	0 / 4086	0 / 4086

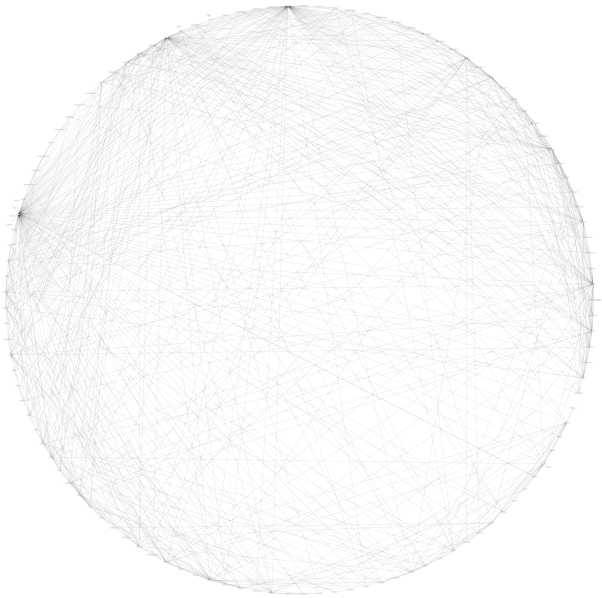


Fig. 7: Nginx Object-Encapsulation Model

TABLE VI: Nginx External Read-Only Global Variable Rank

#	Encapsulated-Object	Ext. R/O ↑
1	os/unix/ngx_process_cycle.c	47
2	event/ngx_event_timer.h	43
3	event/ngx_event.c	41
4	http/ngx_http_core_module.c	40
5	http/ngx_http_upstream.c	39
6	http/ngx_http_request.c	38
7	core/nginx.c	30
8	event/modules/ngx_epoll_module.c	27
9	http/modules/ngx_http_proxy_module.c	27
10	http/ngx_http_variables.c	25

TABLE VII: Nginx External Writable Global Variable Rank

#	Encapsulated-Object	Ext. R/W ↑
1	os/unix/ngx_process_cycle.c	36
2	core/ngx_times.c	30
3	core/nginx.c	26
4	os/unix/ngx_process.c	20
5	event/ngx_event.c	17
6	event/modules/ngx_epoll_module.c	16
7	core/ngx_regex.c	9
8	core/ngx_cycle.c	7
9	event/ngx_event_accept.c	6
10	http/modules/ngx_http_ssi_filter_module.c	6

D. Use Cases of object-encapsulation model

1) Nginx Analysis: Figure 7 is a overview visualization of the encapsulated-objects Program Capability Graph to apply the object-encapsulation model on Nginx. We have analyzed the Nginx system based on the encapsulated-objects PCG, without manually looking at its actual code, assuming that we are security programmers with no knowledge of Nginx source code that would like to analyze and harden the software.

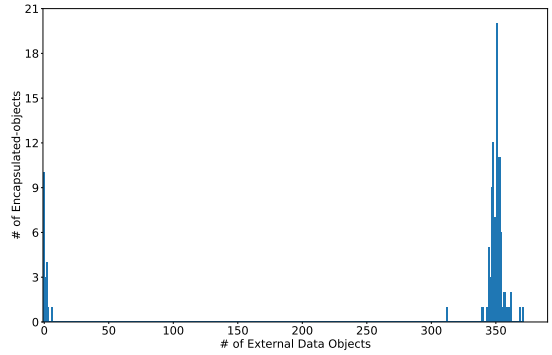
An observation on Figure IV suggests that some encapsulated objects are more referenced by other encapsulated objects, and some reference more encapsulated objects. In both cases, we should take extra effort hardening the objects as they bear more security implication. For example, Table VI and Table VII show the encapsulated-objects with highest number of reads and writes to global variables. Therefore, the Nginx engineer would probably put more effort ensuring the security of the listed files, such as `core/ngx_process_cycle.c`. Thus, we could define a metric for each encapsulated object by ranking the degree of read-only and read-write global variables.

In order to further understand the number of capabilities among each encapsulated-objects in Nginx, we collect the number of external code and data object accessible to each encapsulated-object. We draw two bar charts of the absolute numbers in Figure 8a and 8c. We also draw two cumulative distributions of encapsulated-objects for the number of accessible code and data objects in Figure 8b and 8d.

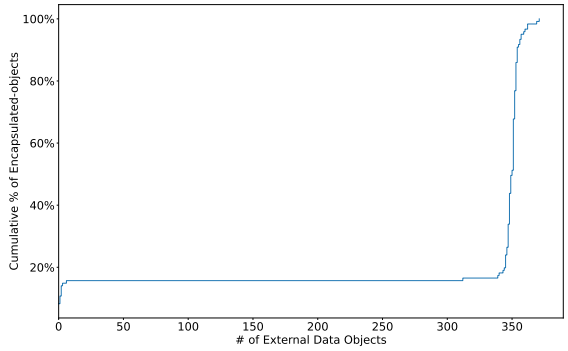
The result shows a bipolar structure: a large number of encapsulated-objects is accessible to less than 10 data objects and less than 40 code objects; in contrary, there is still a large portion accessible to more than 300 data objects and more than 600 code objects. A further discovery on the encapsulated-objects PCG shows that the encapsulated-objects with large external code and object accessible counts are common in object within a single large alias set.

Overall, our analysis on Nginx showed that we could automatically reduce a high 52.60% of capabilities in terms of lexical encapsulation. It showed that there is a significant portion of encapsulated-objects requiring as high as 1000 external objects, that we have automatically identified and could set methods to sanitize and sandbox them.

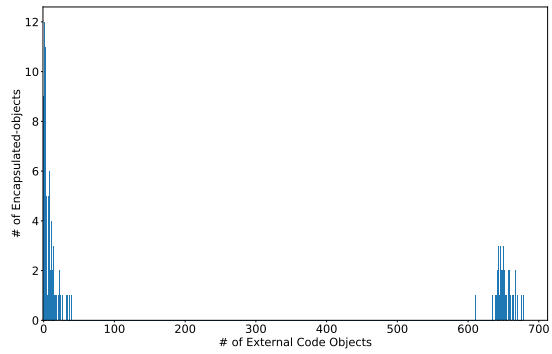
2) JSON Library Evaluation: Specifically, all JSON libraries parse and convert JSON to an in-memory data struc-



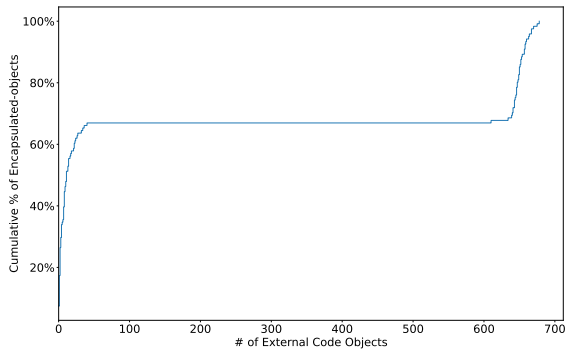
(a) # of External Data Objects for each Encapsulated-Object



(b) Cumulative Percentage Distribution of External Data Objects



(c) # of External Code Objects for each Encapsulated-Object



(d) Cumulative Percentage Distribution of External Code Objects

Fig. 8: The Statistics of External Calls and Objects Accessible in each Encapsulated-Object in Nginx

ture. The similarity in the libraries allows us to compare and choose JSON libraries in terms of privilege set. We have analyzed json-c, facil.io, jansson, and cJSON as the JSON libraries written in C. Among them, cJSON is a mono file project that only consists of a source file (`cJSON.c`) and a header file (`cJSON.h`). Thus, it is unsuitable to be analyzed by the lexical encapsulation model.

Table VIII shows the PSFR of the four JSON library. In terms of the view of object-encapsulation, the single-file library cJSON is not a suitable target that the parser object is hard to be encapsulated and sandboxed. In terms of the metric, we observe that the json-c library accesses the least external objects, making the json-c library the most suitable in terms of parser encapsulation.

3) XML Library Evaluation: Likewise, three XML parsing libraries, libxml2, libroxml, and libexpat, are collected and compared. The results are shown in Table IX. All three libraries are similar, that there is little external or inaccessible objects. Almost every object belongs to the parser encapsulation. This indicates that the three xml libraries are composed of parsers only, indicating that the manual encapsulation with parsers might not be the most suitable way to set the encapsulation boundaries.

As we have applied the metric on the JSON and XML parsing libraries, we have a basic concept of how much data maybe accessed by the parsers in terms of the object-encapsulation model. We are able to conclude that whether the encapsulation boundary is suitable. If so, we can choose a single library that is the best for parser security in terms of accessing less external data objects from the object-encapsulation model.

VIII. Conclusion

This paper introduced the Object-Encapsulation Model, a new program representation and static analysis framework to measure privilege-sets of encapsulated-objects. We define the encapsulated-objects PCG to capture encapsulation models from source code. We also provide threat labeling to measure concrete threats to the program. Then, based on the encapsulated-objects PCG, we setup the Encapsulation Analysis and the Threat Analysis. We analyzed a set of parsers and software systems using our analysis metrics including the Accessibility Analysis, the EAR, and the PSFR. We conclude that for many parsers, they are well encapsulated and can be easily compartmentalized by applying our Object-Encapsulation Model and with the help of automatic or manual encapsulation boundary placement. Our Dynamic Syscall En-

TABLE VIII: Comparative study of JSON libraries using PSFR.

Name	Parser Data	Parser Code	External Data	External Code	Inaccessible Data	Inaccessible Code
json-c	67 (7.70%)	16 (1.84%)	8 (0.92%)	19 (2.18%)	580 (66.67%)	180 (20.69%)
facil.io	3284 (73.63%)	1078 (24.17%)	45 (1.01%)	0 (0.00%)	53 (1.19%)	0 (0.00%)
jansson	138 (13.5%)	32 (3.14%)	14 (1.37%)	26 (2.55%)	650 (63.79%)	159 (15.60%)
cJSON	325 (74.37%)	112 (25.63%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)

* The sum of the six percentages equals to 1.

TABLE IX: Comparative study of XML libraries using PSFR.

Name	Parser Data	Parser Code	External Data	External Code	Inaccessible Data	Inaccessible Code
libxml2	13155 (81.82%)	2915 (18.13%)	7 (0.04%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
libroxml	739 (81.93%)	160 (17.74%)	3 (0.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
libexpat	1785 (82.79%)	357 (16.56%)	14 (0.65%)	0 (0.00%)	0 (0.00%)	0 (0.00%)

* The sum of the six percentages equals to 1.

capsulation shows the ability to select and evaluate authority context. The Threat Analysis and the Access Distance Metric allows a reduction in exposure of sensitive objects automatically given appropriate permission control.

Acknowledgments

We would like to acknowledge the anonymous reviewers and LangSec for providing instructive feedback that improved this paper. This research was supported in part by National Science Foundation Awards #2146537 and #2008867. This research was supported in part by the DARPA SafeDocs program as a subcontractor to Galois under HR0011-19-C-0073.

References

[1] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-Privilege Compartments". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, Apr. 16, 2008, pp. 309–322.

[2] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. "Shreds: Fine-Grained Execution Units with Private Memory". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 2016, pp. 56–71.

[3] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. "The program dependence graph and its use in optimization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (July 1987), pp. 319–349.

[4] Andrada Fiscutean. *How patch Tuesday keeps the beat after 20 years*. Mar. 2023. URL: https://www.darkreading.com/edge-articles/how-patch-tuesday-keeps-the-beat-after-20-years.

[5] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. "Enclosure: Language-Based Restriction of Untrusted Libraries". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 255–267. URL: https://doi.org/10.1145/3445814.3446728.

[6] 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. New York, NY, USA: ACM, 2015, pp. 1016–1031. URL: http://doi.acm.org/10.1145/2810103.2813611.

[7] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. "The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization". In: *CoRR* abs/2108.03705 (2021). arXiv: 2108.03705. URL: https://arxiv.org/abs/2108.03705.

[8] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. New York, NY, USA: ACM, 2015, pp. 291–302.

[9] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. "Program-Mandering: Quantitative Privilege Separation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, Nov. 2019, pp. 1023–1040.

[10] LLVM. *LLVM Testing Infrastructure Guide — LLVM 3.6 Documentation*. 2014. URL: http://llvm.org/docs/TestingGuide.html.

[11] Mark Samuel Miller. "Robust Composition: Towards a Unified Approach to Access Control and Concurrency

Control". AAI3245526. PhD thesis. Baltimore, MD, USA: Johns Hopkins University, 2006.

[12] Scott Moore, Christos Dimoulas, Robert Bruce Findler, Matthew Flatt, and Stephen Chong. "Extensible Access Control with Authorization Contracts". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 214–233. URL: http://doi.acm.org/10.1145/2983990.2984021.

[13] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, André DeHon, Jonathan M Smith, and Nathan Dautenhahn. "μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts". en. In: *In 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21)*. ACM, 2021, p. 16.

[14] Yulei Sui and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM". In: *Proceedings of the 25th international conference on compiler construction*. 2016, pp. 265–266.