Fast, Flow-Sensitive C Program Partitioning via Iterative Value-Flow Refinement

Maxwell Levatich
Columbia Unversity
New York, USA
ml4553@columbia.edu

Stephen A. Edwards
Columbia Unversity
New York, USA
sedwards@cs.columbia.edu

ABSTRACT

Software vulnerabilities that expose confidential data remain a persistent threat. Even vulnerabilities in unprivileged code can expose data in privileged code when they share memory. *Program partitioning* safeguards against this by dividing a monolithic program into component programs according to a *security policy* for sharing confidential data, following the Principle of Least Privilege: each component of a system should operate with minimum privileges.

Automatic C program partitioners determine a partition by tracking dataflows of confidential values; when pointers are involved, this means a pointer analysis, so precision and performance tradeoffs are inevitable. Existing work uses Andersen's flow-insensitive analysis, which is efficient but rejects secure programs—on the other hand, flow-sensitive analysis has not traditionally scaled.

In this paper, we present *CSP* (C Sparse Partitioner), a C partitioner that supports robust security policies for mutual distrust and employs sparse demand-driven pointer analysis to achieve flow-sensitive precision at a fraction of the cost of whole-program flow-sensitive analysis. Our key innovation is an *iterative value-flow refinement* where a flow-insensitive constraint system is locally refined for flow-sensitivity as partitioning conflicts are found. We evaluate CSP on large real-world C programs with security requirements such as OpenSSH and Sendmail, and find that our technique outperforms whole-program flow-sensitive pointer analysis by up to an order of magnitude without sacrificing precision.

CCS CONCEPTS

• **Security and privacy → Information flow control**; • **Software and its engineering → Automated static analysis**; *Software evolution.*

KEYWORDS

Program partitioning, privilege separation, information-flow control, constraint satisfaction, demand-driven pointer analysis, sparse analysis, flow-sensitivity

ACM Reference Format:
Maxwell Levatich and Stephen A. Edwards. 2026. Fast, Flow-Sensitive C Program Partitioning via Iterative Value-Flow Refinement. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787840

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787840

APPLICATION UNDER RULE 41 FOR A WARRANT TO SEARCH AND SEIZE

I, [redacted], being first duly sworn, hereby depose and state as follows:

INTRODUCTION AND AGENT BACKGROUND

1. The government is conducting a criminal investigation concerning the improper

Figure 1: A public redacted court document; only credentialed users should have access to the unredacted version.

1 INTRODUCTION

Software security, always paramount, has grown in importance as software enters more areas of our lives. Yet despite decades of research, software vulnerabilities that expose confidential data remain a constant threat. Vulnerabilities in unprivileged code can often expose privileged code and data to adversaries because both are in the same program or share memory, such as in documented exploits of GNU Beep and Microsoft Office [2, 3]. The Principle of Least Privilege (PoLP) [46] offers a guideline for securing software in the face of unknown vulnerabilities—if each program or module in a software system has only the minimum privileges it requires, the compromise of an unprivileged module will not lead to the whole system being compromised.

1.1 The automatic partitioning problem

Program *partitioning*, also called *compartmentalization* or *modularization*, is a realization of the PoLP in which a monolithic program is divided into isolated *component* programs that have disjoint address spaces or run on physically separate hardware. Together, the partitioned components implement the original program, but interact only through controlled channels, such as `ecall`/`ocall` functions for Trusted Execution Enclaves (TEEs), or remote procedure calls.

The intent of partitioning is to prevent an adversary who gains control of an unprivileged component from accessing confidential data in a privileged component, but a manual partitioning effort makes no guarantee of this. For example, programs partitioned for TEEs frequently suffer from *enclave leakage* [10], where the privileged component accidentally releases confidential data out of the enclave, due to an algorithmic bug or partitioning mistake.

Automatic program partitioners have been proposed for various application domains [4, 12, 18, 25, 47, 55, 60] to cut down on tedious programmer effort and offer a machine-checked assurance that the desired security properties actually hold. An automatic partitioner takes as input a monolithic program and a *security policy* that identifies components' confidential data and cross-component sharing rules (e.g., encrypted data is shareable but cleartext data is not). Using either dataflow or type-based analysis, the partitioner tracks

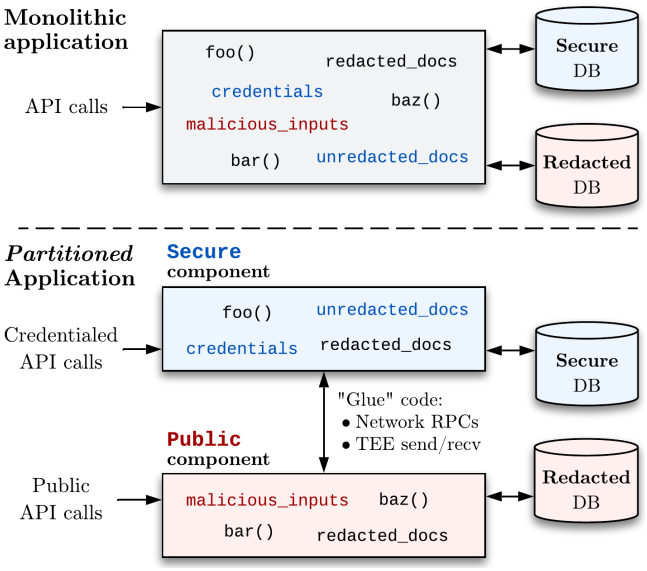

Figure 2: A desirable secure partition of the court document example. The database of unredacted documents is accessed by the `Secure` component; a bug in `Public` will not expose it.

the flow of confidential values to all of their potential uses throughout the program to determine whether a policy-compliant division exists. The output *partition* is most typically an assignment of each function and global variable to a component, since function calls are a natural interface between different components that minimizes the refactoring required for cross-component communciation.

In this paper, we present *CSP* (C Sparse Partitioner), an LLVM-based automatic partitioner that improves upon the state of the art in C program partitioning, a natural target for partitioners [8, 18, 23, 27, 29–32, 36, 42, 57] due to C's ubiquity and relative lack of security. CSP enables new partitions by using a novel *iterative value-flow refinement* to more precisely track the paths of confidential values through a program while maintaining the scalability of exisiting partitioners' imprecise analysis, and by supporting *robust* security policies for mutual distrust and controlled declassification. We first motivate the development of CSP with a running example (Section 1.2) before outlining our contribution in detail (Section 1.3).

1.2 Motivating example

Consider a legacy C application that manages criminal court records. By law, the records must be available to the public, but only in redacted form, e.g., Fig. 1. The original, unredacted documents are maintained in a separate, secure database that should only be accessible to credentialed users. The top of Fig. 2 illustrates the existing program. The bottom of Fig. 2 shows a desirable partition: the database access logic is split into memory-isolated secure and public components. In the partitioned architecture, an adversary that exploits a vulnerability to gain control of the public component cannot leak unredacted documents. Since the application is complex, a secure partition is best found by an automatic partitioner.

Fig. 3 shows a code fragment from the application that publishes a court document to the secure database, creates a redacted version,

```
void pubWrite(char s[DOC_LEN]) { _pub_insert(s); }
typedef struct DBInterface {
    char* (*read)(int);
    void (*write)(char[DOC_LEN]);
} DBI;
void setSecureEndpoint(DBI *db) {
    db->read = secRead;
    db->write = secWrite;
}
void setPublicEndpoint(DBI *db) {
    db->read = pubRead;
    db->write = pubWrite;
}
void publish(char crt_doc[DOC_LEN], DBI *db) {
    // Publish cleartext document to secure DB
    setSecureEndpoint(db);
    db->write(crt_doc);
    // Redact the document
    char redact_buf[DOC_LEN];
    redact(crt_doc, redact_buf);
    // Publish redacted document to public DB
    setPublicEndpoint(db);
    db->write(redact_buf); }
```

Figure 3: A C code fragment of the database app (Fig. 2) that publishes a court document to the secure database, redacts it, and writes the redacted version to the public database.

```
components:   [SECURE, PUBLIC]  # Split into two components
confidential-values:
  SECURE:     [crt_doc]         # Secure unredacted docs
pinned-functions:
  PUBLIC:     [_pub_insert]     # Accessing public database
declassifiers:
  redact_buf: [PUBLIC]          # Redaction makes data public
```

Figure 4: A YAML security policy for the code in Fig. 3: the confidential value `crt_doc` is owned by the `SECURE` component and must not leak to the public database unless redacted.

and writes the redacted version to the publicly accessible database. The authors of Fig. 3 have avoided writing redundant code by creating a common interface to the secure and redacted databases through the `DBI` struct of function pointers `read` and `write`.

1.2.1 Specifying security policies. We use a YAML *security policy* to represent an application's cross-component data-sharing requirements. Fig. 4 gives the policy for the code in Fig. 3: there are two components, `SECURE` and `PUBLIC`; data in the cleartext document variable `crt_doc` is owned by `SECURE` and must not be released to `PUBLIC`; the `_pub_insert` function, which communicates with the redacted database, resides in `PUBLIC`; and the `redact_buf` variable holds redacted data that is therefore *declassified* to `PUBLIC`.

In addition to expressing isolation of confidential values from unprivileged components, our policies express *ownership*—even after `redact_buf` reaches `PUBLIC`, it cannot be further declassified to some third component, because the original value is owned by `SECURE`—and express *mutual distrust*—a policy can also grant

PUBLIC ownership of values that should not be shared with SECURE (e.g., unsanitized user inputs). Most existing C partitioners use a simple "safebox" security model consisting of one privileged and one unprivileged component with permissive declassification [25]. Our more robust policies are instead based on the Decentralized Label Model [39], a foundational model in information-flow control long used by Java-based partitioners [60]. Mutual distrust and controlled declassification enable CSP to be used for complex multi-party computations and protocols. The policy in our running example is simple for the sake of illustration; in Section 3, we evaluate CSP on policies that cannot be expressed by most C partitioners.

1.2.2 Scaling pointer analysis. Fig. 3 is secure and partitionable; SECURE values in `crt_doc` do flow to `_pub_insert()`, but only after passing through `redact_buf`, which is blessed by the policy to declassify `crt_doc` to PUBLIC.

Existing C partitioners *cannot statically determine this example is secure, however, because they are not flow-sensitive.* Automatic partitioners rely on dataflow analysis to discover how confidential data flows through a program; this means performing pointer analysis, which is undecidable [43], so precision and performance tradeoffs are inevitable. Existing C partitioners largely rely on a fast, conservative *flow-insensitive* analysis, typically Andersen's field-sensitive analysis [5], that computes a single alias set for each variable in the program. For example, a flow-insensitive analysis of Fig. 3 will find the `db->write()` function pointer can refer to both `secWrite()` (due to line 10) and `pubWrite()` (due to line 14). This implies an insecure (unredacted) flow of `crt_doc` into `_pub_insert()` (Fig. 5). But this flow is *spurious*; it does not occur in the actual program.

Flow-sensitive pointer analysis is precise enough to resolve this discrepancy. It tracks alias sets for each variable *at each program point.* On line 19, it determines that `db->write()` only aliases `secWrite()`, because `setPublicEndpoint()` is not called until line 26. Researchers have found flow-sensitivity can benefit static security analyses [16], but a traditional IFDS-based [44] flow-sensitive pointer analysis has not been considered for partitioning because it does not scale to large real-world applications [19, 22, 30].

But flow-sensitive pointer analysis has seen substantial performance gains in the past decade. In *sparse* or *staged* pointer analysis [20, 52], Andersen's analysis is first run to compute a flow-insensitive *value-flow graph* (VFG) (alternatively, *dependence graph* [15]) such as Fig. 5 that connects top-level and address-taken variable definitions directly to all of their (potential) uses. A flow-sensitive pass can then propagate points-to sets "sparsely" over the VFG, targeting both interprocedural memory dependencies and refinement of indirect call targets, which is sidestepped in partially flow-sensitive approaches [34, 42, 61].

Whole-program sparse pointer analysis remains costly, however, and is significant wasted computation for our use-case—in an application of potentially tens of thousands of lines, the only information we are missing to determine a secure partition is that the function pointer on line 19 of Fig. 3 doesn't point to `pubWrite()`.

1.3 Our contribution: Iterative value-flow refinement

Recent work on *demand-driven* sparse pointer analysis [53] enables clients to query individual pointers for flow-sensitive points-to information resolved via a backwards analysis on the pre-computed VFG. Demand-driven flow-sensitive queries appear ideal for precise partitioning, but *the necessary pointers to query are not known prior to partition analysis.* We therefore introduce our key algorithmic contribution, *iterative value-flow refinement* (Fig. 6):

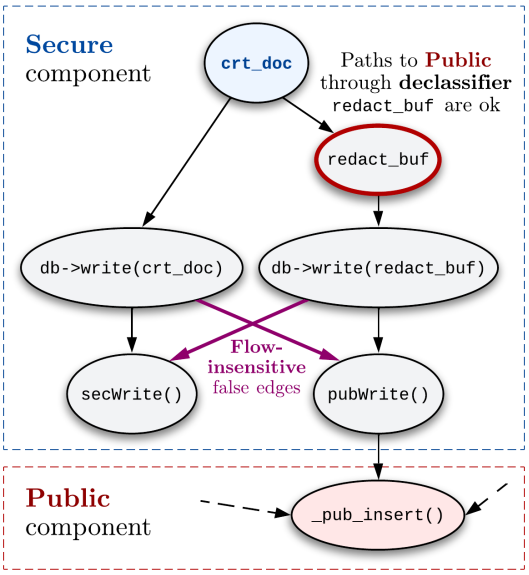

Figure 5: A slice of the value-flow graph for Fig. 3. It contains spurious edges that suggest `crt_doc` flows to the public database, so a flow-sensitive refinement is needed.

(1) As in sparse pointer analysis, we first cheaply compute a flow-insensitive VFG with Andersen's pointer analysis.
(2) We formulate partitioning as a *constraint satisfaction problem* over the VFG, where the constraints capture both the desired security requirements, e.g., confidential data should not flow into an unprivileged component, and program *validity* requirements, e.g., pointers should not be passed between memory-isolated components.
(3) Solving yields an *unsat core*; a set of unsatisfiable constraints corresponding to an insecure flow, e.g., the path without a declassifier from `crt_doc` to `_pub_insert()`.
(4) Pointers used along this path are queried for their flow-sensitive points-to sets.
(5) The new points-to sets are used to refine the VFG, possibly eliminating the insecure path.

Steps (2) through (5) repeat until solving yields a secure partition, or all pointers along the insecure path have already been queried, indicating the code cannot be partitioned. Our approach is similar in spirit to Counterexample-Guided Abstraction Refinement (CEGAR) [13], a technique in symbolic model checking where an abstract program model is refined based on spurious conflicts that appear in the overly abstract model but not the actual program.

Applied to Fig. 3, we first compute the flow-insensitive VFG, which gives Fig. 5. Security constraints arise from the policy in Fig. 4; validity constraints arise from the code, e.g., `setSecureEndpoint()` and `setPublicEndpoint()` are called by `publish()` with a pointer

argument—all three must be assigned to the same component to avoid the possibility of dereferencing a pointer allocated in a different component. Solving the constraints on this VFG fails because of the (false) insecure path from `crt_doc` to `_pub_insert()`. Because `db->write(crt_doc)` is on the insecure path, we query the function pointer for its flow-sensitive points-to set, which no longer includes `pubWrite()`. The corresponding edge is removed in the refined VFG, and re-solving yields the secure partition in Fig. 5. The main benefit of our approach is that we only invoke expensive flow-sensitive analysis on the small number of pointers along the insecure path. This yields a partitioning speedup of up to an order of magnitude, as witnessed by our evaluation (Section 3).

CSP implements partitioning with iterative value-flow refinement on top of SVF [52], an open-sourced and well-studied pointer analysis toolchain that supports VFGs and sparse demand-driven queries. We evaluate CSP's performance on a suite of large open-source C programs such as OpenSSH and Sendmail.

In summary, this paper makes the following contributions:

(1) We introduce an *iterative value-flow refinement* algorithm for scalable, flow-sensitive tracking of dataflows through indirect calls and interprocedural pointer dependencies when demand-driven points-to queries are not known upfront.
(2) We present an LLVM-based *C program partitioner*, CSP, that implements this technique to efficiently find flow-sensitive partitions, while supporting expressive security policies with *ownership* and *mutual distrust*.
(3) We conduct an *evaluation on large real-world C programs* with security requirements, demonstrating that CSP can partition programs over 100 KLoC via iterative value-flow refinement up to 10x faster than via whole-program sparse flow-sensitive analysis.

2 TECHNICAL APPROACH

We describe our approach in detail below, starting with a definition and justification for what it means for a partition to be *secure* with respect to a security policy (Section 2.1). We then explain our partitioning algorithm (Fig. 6), which performs a *constraint satisfaction* step that searches for a valid secure partition (Section 2.2) and, if it fails, uses the results to apply *flow-sensitive refinements* to the underlying value-flow graph (Section 2.3) before re-solving.

2.1 Defining a Secure Partition

From a program, our ultimate goal is a *partition*—an assignment of each function and global variable to a component—that is *secure*; each component may never access any information about a confidential value owned by a different component unless it is explicitly declassified, such as how `redact_buf` in Fig. 3 is defined in Fig. 4.

We begin by formally defining a C program partition and its security. We consider a C program to be a set of global definitions produced by compiling it to LLVM IR, a well-specified and widely-used intermediate representation suited to static analysis.

Definition 2.1 (Partition). Let $P = \{g_1, g_2, \ldots, g_n\}$ be an LLVM IR program, and $C = \{c_1, c_2, \ldots, c_m\}$ be a finite set of *components*. A *partition* of P is a total function $p : P \to C$ mapping each definition to a component. A partition leads to a set of *component programs*; for a component $c \in C$, the partition's component program is the set of definitions mapped to that component $P_c := \{g \in P \mid p(g) = c\}$.

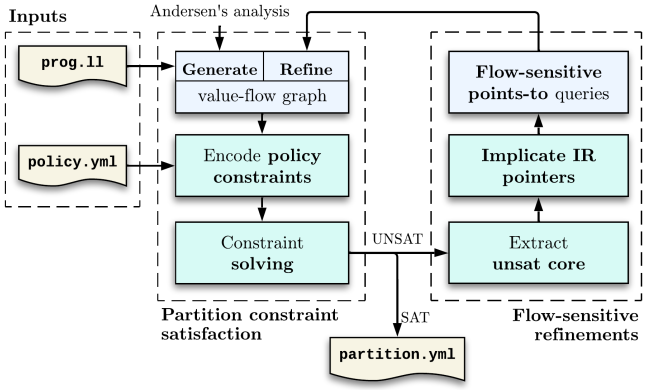

Figure 6: A high-level view of our value-flow refinement partitioning technique. Blue nodes are in the SVF toolchain.

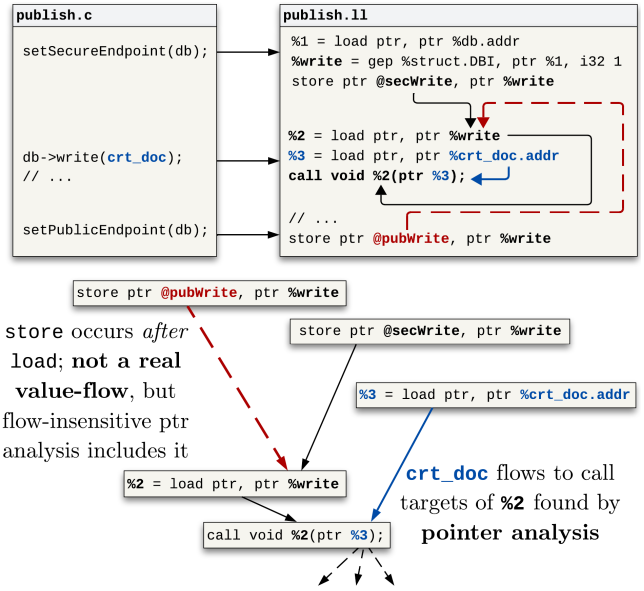

Figure 7: Value-flows in a snippet of Fig. 3 compiled to LLVM IR. Discovering all value-flows requires a pointer analysis, but flow-insensitive analysis will over-approximate them.

2.1.1 The value-flow graph. To reason about whether component programs may access confidential values, we use a *value-flow graph* (VFG) [15], a program representation for dataflow analyses that characterizes the propagation of values among program statements.

In LLVM IR, values reside in virtual registers or in memory locations addressed by pointers. LLVM instructions `load`, `store`, `gep` (address-of calculations), `call`, and `ret`, as well as `binops` and `phi` instructions in SSA form, move information about a value between storage locations. Sequences of these on the same value are *def-use* chains, or *value-flows*. Fig. 7 illustrates some of the LLVM IR and value-flows for the `publish()` function from Fig. 3.

The VFG is a directed graph of a program's value flows; a path between two nodes in the VFG is a possible value flow in the original program. The VFG is an over-approximation—capturing indirect value flows through `loads` and `stores` requires a pointer analysis, so precision and performance tradeoffs must be made. Existing C partitioners [27, 30, 42], if they capture indirect value flows at all, build VFGs using a fast, flow-insensitive pointer analysis such as Andersen's analysis [5], resulting in spurious value flows like the one shown in red in Fig. 7. We take a *sparse*, *demand-driven* approach, initially building a flow-insensitive VFG and later refining it for flow-sensitivity (Section 2.3). We construct VFGs using the state-of-the-art open-source pointer analysis library SVF [52]. SVF uses input pointer analysis results to convert a program to *Memory SSA*, in which both top-level and address-taken variables are in SSA form, and value-flows are readily available by simply connecting a definition directly to all of its uses. Thus the VFG is straightforward to build from Memory SSA.

Information-flow properties are easy to express and efficient to check over the VFG: given a confidential variable v and a component program P_c, we use the absence of a path from v's allocation node in the VFG to any function or global in P_c to indicate that P_c cannot access any information about values flowing from v.

2.1.2 Robust security policies. That a component program may not access information about another component's confidential values is a *non-interference* property [51]. But complete non-interference is rarely useful in real systems. For example, while a user's password is certainly confidential, even a failed login attempt by a different user is a flow of information about the correct password (namely, that it is not the one tried). As such, pratical policies for information-flow control allow *declassification*, where a confidential value, or partial information about it, is released, such as the redaction in Fig. 3.

Yet careless declassification undermines the strong security guarantee of non-interference. The robustness of a security policy therefore depends on its *dimensions* of declassification [45]—when, where, and how declassification is allowed. State-of-the-art C partitioners are maximally permissive—values are declassified to all components and are functionally ignored during analysis.

Our policies follow the Decentralized Label Model [38] common in Java partitioners [60], which uses *owner-based* declassification. In Fig. 4, `redact_buf` is declassified specifically to `PUBLIC`; within the `PUBLIC` component, it cannot be further declassified because the original value is owned by `SECURE`. This property is crucial for computations of many parties with mutual distrust, enabling CSP to secure complex distributed protocols from a serial prototype.

Nevertheless, Fig. 4 shows that our policies are simple to express over identifiers in the source code as only a small fraction of names need to be mentioned. We project a source-level security policy onto the VFG by mapping unambiguous identifiers to the VFG nodes representing their initial definitions; when multiple scopes define the same identifier, the correct node must be disambiguated.

Definition 2.2 (Security policy)**.** Let P be a program, and (N, E) be a value-flow graph for P; N is a set of nodes and $E \subseteq N \times N$ is a set of edges expressed as pairs (n_1, n_2).

A *security policy* for P is a tuple $(C, S, D, f_{\text{pin}})$, where

- C is the set of components (*components* in Fig. 4).
- $S : N \to 2^C$ defines which components declare ownership of confidential values at a node. Specifically, $S(n)$ is the set of components that *own* values at n (*confidential-values* in Fig. 4: values flowing from `crt_doc` are owned by `SECURE`). Owners may read these values and declassify them to new readers. When $S(n) = \emptyset$, n is not a confidential value source.
- $D : N \to 2^C$ defines which additional components are being given access to confidential values at a node. Specifically, $D(n)$ is the set of components to which node n declassifies values (*declassifiers* in Fig. 4: values stored to `redact_buf` are declassified to `PUBLIC`). When $D(n) = \emptyset$, the typical case, n does not further declassify any values.
- $f_{\text{pin}} : P \to C$ is a partial mapping that *pins* functions to specific components. Pinned functions are a partitioning utility allowing a developer to assert that a computation happens in a particular component (*pinned-functions* in Fig. 4: `_pub_insert` is pinned to `PUBLIC` to enforce that all accesses to the redacted database must occur in `PUBLIC`).

2.1.3 Partition security. The central goal of our algorithm is to find a *node-level partition*,

$$t : N \to C,$$

that assigns each node in the VFG to a component. In particular, we want a *secure* and *valid* partition that meets constraints on the program's structure and the security of the data flowing it. We write t because the assignment to components is a sort of taint.

Intuitively, a VFG is secure if each confidential value either never leaves its owner component, or if its owner declassifies it to all other components it flows to. Formally, this amounts to path constraints:

Definition 2.3 (Partition security)**.** Let (N, E) be a value-flow graph; N is a set of nodes, and $E \subseteq N \times N$ is a set of pairs (n_1, n_2).

A *path* in (N, E) from n_1 to n_k is a sequence of $k \in \mathbb{N}$ nodes $(n_1, n_2, \ldots, n_k)$ where $n_i \in N$, and for each $i \in [1, k-1]$, $(n_i, n_{i+1}) \in E$. Note that the singleton sequence (n) is a path.

A node-level partition t of a value-flow graph (N, E) with policy $(C, S, D, f_{\text{pin}})$ is *secure* if and only if, for every path from a confidential value source $n_s \in N$ with $S(n_s) \neq \emptyset$ to some node $n \in N$ with component $t(n)$, at least one of the following holds:

(1) $t(n) \in S(n_s)$ (n owns the confidential value); or
(2) on the path there is a *declassifier* $n_d \in N$ with
   (a) $t(n_d) \in S(n_s)$ (n_d owns the confidential value) and
   (b) $t(n) \in D(n_d)$ (n_d declassifies to n's component).

2.2 Partitioning via Constraint Satisfaction

We now detail our algorithm for finding a secure partition.

In a safebox security model with a hierarchy of privilege levels (often just two: sensitive and not) and permissive declassification, partitioning is a *taint analysis* over the VFG—confidential values are taint sources, declassifiers are taint sanitizers, and a taint propagation from each confidential value is sufficient to find a secure node-level partition. CSP, however, cannot use a simple forward traversal due to our richer security policies:

- Confidential values may have multiple owners to propagate, but each node is ultimately assigned to one component.

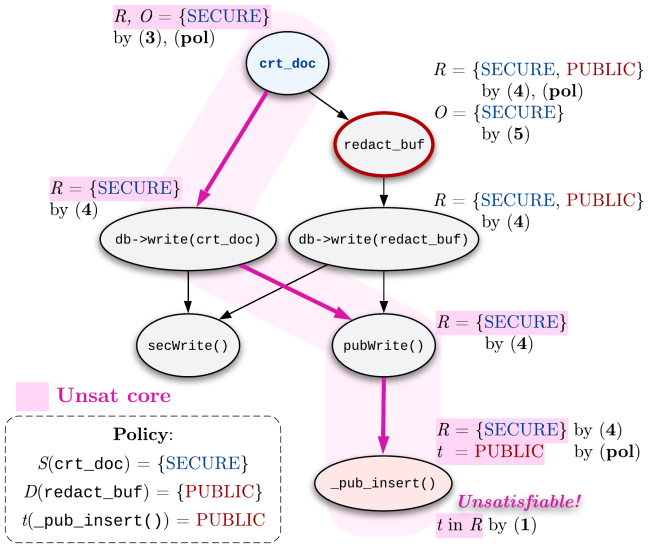

Figure 8: Applying the security constraints in Definition 2.4 to our motivating example. Solving yields an *unsat core*: a minimal set of conflicting constraints.

- There is no privilege hierarchy; the confluence of two confidential value-flows with disjoint readership is a conflict, not just a higher privilege assignment superseding the lower.
- Whether a declassifier node is allowed to declassify incoming values depends on its assigned component.

As such, we formulate partitioning as a *constraint satisfaction* problem that seeks to assign each node n to a component $t(n)$ in a way that satisfies *security* and *validity constraints* on t. We use Z3 [14] to solve the constraints; If the VFG is too conservative, Z3 will be unable to find a satisfying partition even if one exists, necessitating VFG refinement (Section 2.3).

2.2.1 Security constraints. We first address security constraints. Definition 2.3 is an all-paths property, but we reduce it to edge constraints by defining *reader* and *owner* sets for each node; edge constraints make solving tractable for Z3. As defined below, owners for confidential value nodes are given by the policy (3) and propagate unchanged throughout the program (5) to indicate which nodes may declassify information (2); readers constraint what data each node may see (1) and also propagate from confidential value sources (3), but may be expanded at declassifier nodes (4).

Definition 2.4. *Readers* R and *owners* O for a partition p of a program P with value-flow graph (N, E), policy $(C, S, D, f_{\text{pin}})$, and node-level partition t are functions $R, O : N \to 2^C$ where, for all nodes $n \in N$, n must be readable by its assigned component $t(n)$,

$$t(n) \in R(n); \tag{1}$$

a declassifying node $d(n) \neq \emptyset$ must own the value it declassifies,

$$D(n) \neq \emptyset \implies t(n) \in O(n); \tag{2}$$

the owners of a confidential value source own and read their data

$$S(n) \neq \emptyset \implies O(n) \subseteq S(n) \wedge R(n) \subseteq S(n); \tag{3}$$

a reader of a node n with a predecessor n_i is either declassified-to by n or it must be a reader of all its predecessors,

$$\exists n_i.(n_i, n) \in E \implies R(n) = D(n) \cup \bigcap_{(n_i,n)\in E} R(n_i); \text{ and} \tag{4}$$

for nodes with predecessors, its owners must be common to all its predecessors,

$$\exists n_i.(n_i, n) \in E \implies O(n) = \bigcap_{(n_i,n)\in E} O(n_i). \tag{5}$$

Fig. 8 shows how constraints on R and O are applied to the VFG in Fig. 5. We now prove that finding an appropriate R and O suffices to show security (Definition 2.3).

Lemma 2.1. Let t be a node-level partition of a value-flow graph (N, E) with policy $(C, S, D, f_{\text{pin}})$ and reader and owner functions R and O that follow Definition 2.4. Consider a path from a confidential value source node $n_s \in N$, $S(n_s) \neq \emptyset$ to a node $n \in N$. If there is a component $c \in R(n)$ and $c \notin S(n_s)$, then there exists a declassifier node n_d on the path with $c \in D(n_d)$.

Proof. Let $(n_1, n_2, \ldots, n_k)$ be the path from n_s to n, i.e., $n_1 = n_s$ and $n_k = n$. Because $S(n_s) \neq \emptyset$, it follows from (3) that $R(n_1) \subseteq S(n_s)$. For every node n_i on the path, it follows from (4) that $R(n_i) = D(n_i) \cup \bigcap_{(n_j,n_i)\in E} R(n_j)$. Then for the edge (n_{i-1}, n_i) on the path, $R(n_i) \subseteq D(n_i) \cup R(n_{i-1})$. By induction, it follows that $R(n) \subseteq \bigcup_{n_i} D(n_i) \cup R(n_1)$. Let c be the component with $c \in R(n)$ and $c \notin S(n_s)$. Then $c \in \bigcup_{n_i} D(n_i) \cup R(n_1)$.

However, $c \notin S(n_s)$ and $R(n_1) \subseteq O(n_s)$, so $c \notin R(n_1)$. Therefore, c must be in $\bigcup_{n_i} D(n_i)$, so there must be some n_d on the path with $c \in D(n_d)$. □

Theorem 2.2. Let t be a node-level partition of a value-flow graph (N, E) with policy $(C, S, D, f_{\text{pin}})$. If R and O are reader and owner functions for t, then t is secure (Definition 2.3).

Proof. Consider a path in (N, E) from a confidential value source node $n_s \in N$, $S(n_s) \neq \emptyset$ to some node n.

If $t(n) \in S(n_s)$, then the path satisfies case (1) of Definition 2.3.

Now consider $t(n) \notin S(n_s)$. From (1), we have $t(n) \in R(n)$, therefore $R(n) \not\subseteq S(n_s)$. From Lemma 2.1 when c is $t(n)$, there exists a node n_d on the path with $t(n) \in D(n_d)$.

Since $D(n_d) \neq \emptyset$, it follows from (2) that $t(n_d) \in O(n_d)$. By induction on (5), $O(n_d) \subseteq O(n_s)$, and we have $O(n_s) \subseteq S(n_s)$ from (3), so $t(n_d) \in S(n_s)$. So the path satisfies case (2) of Definition 2.3.

Because any path in (N, E) starting from a confidential value source satisfies Definition 2.3, t is secure. □

2.2.2 Validity constraints. In addition to guaranteeing security, partitioners must not introduce bugs; the partitioned program must behaves the same as the original, modulo the "glue" code implementation. For example, C programs frequently pass pointers between functions, but a partition that places a callee with pointer arguments in a different, memory-isolated component than the caller causes undefined behavior if the callee dereferences its inputs. *Validity* constraints prevent these cases.

Let t be a node-level partition of a value-flow graph (N, E) with policy $(C, S, D, f_{\text{pin}})$. Our validity constraints are, for all nodes $n \in$

VFG node type	Sample instruction	Query points-to
LoadVFGNode	`%0 = load ptr, ptr %write, align 8`	`%write`
StoreVFGNode	`store ptr @secWrite, ptr %write, align 8`	`@secWrite`, `%write`
GepVFGNode	`%write = getelementptr inbounds %struct.DBInterface, ptr %1, i32 0, i32 1`	`%1`
ActualParmVFGNode†	`call void %0(i32 noundef %1)`	`%0`, `%1`
ActualRetVFGNode	`ret i32 %0`	`%0`

†Note that for indirect calls such as this one, the called pointer is also queried

Table 1: The implicated pointers extracted from VFG node types. Unlisted node types are passed over.

N, a node must be assigned to the same component as its containing definition if it has one,

$$\text{inFun}(n) \neq \text{null} \implies t(n) = t(\text{inFun}(n)), \tag{6}$$

where $\text{inFun}(n)$ indicates the function, if any, that the node n belongs to; a pinned function must be in its assigned component,

$$\text{typ}(n) = \text{fun} \wedge f_{\text{pin}}(n) \text{ is defined} \implies t(n) = f_{\text{pin}}(n), \tag{7}$$

where $\text{typ}(n)$ indicates whether the node n is a function definition, a load node, etc.; nodes that `load`, `store`, or `gep` data from global variables or functions must not do so between components,

$$\text{typ}(n) \in \{\text{load}, \text{store}, \text{gep}\} \implies t(n) = t(\text{src}(n)) = t(\text{dst}(n)), \tag{8}$$

where $\text{src}(n)$ is the node corresponding to the source operand and $\text{dst}(n)$ is the destination operand; a function call must not go between components if any parameter is (or contains) a pointer,

$$\text{typ}(n) = \text{parm} \wedge \text{isP}(n) \implies \forall d \in \text{tgts}(n) \,.\, t(\text{src}(n)) = t(d), \tag{9}$$

where $\text{tgts}(n)$ are call targets and $\text{isP}(n)$ is true if the value at node n is or contains a pointer; a function return must not go between components if the return value is (or contains) a pointer,

$$\text{typ}(n) = \text{ret} \wedge \text{isP}(n) \implies \forall d \in \text{tgts}(n) \,.\, t(\text{src}(n)) = t(d); \tag{10}$$

and indirect calls must not go between components,

$$\text{typ}(n) = \text{call} \wedge \text{isP}(n) \implies \forall d \in \text{tgts}(n) \,.\, t(n) = t(d). \tag{11}$$

A formal definition and proof of partition validity requires a bisimulation of LLVM operational semantics, including semantics for RPC communication, which is outside the scope of this paper. On the correctness of our validity constraints, we comment informally:

- (8) should prevent an expression using an identifier from a different component (a compile-time error), and (9), (10), and (11) should prevent a pointer leaving the component in which it was allocated (undefined behavior). We consider these cases as the main threats to partition validity.
- Real C programs pass pointers between functions so frequently that (9) and (10) may severely limit solving. We include a flag in CSP that turns off (9) and (10); we assume in this case a developer will manually marshal and unmarshal cross-component pointers in the RPC implementation.

Like other novel contributions in partition analysis [30, 33], we do not implement the orthogonal task of automatic code division and glue code generation to create an executable partition. Code generation depends on the target architecture and cross-component communication mechanism (IPC, Intel SGX primitives, etc.)—our technique is agnostic to these choices. In the absence of a testable, executable partition, validity constraints witness that a determined partition is at least "realizable". They also demonstrate the ease of integrating correctness requirements and security requirements via an expressive constraint language, an advantage of our approach. Static validity constraints cannot, however, capture *dynamic* partitioning considerations such as the runtime overhead of cross-component communication, so supporting automatic code generation remains an important future direction (Section 3.5).

2.2.3 Constraint encoding. We use the well-vetted SMT solver Z3 [14] to solve security and validity constraints. Constraints (1)-(6) are encoded only over forward slices from confidential value nodes in the VFG, and constraints (7)-(11) on individual nodes are encoded as constraints on their containing functions, minimizing the number of nodes for Z3 and speeding solving time.

On a satisfiable result, we obtain a secure source-level partition p from the secure node-level partition by querying Z3's model for the assigned component $t(n)$ of each function and global definition node n; due to (6), this conversion is "lossless." Below, we discuss how we handle an unsatisfiable result.

2.3 Iterative Value-Flow Refinement

If no secure partition of the VFG exists, Z3 will report unsatisfiability. But spurious value-flows in the flow-insensitive VFG may be blocking a secure source-level partition, as in Fig. 5. We address this through *iterative value-flow refinement*: we increase the precision of the VFG as needed until solving yields a satisfiable partition, foregoing a whole-program flow-sensitive pointer analysis.

2.3.1 Discovering insecure VFG paths. We begin with the observation that flow-sensitivity is only needed along a subset of paths from confidential value sources; a whole-program analysis is wasted work. But how to discover which paths to refine?

On an unsatisfiable result, Z3 supports computing an *unsat core*, a minimal subset of the full constraints that still cannot be solved in isolation. In effect, the core is a distilled "reason" the overall constraint problem is unsatisfiable.

Because every one of the constraints in Section 2.2 is associated with a specific VFG node, we map the unsat core—a set of constraints—to a path or set of intersecting paths in the VFG that are insecure. Fig. 8 shows the offending path for our motivating example. If a secure partition exists for the original program, then these unsatisfiable paths must be spurious value flows that cannot occur in reality; a flow-sensitive analysis may eliminate them.

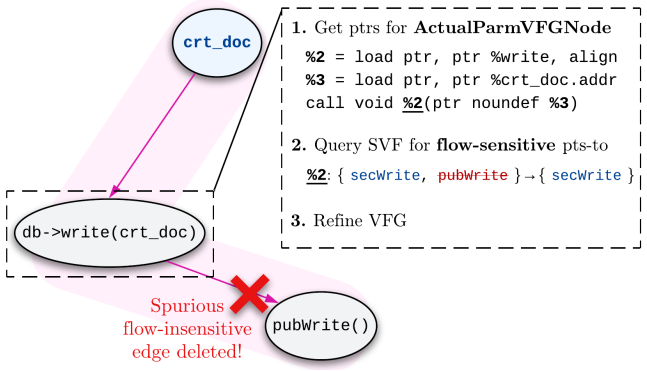

Figure 9: We extract pointers from each node in the unsat core and query SVF for flow-sensitive refinement. From the `db->write(crt_doc)` node, we extract `%2`—our query reveals it does not point to `pubWrite()`, so we remove the VFG edge.

2.3.2 Implicating and refining IR pointers. Fig. 9 demonstrates how the insecure path, once found, is locally refined for flow-sensitivity. First, we extract the pointers used at each node via a mapping from node type to associated instructions to pointer operands (Table 1).

Next, we leverage SVF's *demand-driven* strong update analysis [53] algorithm operating over the VFG to *query* each pointer used on the insecure path for its flow-sensitive points-to set. SVF resolves an individual flow-sensitive query by computing a backwards reachability relation over the VFG, starting from the definition node of the queried pointer. During backwards traversal, predecessor nodes are always reachable through direct value-flow edges, but indirect value-flows through address-taken variables are refined for flow-sensitivity on-the-fly through the detection of *strong updates*, where a `store` instruction "kills" other potential contents of a memory location. The final set of reachable memory objects forms the refined flow-sensitive points-to set. In the case of `%2` in Fig. 9, the memory object corresponding to the function definition `secWrite` is backwards-reachable, while that of `pubWrite` is not (note that these nodes are different from the `secWrite()` `pubWrite()` nodes in Figures 5, 8, and 9, which represent dataflows through call instructions). A more comprehensive description of the algorithm and reachability relation is given in Section 4.2 of [53].

2.3.3 Flow-sensitive refinement loop. By the above algorithm, each flow-sensitive query may recursively refine points-to sets for other pointers; for example, refining the points-to set of a pointer variable in the body of an indirect callee might also require refining the called function pointer. After all pointers on the insecure path are queried, we merge all refined points-to sets into the initial whole-program flow-insensitive points-to sets found by Andersen's analysis.

This merged result amounts to a *partially* flow-sensitive pointer analysis. In the same way that SVF accepts the results of Andersen's analysis to build the initial VFG, we feed it the new partially flow-sensitive points-to data to rebuild a refined VFG. As shown in Fig. 9, the spurious insecure edge is no longer present in the refined VFG.

We re-run constraint satisfaction to see if our refined VFG has a secure partition. This may expose a different conflicting path, which needs to be resolved with new flow-sensitive queries on the refined VFG, and so on, an *iterative value-flow refinement* in the style of Counterexample-Guided Abstraction Refinement (CEGAR) [13]. Algorithm 1 summarizes the complete partitioning process.

Algorithm 1 Partitioning via iterative value-flow refinement

```
function Partition(llvm, policy)
  queried ← ∅
  globals ← extractGlobals(llvm)
  pta ← svfAndersensAnalysis(llvm)
  loop
    vfg ← svfBuildVFG(llvm, pta)
    constraints, c2node ← encode(vfg, policy)
    if z3Solve(constraints) then
      return {z3ModelFor(g) : g ∈ globals}
    else
      core ← z3UnsatCore(), queries ← ∅
      for all constraint ∈ core do
        node ← c2node[constraint]
        queries ← queries ∪ extractPtrs(node)
      if queries ⊆ queried then
        return failure
      for all q ∈ queries do
        pta ← svfFlowSensitivePtaQuery(vfg, pta, q)
      queried ← queried ∪ queries
```

Iterative value-flow refinement terminates either when a secure partition is found, or when the set of implicated pointers is a subset of pointers previously queried, indicating a conflict that cannot be resolved with flow-sensitivity. If a partition is found, the final VFG will be only as flow-sensitive as needed, querying a small fraction of all pointers—thus we avoid an expensive whole-program pointer analysis while achieving the same partitioning precision.

3 EVALUATION

To evaluate the effectiveness of partitioning via iterative value-flow refinement, we used CSP to partition a mix of real-world and synthetic benchmarks between 20-300k LoC. CSP, our benchmarks, and replication instructions are available in an open-source repository: https://github.com/mlevatich/CSP.

3.1 Experimental setup

Table 2 summarizes our benchmarks and results. We selected a representative set of benchmarks used to evaluate existing C partitioners, taint analysis tools, and pointer analysis libraries; OpenSSH is used by PrivTrans [8], PtrSplit [30], and many others. Wget and Thttpd were used by PtrSplit, PM [31], ProgramCutter [57], and others. SendMail and Make were used by the authors of SUPA [53], SGX-sqlite3 by STELLA [10], and SecDesk by CAPO [27]. Most of these works give the threat model, confidential data, and partitioning intent for each benchmark (e.g., protecting private keys in OpenSSH), and our security policies are formed by aggregating theirs. An exception is SUPA, a general-purpose pointer analysis—for SendMail and Make we chose policies specifically to surface insecure flows, without any particular threat model considered.

We ran CSP in three "modes"; `ander-wpa` (**A**), which only attempts partitioning over the initial flow-insensitive VFG; `fs-wpa`

Program	KLoC	Part'd			Ptr. Analysis (s)			VFG Build (s)			Refinement	Partition Policy
		A	W	I	A	W	I	A	W	I	Steps	
OpenSSH-10.0 [58]	100	N	Y	Y	8.2	233.5	11.8	22.0	13.4	18.2	1	Protect private key, keep sshbuf creation in insecure part
Wget-1.25 [40]	61	N	Y	Y	2.7	20.5	3.7	7.2	4.7	5.9	1	Isolate downloaded file, but validate in secure part
"		N	Y	Y	2.7	20.5	5.3	7.2	4.7	6.0	4	Force numerous functions to secure part (arbitrary)
Thttpd [41]	22	Y	Y	Y	0.5	1.4	0.5	0.9	0.8	0.9	0	Isolate server authfile from client handlers
Sendmail-8.18 [59]	260	N	Y	Y	14.7	80.6	16.2	36.4	33.4	35.8	1	Isolate client name from daemon `addrcmp` (arbitrary)
SGX-sqlite3 [37]	214	Y	-	Y	255.0	–	255.7	452.2	–	452.6	0	Isolate enclave `ocall`s that don't depend on db conn
Make-4.2 [56]	40	N	N	N	3.6	53.3	6.5	11.8	9.5	10.3	4	Isolate `job_counter`, `_getopt_initialize` (impossible)
SecDesk [27]	24	N	Y	Y	5.0	103.9	7.1	19.7	13.7	17.1	1	Isolate facial recognition (see RQ3)
Gen16	344	N	Y	Y	41.3	93.0	42.9					Synthetic: same as motivating example

Table 2: Our benchmark set, results, and security policies for three different analyses; (A) `ander-wpa` partitions over the pre-computed VFG based on Andersen's flow-insensitive analysis; (W) `fs-wpa` refines the pre-computed VFG with a whole-program sparse flow-sensitive analysis, and; (I) `fs-ivfr` performs iterative value-flow refinement on the pre-computed VFG.

(**W**), which refines the VFG with a whole-program sparse flow-sensitive pointer analysis before partitioning; and `fs-ivfr` (**I**), which implements iterative value-flow refinement for flow- sensitivity. For each benchmark, we record whether a partition was found (**Part'd**) and time spent on pointer analysis and VFG construction (encoding and solving for a partition had negligible runtime). For `fs-ivfr`, we record the number of refinement iterations.

Static C partitioners in the literature, including Glamdring [29], PtrSplit, PM, SeCage [32], and PrivTrans, unfortunately do not include partitioning time statistics for their benchmarks and their implementations are not available for comparison. Our `ander-wpa` mode provides the next-best baseline: `ander-wpa` is *at least* as precise as existing static C partitioners, and Andersen's algorithm is the best known field-sensitive, flow-insensitive pointer analysis; we consider the time taken by Andersen's analysis in `wpa-ander` to be a "floor" for the time taken to statically and conservatively determine a partition with flow-insensitive precision. We infer that the worst-case overhead of achieving flow-sensitivty with `fs-ivfr` is the difference between its total runtime and the pointer analysis runtime of `wpa-ander`, which we find is a significant improvement over `fs-wpa`. We cannot, however, extend these assumptions to non-static partitioners that analyze execution traces (e.g., ProgramCutter) or incorporate developer feedback (e.g., SOAAP [18]).

Our evaluation is guided by the following research questions:

- **RQ1 (Applicability)**: Can CSP partition large real-world software with realistic security demands, and do these partitions require a flow-sensitive analysis?
- **RQ2 (Performance)**: Does iterative value-flow refinement outperform whole-program flow-sensitive pointer anaylsis?
- **RQ3 (Expressiveness)**: Can CSP partition programs whose policies cannot be expressed by existing C partitioners, i.e., many-party computations with mutual distrust?

3.2 RQ1: Applicability

OpenSSH. OpenSSH [58] is an open-source implementation of the SSH protocol for secure networked communication. Because it requires root privileges, it is a target for privilege escalation vulnerabilities [1]; OpenSSH implements a process-based privilege separation; partitioning can further reduce the attack surface of privileged processes. But conservative pointer analysis thwarts this effort; Fig. 10 shows an unmodified snippet from OpenSSH's private key handling code, for which flow-insensitive analysis considers both variables to point to the private key after the swap. Spurious "confidential" value-flows from the private key then taint many functions, including the frequently-called `sshbuf_new()`. If we wish to pin `sshbuf_new()` to an insecure component (perhaps for performance reasons), a flow-sensitive pointer analysis is required.

```
// Make a scrubbed, public-only copy of our private key
if ((r = sshkey_from_private(k, &kswap)) != 0)
    goto out;
tmp = *kswap; // Swap the private key out
*kswap = *k;
*k = tmp;
```

Figure 10: A snippet from `sshkey_shield_private()` in OpenSSH that benefits from flow-sensitive analysis.

We tested CSP on OpenSSH using this policy. The buffer which reads a private key in `sshkey_parse_private2()` is marked a confidential value source, and the swapped public key is declassified. As Table 2 shows, the flow-insensitive VFG (`ander-wpa`) is insufficient to find a partition, while whole-program sparse flow-sensitive analysis (`fs-wpa`) is considerably more expensive. Iterative value-flow refinement (`fs-ivfr`) finds a partition with very little overhead over Andersen's analysis, refining only 6 points-to sets out of 72460.

Wget. Though we have described our policies as tracking "confidential" value-sources, they can as easily model data *integrity*. Wget [40] is a command-line utility that retrieves files via HTTP or FTP; a partition of Wget that isolates the downloaded file can guard the integrity of user data against a malicious download.

We used CSP to partition Wget with the downloaded file buffer as a tainted value, while pinning metadata validation functions on the download (`has_insecure_name_p`, `is_invalid_entry`) to a secure component, so that if the download's component were compromised, it could not bypass validation checks to spoof trust in a malicious file. `ander-wpa` finds a spurious value-flow of the

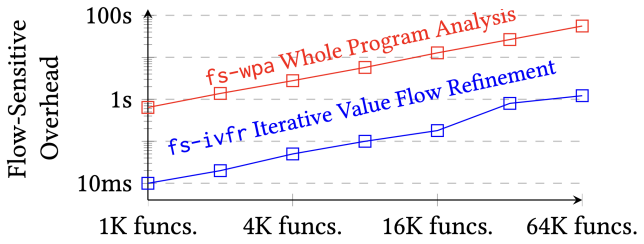

Figure 11: The extra cost of flow-sensitive analysis on Gen

file contents into these validation functions, so a flow-sensitive partition is again needed—we find that `fs-ivfr` needs only 4s for flow-sensitive precision, compared to 20s for `fs-wpa`. Section 3.3 describes our second Wget experiment with a different policy.

Thttpd. Flow-sensitivity is not always needed—For Thttpd [41], a simple HTTP server, we mark the server-side `authfile` as confidential, and pin handlers to an `INTERFACE` component, enforcing that they do not send information about the authfile back to the client, even if the `INTERFACE` is compromised by malicious client requests. In this case, `ander-wpa` is sufficient to find a secure partition. Because `fs-ivfr` starts from Andersen's analysis, it performs no additional work for the same result, as opposed to eager `fs-wpa`.

Overall, we find that some partitions of real-world security-conscious software *do* require flow-sensitivity, and iterative value-flow refinement can efficiently achieve this precision.

3.3 RQ2: Performance

Our results for OpenSSH and Wget clearly demonstrate the performance benefit of querying a small number of pointers for flow-sensitivity. We further assess scalability with a set of synthetic benchmarks; Gen is an autogenerated program with a binary tree callgraph of customizable depth. Each function is modeled after `publish()` (Fig. 3), requiring flow-sensitive tracking of indirect call targets to determine that a `SECURE` value allocated in the root function does not reach a particular `PUBLIC` leaf node.

Fig. 11 compares the overhead of `fs-ivfr`'s queries against `fs-wpa`'s whole-program analysis after computing the initial VFG, showing that value-flow refinement maintains a large constant advantage even at scale; at depth 16, the source file is 300k SLoC.

Our largest real-world benchmarks are open-source mailserver Sendmail [59] and SGX-sqlite3, a version of sqlite3 modified to run in an Intel SGX trusted execution enclave [37]. For Sendmail, we use an arbitrary policy designed to require flow-sensitivity and find `fs-ivfr` spends 5x less time on pointer analysis compared to `fs-wpa`. On SGX-sqlite3, meanwhile, we pin enclave `ocalls` to untrusted syscalls to a separate component from the database connection. Andersen's analysis is sufficient to find the partition, but the cost of pointer analysis blows up—`fs-ivfr` avoids the prohibitive expense of `fs-wpa` (which times out) where it is not necessary.

We use an unsatisfiable policy for Make [56] to test whether CSP fails scalably; we mark `job_counter` as `SECURE`, despite it flowing to the `PUBLIC`-pinned `_getopt_initialize()`. `ander-wpa` quickly finds no partition—`fs-ivfr` refines four different value-flows before agreeing, testing the possibility of a flow-sensitive partition faster than `fs-wpa`. But even though dramatically less time is spent on pointer analysis, those gains are mostly lost to VFG construction overhead. Our value-flow refinement rebuilds the VFG each iteration, since the underlying library SVF does not support updating the VFG "in-place" based on a flow-sensitive query. This is an inefficient approach that leaves performance gains on the table; we leave a bespoke VFG refinement algorithm for future work.

We should also expect successful flow-sensitive partitions to sometimes take multiple refinement iterations with `fs-ivfr`. Our second test of Wget demonstrates this, taking 4 iterations to partition after we modified the policy to pin 20 extra functions to the secure component. These were chosen randomly from a "saturation" test—we collected the set of functions reachable by the file buffer in the flow-insensitive VFG, and compared it against the reachable set in the flow-sensitive VFG. The difference identifies the functions that only a flow-sensitive analysis can assign to the secure component. Since the spurious insecure flows to each are unrelated, `fs-ivfr` needs multiple iterations to eliminate them.

We do not see this behavior with our real, security-aware policies. This is in part a vindication of our technique: while a large program will have many spurious dependencies, `fs-ivfr` only needs to consider those along an insecure path. Nor does the single-round refinement indicate only a single insecure flow; on OpenSSH, `fs-ivfr` queries 6 pointers, but the total number of points-to sets refined is 45, spanning interconnected flows. More likely is that the single-round indicates a collection of closely related insecure flows due to the small, surgical policies we have used to test real security concerns. We feel real policies are more likely to cohere this way compared to the arbitrary policies of Make and our second Wget experiment, but we would be eager to test our technique on real benchmarks with many insecure flows that are both spurious and independent—it is not easy to find or develop these.

Our saturation tests also showed that our partitions are not "trivial", with only a small number of functions; flow-insensitive saturation of confidential values for OpenSSH was 39% of all functions in the program, and 35% for Wget, enabling substantial functionality in both components. In practice, CSP sometimes chooses "lopsided" partitions because we omit balancing or minimizing heuristics.

3.4 RQ3: Expressiveness

Many of the above security policies are inspired by C partitioners in the literature, but they are simple two-component systems with no need for mutual distrust or owner-based declassification. We exercise CSP's ability to handle more complex security requirements with SecDesk, an application purpose-built to test C partitioner CAPO [27]. SecDesk is a webapp that authenticates users via facial recognition and maintains personal information. A vulnerability in the web frontend could expose personal data or hijack the facial recognition service to harvest photo data—a partition should isolate both the recognizer and the lookup service for personal data, with each declassifying data to the webapp, *but not to each other.*

To express this security intent, a policy must support owner-based declassification. We mark personal data as owned by the lookup service and declassified exclusively to the webapp; CSP tracks flows of declassified data through the webapp, guaranteeing that it does not flow to the recognizer. We have also introduced

a flow-sensitive dependency at the boundary of the webapp and recognizer, to exercise `fs-ivfr`. Table 2 shows no performance penalty even when our security constraints are fully exercised, with `fs-ivfr` finding a partition over four times faster than `fs-wpa` and spending only 7s on pointer analysis compared to 104s for `fs-wpa`.

3.5 Limitations And Future Work

We avoided refactoring benchmarks, but this limits CSP along important dimensions. For one, our constraint restricting pointers from passing between components (Section 2.2.2) was often unenforceable, because real C programs use so many pointer parameters. Automatic refactors such as marshalling and unmarshalling cross-component pointers [30], replicating code in multiple components [62], and breaking up functions [23] would benefit CSP.

We did not pursue "non-policy" optimizations; CSP makes no attempt to minimize the size of a high-privilege component [29] or the number of cross-component calls [31], which impact attack surface and performance respectively. We also noticed many spurious flows arising from CSP's lack of *context-sensitivity* [43]—flow- and context-insensitivity often conspire to yield overwhelming false negatives [53]. Context-sensitivity, as well as *path-sensitivity* to branch conditions are promising avenues for improving precision.

Lastly, turning the assignment $p : P \to C$ of definitions to components into an executable application entails dividing source files, linking cross-component calls to a "glue code" implementation (e.g., SGX `ecall`/`ocall` or networked RPC), and potentially replicating `#includes`, dependencies, macros, and struct definitions—a nontrivial effort that benefits from automation and enables dynamic analyses such as measuring a partition's runtime overhead [27, 42].

4 RELATED WORK

Automatic C program partitioners. Lefeuvre et al. [25] categorize partitioners by abstractions and mechanisms. In their framework, CSP is a *static, code-centric* partitioner with *function* granularity and *synchronous message passing* communication. We automate security through *policy refinement* on *annotations* that express a *mutual distrust model* and enforce data *confidentiality* and *integrity*.

Existing C partitioners lack a sparse, demand-driven analysis and do not achieve CSP's flow-sensitive precision. PtrSplit [30], PM [31], and CAPO [27] use a static flow-insensitive pointer analysis. SeCage [32], PrivTrans [8], and ProgramCutter [57] narrow pointer targets with dynamic execution traces, an efficient but unsound approach. Trellis [36] analyzes the call graph but ignores the indirect call targets we heed. SOAAP [18] and AutoSlicer [42] respect instruction ordering for intra-procedural flow-sensitivity, but work on a coarse callgraph from a type-based analysis. TyPM [33] eschews pointer analysis for flow-insensitive type checking. Glamdring [29] tracks confidental data via a VFG without refinement.

Our security policies are rich and easy to define, providing more use cases at the expense of increased analysis complexity. More common is a safebox model with one privileged and one unprivileged component and permissive declassification [8, 29–31, 33, 42]. Trellis provides a hierarchy of levels, but not mutual distrust; SeCage only declassifies to one unprivileged component, and ProgramCutter ignores declassification. SOAAP supports mutual distrust and a `__soaap_declassify` annotation, but not owner declassification. CAPO handles mutual distrust and owner declassification, but declassification is per-function and only operates cross-component. None of these models is expressive enough to capture our owner-based declassification to select parties through select dataflows.

Demand-driven sparse pointer analysis. Our iterative value-flow refinement uses demand-driven sparse pointer analysis for C. Hardekopf et al. [19, 20] introduced sparsity for pointer analysis; SVF [52] refines a sparse VFG for whole-program analysis; and SUPA [53] extends SVF with the *demand-driven* queries we utilize.

Further improvements to demand-driven sparse analysis complement our work. Recent SVF improvements address applicability, performance, and precision [6, 7, 11, 26]. Pinpoint [50], SparseBoomerang [24], and He et al. [21] speed demand-driven sparse pointer analysis by leveraging demand when computing the initial VFG, though this assumes knowledge of later queries, which our iterative refinement cannot provide. Li et al. [28] combine SVF's dataflow-based sparsification with type-based to improve sparsity.

Static taint analysis for C. Program partitioning resembles *taint analysis*, which checks whether taint *sources* such as user inputs can reach taint *sinks* such as system calls. Static taint analysis for C has a long history [49], is naturally demand-driven [17, 54], and benefits from increased precision, but cost pushes modern work towards partial flow-sensitivity. STELLA [10] employs SVF to detect enclave leakage but is not flow-sensitive. PhASAR [48] exercises their C/C++ static analysis framework with taint analysis and supports whole-program flow-sensitive analysis, but this analysis is not sparse. Chen et al. [9] analyzes packet injection using an *intra-procedural* non-sparse flow-sensitive taint analysis that summarizes points-to information from callees, trading off precision differently. UniSan [34], SUTURE [61], and DR. CHECKER [35] detect kernel vulnerabilities with a non-sparse taint analysis that is flow-sensitive for intra-procedural data pointers but relies on a coarse callgraph that guesses call targets by type, again trading precision for performance. Unlike these tools, CSP does not sacrifice performance for flow-sensitive precision due to its sparse, demand-driven approach.

5 CONCLUSION

We have presented *CSP*, an LLVM-based C program partitioner that uses a novel *iterative value-flow refinement* to analyze confidential value-flows flow-sensitively at scale, and employs constraint satisfaction to support *robust security policies* with mutual distrust and owner-based declassification. We have exercised CSP on large real-world C programs with security policies that can only be partitioned via flow-sensitive analysis, and demonstrated that our iterative value-flow refinement achieves significantly better performance than sparse whole-program flow-sensitive analysis. Our scalable flow-sensitivity and robust policies are two enhancements of many under active research, as automatic C partitioners continually strive to meet the needs of industry applications [25].

ACKNOWLEDGMENTS

This material is an extension of prior work supported by DARPA under the GAPS project, and we are grateful for their collaboration and insights. We also thank our reviewers for their thoughtful suggestions, which substantially strengthened this work.

REFERENCES

[1] 2015. OpenSSH CVE-2015-6565 sshd Denial of Service Vulnerability. Available from MITRE, CVE-ID CVE-2015-6565.. https://www.cve.org/CVERecord?id=CVE-2015-6565

[2] 2017. GNU Beep CVE-2018-0492 Privilege Escalation Vulnerability. Available from MITRE, CVE-ID CVE-2018-0492.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-0492

[3] 2018. Microsoft Office CVE-2018-8412 Privilege Escalation Vulnerability. Available from MITRE, CVE-ID CVE-2018-8412.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8412

[4] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C Myers, and Elaine Shi. 2021. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 740–755.

[5] L. O. Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph. D. Dissertation. DIKU, University of Copenhagen.

[6] Mohamad Barbar and Yulei Sui. 2021. Compacting points-to sets through object clustering. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[7] Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object versioning for flow-sensitive pointer analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 222–235.

[8] David Brumley and Dawn Song. 2004. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, Vol. 57.

[9] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 388–400.

[10] Yang Chen, Jianfeng Jiang, Shoumeng Yan, and Hui Xu. 2022. STELLA: sparse taint analysis for enclave leakage detection. *arXiv preprint arXiv:2208.04719* (2022).

[11] Xiao Cheng, Jiawei Wang, and Yulei Sui. 2024. Precise sparse abstract execution via cross-domain interaction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[12] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, K Vikram, Lantian Zheng, and Xin Zheng. 2009. Building secure web applications with automatic partitioning. *Commun. ACM* 52, 2 (2009), 79–87.

[13] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Proceedings of the International Conference on Computer-Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 1855)*. Chicago, Illinois, 154–169.

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349.

[16] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–34.

[17] Neville Grech and Yannis Smaragdakis. 2017. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[18] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1016–1031.

[19] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. *ACM SIGPLAN Notices* 44, 1 (2009), 226–238.

[20] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298.

[21] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 267–279.

[22] M. Hind and A. Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Proceedings of the Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 1503)*. Springer, Pisa, Italy, 57–81.

[23] Zhen Huang, Trent Jaeger, and Gang Tan. 2021. Fine-grained Program Partitioning for Security. In *Proceedings of the 14th European Workshop on Systems Security*. 21–26.

[24] Kadiray Karakaya and Eric Bodden. 2023. Two sparsification strategies for accelerating demand-driven pointer analysis. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 305–316.

[25] Hugo Lefeuvre, Nathan Dautenhahn, David Chisnall, and Pierre Olivier. 2024. SoK: Software Compartmentalization. *arXiv preprint arXiv:2410.08434* (2024).

[26] Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023. Recursive state machine guided graph folding for context-free language reachability. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 318–342.

[27] Maxwell Levatich, Robert Brotzman, Benjamin Flin, Ta Chen, Rajesh Krishnan, and Stephen A Edwards. 2022. C Program Partitioning with Fine-Grained Security Constraints and Post-Partition Verification. In *MILCOM 2022-2022 IEEE Military Communications Conference (MILCOM)*. IEEE, 285–291.

[28] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. 2023. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4211–4228.

[29] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 285–298.

[30] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of Computer and Communications Security (CCS)*. ACM, Dallas, Texas, 2359–2371. https://doi.org/10.1145/3133956.3134066

[31] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1023–1040.

[32] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1607–1619.

[33] Kangjie Lu. 2023. Practical program modularization with type-based dependence analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1256–1270.

[34] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 920–932.

[35] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. 1007–1024.

[36] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. 2016. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 437–456.

[37] Yerzhan Mazhkenov. 2018. SGX_sqlite. https://github.com/yerzhan7/SGX_SQLite.

[38] Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review* 31, 5 (1997), 129–142.

[39] Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.

[40] Hrvoje Niksic. 1997. wget. https://github.com/jnothman/wget.

[41] Jeff Poskanzer. 2004. thttpd. https://github.com/Cloudxtreme/thttpd.

[42] Weizhong Qiang and Hao Luo. 2022. AutoSlicer: Automatic Program Partitioning for Securing Sensitive Data Based-on Data Dependency Analysis and Code Refactoring. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 239–247. https://doi.org/10.1109/TrustCom56396.2022.00042

[43] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 162–186. https://doi.org/10.1145/345099.345137

[44] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of Principles of Programming Languages (POPL)*. San Francisco, 49–61. http://citeseer.nj.nec.com/reps95precise.html

[45] Andrei Sabelfeld and David Sands. 2005. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 255–269.

[46] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.

[47] Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. 2023. HasTEE: Programming Trusted Execution Environments with Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell '23)*. ACM, 72–88. https://doi.org/10.1145/3609026.3609731

[48] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410.

[49] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. 2001. Detecting Format String Vulnerabilities With Type Qualifiers. In *Proceedings of the 10th*

USENIX Security Symposium. Washington, DC, 201–216.

[50] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706.

[51] Geoffrey Smith. 2007. Principles of secure information flow analysis. In *Malware Detection*. Springer, 291–307.

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

[53] Yulei Sui and Jingling Xue. 2018. Value-flow-based demand-driven pointer analysis for C and C++. *IEEE Transactions on Software Engineering* 46, 8 (2018), 812–835.

[54] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*. Springer, 210–225.

[55] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2020. MIR: Automated Quantifiable Privilege Reduction Against Dynamic Library Compromise in JavaScript. *arXiv preprint arXiv:2011.00253* (2020).

[56] wkusnierczyk. 1988. make. https://github.com/wkusnierczyk/make.

[57] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 323–333.

[58] Tatu Ylonen. 1995. openssh-portable. https://github.com/openssh/openssh-portable.

[59] zarzycki. 2003. sendmail. https://github.com/aosm/sendmail/tree/master/sendmail.

[60] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Transactions on Computing Systems* 20, 3 (Aug. 2002), 283–328.

[61] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 811–824.

[62] Lantian Zheng, Stephen Chong, Andrew C Myers, and Steve Zdancewic. 2003. Using replication and partitioning to build secure distributed systems. In *2003 Symposium on Security and Privacy, 2003.* IEEE, 236–250.