Precise Sparse Abstract Execution via Cross-Domain Interaction

Xiao Cheng
University of New South Wales
Sydney, NSW, Australia

Jiawei Wang
University of New South Wales
Sydney, NSW, Australia

Yulei Sui
University of New South Wales
Sydney, NSW, Australia

ABSTRACT

Sparse static analysis offers a more scalable solution compared to its non-sparse counterpart. The basic idea is to first conduct a fast pointer analysis that over-approximates the value-flows and propagates the data-flow facts sparsely along only the pre-computed value-flows instead of all control flow points. Current sparse techniques focus on improving the scalability of the main analysis while maintaining its precision. However, their pointer analyses in both the offline and main phases are inherently imprecise because they rely solely on a single memory address domain without considering values from other domains like the interval domain. Consequently, this leads to conservative alias results, like array-insensitivity, which leaves substantial room for precision improvement of the main data-flow analysis.

This paper presents CSA, a new Cross-domain Sparse Abstract execution that interweaves correlations between values across multiple abstract domains (e.g., memory address and interval domains). Unlike traditional sparse analysis without cross-domain interaction, CSA performs correlation tracking by establishing implications of values from one domain to another. This correlation tracking enables online bidirectional refinement: CSA refines spurious alias relations using interval domain information and also enhances the precision of interval analysis with refined alias results. This contributes to increasingly improved precision and scalability as the main analysis progresses. To improve the efficiency of correlation tracking, we propose an equivalent correlation tracking approach that groups (virtual) memory addresses with equivalent implication results to minimize redundant value joins and storage associated.

We apply CSA on two common assertion-based checking clients, buffer overflow and null dereference detection. Experimental results show that CSA outperforms five open-source tools (Infer, Cppcheck, IKOS, Sparrow and KLEE) on ten large-scale projects. CSA finds 111 real bugs with 68.51% precision, detecting 46.05% more bugs than Infer and exhibiting 12.11% more precision rate than KLEE. CSA records 96.63% less false positives on real-world projects than the version without cross-domain interaction. CSA also exhibits an average speedup of 2.47× and an average memory reduction of 6.14× with equivalent correlation tracking.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639220

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**;
• **Theory of computation** → **Abstraction**.

KEYWORDS

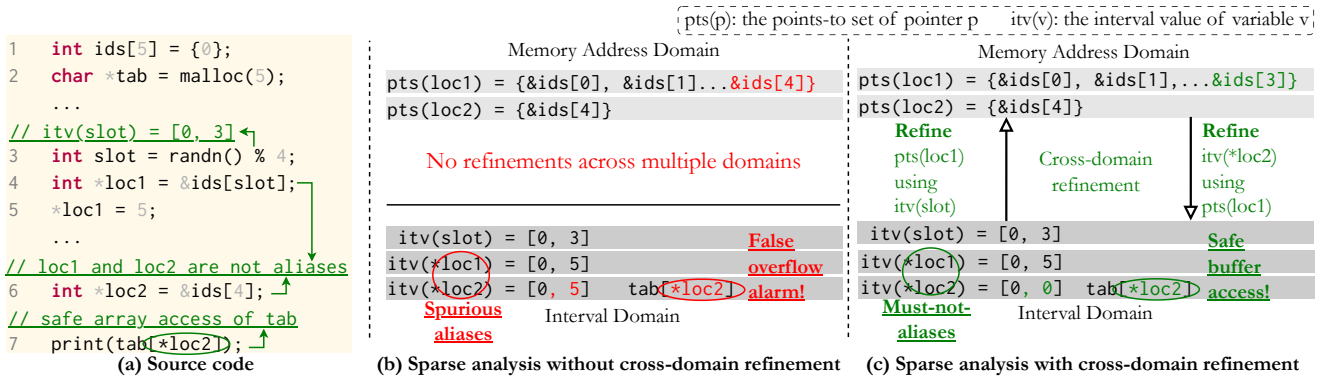
Abstract execution, sparse analysis, cross-domain interaction

ACM Reference Format:
Xiao Cheng, Jiawei Wang, and Yulei Sui. 2024. Precise Sparse Abstract Execution via Cross-Domain Interaction. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639220

1 INTRODUCTION

Sparse static analysis is a more scalable version of its traditional non-sparse counterpart because it prevents expensive data flow propagation along unnecessary control flows [14, 37, 42, 48, 53, 55, 56, 61, 65]. Typical sparse analysis approaches are facilitated by static single assignment (SSA) form, such as LLVM's partial SSA form [39], which explicitly captures the def-use chains for top-level variables, while the def-use relations of address-taken variables (which are only accessible through loads and stores) are established via pre-computed pointer alias information [37, 60]. This results in a sparse representation that allows the propagation of data-flow facts only to the required program points along an over-approximated value-flow graph rather than a control-flow graph, reducing the time and space to compute and maintain the data-flow facts.

Existing efforts and limitations. The current sparse techniques rely solely on conservative pointer analysis performed on a single memory address domain to facilitate the main phase analysis. For example, SFS [37] and SVF [60] conduct flow-sensitive pointer analysis on an over-approximated value-flow graph built with fast Andersen's points-to analysis [10]. Sparrow [49] employs an external flow-insensitive pointer analysis to support the abstract interpretation of scalar variables with pre-computed data dependencies. Fusion [57] utilizes the pointer alias information produced by Pinpoint [56] to resolve path constraints associated with address-taken variables. All these existing sparse approaches leverage auxiliary dependence analysis or pointer analysis, which is conducted on a memory address domain to bootstrap the subsequent main analysis phase. The offline pointer analysis in the pre-analysis phase may lead to redundant data flow propagation along conservative data dependencies that do not exist during runtime, thus impacting the sparsity. Additionally, the pointer analysis in the main phase (e.g., sparse flow-sensitive pointer analysis in SFS and Sparrow), which is unaware of the needs of the main data-flow analysis such as interval analysis, does not leverage the information from other domains, leaving substantial room for precision improvement. Unfortunately, pre-computing a precise dependence (e.g., array-sensitive analysis) can become prohibitively expensive,



(a) Source code (b) Sparse analysis without cross-domain refinement (c) Sparse analysis with cross-domain refinement

Figure 1: Sparse analysis without cross-domain refinement vs sparse analysis with cross-domain refinement.

outweighing the time spent in the main phase and thus defeating the purpose of bootstrap-based sparse analysis [56].

An example. We use an example in Figure 1 to illustrate the imprecision and its introduced redundancy by traditional sparse analysis, which handles two domains without cross-domain refinement. The code fragment in Figure 1(a) represents a typical table/array access scenario. Given an allocated table called `tab` at ℓ_2, it aims to print an item `tab[*loc2]` at ℓ_7 through a location index `*loc2` retrieved from an index array `ids`. Despite the table access at ℓ_7 being safe, both Sparrow [49] and Infer [38] report a false overflow alarm. This is due to imprecisely treating `loc1` and `loc2` as aliases. As a result, the index value 5 stored in `*loc1` at ℓ_5 gets propagated to `*loc2` at ℓ_7 and used for the table access, where `tab[5]` exceeds the size of `tab`. In reality, `loc1` and `loc2` used for table accesses are must-not-aliases. Specifically, `loc1` at ℓ_4 only points to `ids`'s first four elements (`&ids[0]`...`&ids[3]`) given constraint `randn()%4` at ℓ_3, while `loc2` at ℓ_6 only points to the last element `&ids[4]`. Hence any changes to `*loc1` will not affect `*loc2`, which always has a safe index value of 0 initialized at ℓ_1.

A comparison between traditional sparse analysis and the ideal cross-domain analysis is depicted in Figure 1(b) and (c). For traditional analysis [37, 48], both the pre-analysis and main pointer analysis do not consider the information from the numerical domain, making `loc1` conservatively point to all objects in `ids` because the analyses do not consider the interval value $[0, 3]$ for scalar `slot`. For example, Sparrow [49] aggregates all memory addresses of `ids` into a single abstract location when performing pointer analysis. Without the interval result `itv(slot)` = $[0, 3]$ at ℓ_4, the points-to set `pts(loc1)` imprecisely includes `&ids[4]`. Consequently, both `*loc1` and `*loc2` refer to the same object. In turn, this imprecise pointer aliasing causes imprecise interval results, making `*loc2` yield a conservative interval value $[0, 5]$ by imprecisely including the value 5 from `*loc1`. The imprecise interval value of `*loc2` when accessing `tab` at ℓ_7 triggers a false overflow alarm. In contrast, Figure 1(c) shows a precise result across two domains. The points-to set `pts(loc1)` becomes more accurate by excluding `&ids[4]` when considering the interval value of `itv(slot)` from the interval domain. As a result, `loc1` and `loc2` are precisely identified as must-not-aliases, eliminating the spurious def-use chain between ℓ_5 and ℓ_7. Meanwhile, the pointer analysis on the memory address domain refines the interval value `itv(*loc2)` at ℓ_7. Thus, at ℓ_7, `*loc2` (`ids[4]`) has a precise interval value of $[0, 0]$ rather than $[0, 5]$ derived based on the isolated pointer analysis. Hence no false alarms are reported.

Challenges. To address the imprecision of existing sparse techniques, any pointer alias analysis needs to work simultaneously with the data-flow analysis phase on a combined abstract domain to provide improved precision. There are two major challenges: *(1) The analysis across multiple domains* is more precise but more costly because the lattice of a combined pointer and numerical domain can become extremely large and difficult to compute a fixed point without an efficient online refinement. Moreover, the soundness of sparse analysis must not be compromised during the cross-domain interaction. *(2) Excessive memory objects* are another challenge since the number of memory objects can grow substantially when dealing with field- and array-sensitive analysis, like handling each element in `ids` as shown in Figure 1, particularly in large programs. It becomes necessary, albeit difficult, to minimize the computational and memory expenses involved in performing points-to and interval analysis in the presence of excessive memory objects.

Our solution. We introduce CSA, a precise and efficient sparse abstract execution over multiple abstract domains. To tackle Challenge (1), we present *cross-domain refinement*, which effectively captures the correlation between pointer and interval analyses. Rather than simply combining the results from each analysis domain, we develop a novel use of reduced cardinal power [22] to enable an online bidirectional refinement for both analyses. The def-use dependence is initially established using a fast over-approximated pointer analysis [10, 37]. As the main analysis progresses, the pointer alias information is gradually refined on the fly, reducing data flow propagation along spurious data dependencies generated by the offline analysis. Simultaneously, the precise pointer alias information boosts the precision of the interval analysis in the main phase. To address Challenge (2), we propose *equivalent correlation tracking*, which efficiently handles excessive memory objects. This approach identifies and collapses memory address sets with equivalent implication results, aiming to minimize computational overheads and memory costs when dealing with memory objects.

Framework overview. Figure 2 provides an overview of CSA:

(a) Input. The input of our framework is LLVM's intermediate representation (IR) [39], as formally defined in Section 2.2. This IR is then passed to SVF [60], a static analysis tool that produces the interprocedural control flow graph (ICFG) and sparse value flow graph (SVFG) using Andersen's pointer analysis [10].

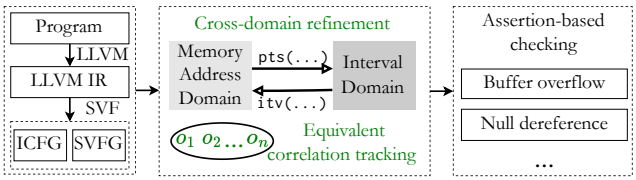


Figure 2: An overview of our framework.

(b) Sparse analysis. CSA works on a combination of two abstract domains, including the memory address domain for pointer information and the interval domain for integer values. The inclusion of interval value `itv(...)` enhances pointer analysis precision, while the precise points-to set `pts(...)`, in turn, improves data dependence and makes the interval analysis more precise. This is accomplished by establishing the implication relations between the two domains through reduced cardinal power. Redundant value joins implied by memory addresses with equivalent correlations are eliminated.

(c) Clients. CSA supports assertion-based checking clients like buffer overflow and null dereference detection. In summary, this paper makes the following major contributions:

- We introduce CSA, a precise cross-domain sparse abstract execution over a combined domain through correlation tracking. Cross-domain refinement enhances the precision of both pointer and interval analysis.
- We propose an implication-equivalent (virtual) memory address grouping approach that efficiently captures correlations between domains by eliminating redundant value join operations and memory costs.
- We conduct a comprehensive evaluation of CSA's performance using 7774 programs from the NIST dataset and ten real-world open-source projects. Experimental results show that CSA detects 46.05% more bugs than Infer and achieves 12.11% more precision rate than KLEE in the ten real-world projects. CSA reduces false positives by 96.63% on real-world projects compared to the version without cross-domain interaction. CSA also demonstrates an average speedup of 2.47× and an average memory reduction of 6.14× when using equivalent correlation tracking.

2 BACKGROUND

We first introduce the basic concepts of combined abstract domains and then describe the target language and sparse abstract execution.

2.1 Combined Abstract Domains

Concrete and abstract domains. The elements of the *abstract domain* $\mathbb{A}$ are abstract values that approximate a set of concrete values, i.e., the values that a variable can take in the *concrete domain* $\mathbb{C}$ during program execution (e.g., integers, floats and strings). The abstract domain $\mathbb{A}$ is an over-approximated abstraction of $\mathbb{C}$ with a concretization function $\gamma \in \mathbb{A} \rightarrow \mathbb{C}$ based on a partial order $\sqsubseteq$ over $\mathbb{A}$ such that $\forall a, a' \in \mathbb{A}, a \sqsubseteq a' \Leftrightarrow \gamma(a) \subseteq \gamma(a')$. The partial order relations of an abstract domain $\mathbb{A}$ form a lattice $\mathfrak{A} = \langle \mathbb{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$, where $\sqcap$ and $\sqcup$ are the meet and join operations, and $\bot$ and $\top$ are unique least and greatest elements of $\mathbb{A}$.

Table 1: LLVM-like Language

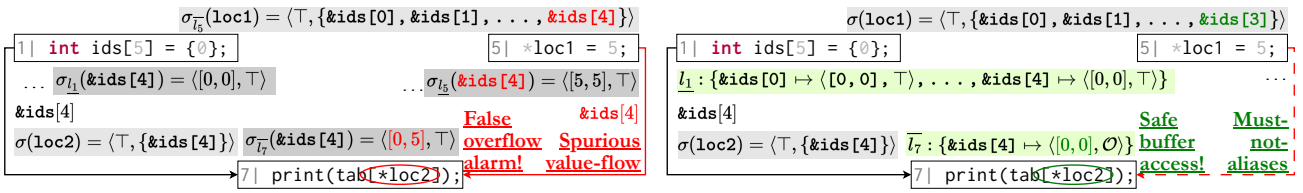
ℓ ::=	
`p = c`	ConsStmt
`p = alloc`$_o$	AddrStmt
`p = &(q→fld)`	GepStmt (Field)
`p = &q[c]` | `p = &q[v]`	GepStmt (Array)
`p = *q`	LoadStmt
`*p = q`	StoreStmt
`p = q`	CopyStmt
`p = phi(`$p_1, p_2, ...p_n$`)`	PhiStmt
`p = ¬q`	UnaryStmt
`r = p` $\odot$ `q`	BinaryStmt
$\odot \in$ {+, -, *, /, %, <<, >>, <, >, &, &&, <=,>=, ≡, ~, |, ∧ }	

Cartesian product. Using a single abstract domain may not be able to precisely cover all program semantics. For instance, one single abstract domain cannot be precisely applied to both interval and pointer analysis. Traditional sparse analysis like the example in Figure 1(b) working over individual domains can be seen as using the Cartesian product × (direct product), which encapsulates the abstract values from each domain [9, 43] with concretization in Definition 1. In this paper, the semantic of Cartesian product simply unifies the value representation where each element of the combined domain ($\mathbb{A} = \mathbb{A}_1 \times ... \times \mathbb{A}_n$) becomes an n-tuple with independently computed values from individual analyses (such as pointer analysis and interval analysis) [21, 24, 54]. In the following sections, the Cartesian product is one of our baselines (referred to as CSA-CP), which is simply a conjunction of the analysis results from different domains with capturing correlations across domains.

Definition 1 (Concretization for Cartesian Product [21, 24, 54])**.** Let $\langle a_1, \ldots, a_n \rangle$ be an abstract value in $\mathbb{A}_1 \times ... \times \mathbb{A}_n$, and $C_1 = \gamma_1(a_1)$, ..., $C_n = \gamma_n(a_n)$ be their corresponding concrete values. The concretized value of $\langle a_1, \ldots, a_n \rangle$ is the intersection of C_1 to C_n, i.e., $\gamma(\langle a_1, \ldots, a_n \rangle) = C_1 \cap \cdots \cap C_n$.

2.2 Language

We perform our sparse analysis on LLVM-like language [37, 39] with each type of program statement listed in Table 1. The set of all variables $\mathcal{V}$ are separated into two subsets, $\mathcal{O}$ that contains all possible abstract objects, i.e., *address-taken variables* of a pointer and $\mathcal{P}$ that contains all *top-level variables*. In LLVM's language, a top-level variable `p, q, r` $\in \mathcal{P}$, including stack virtual registers and global variables, can only be defined once. An address-taken object `o` $\in \mathcal{O}$ can be read/modified only through dereferencing top-level pointers at LoadStmt/StoreStmt. A constant value `c` is always first assigned to a top-level variable at ConsStmt. For AddrStmt `p = alloc`$_o$, `o` is a stack or global variable or a dynamically created abstract heap object. GepStmt models the field accesses of a struct object with its field offset `fld` as a constant value. CSA uses a field-index-based approach to field-sensitivity similar to [12, 52]. The fields of a struct object are distinguished by their unique indices. The variable index when accessing an array (e.g., `p = &q[v]`) is resolved during abstract execution. PhiStmt is a standard SSA instruction introduced at a confluence point on the control flow graph to select the value of a variable from different branches. Passing parameters to and returning results from a callee invoked at a callsite are modeled as CopyStmts.



(a) Sparse analysis over combined domains using Cartesian product (b) Sparse analysis over combined domains using correlation tracking

Figure 3: Sparse analysis on the memory address and interval domains by revisiting the example in Figure 1.

2.3 Abstract State and Abstract Execution

Abstract execution (or abstract interpretation) is the process of collecting program semantics by deriving the abstract state of program variables (Definition 2) at each program control point to form an abstract trace σ (Definition 3). At a high level, abstract execution follows the control flow graph and updates σ by analyzing each statement until a fixed point is reached.

Definition 2 (Abstract State $AS : \mathcal{V} \to \mathbb{A}$)**.** We define the *abstract state* for a given program as a map $AS = \mathcal{V} \to \mathbb{A}$ from program variables $\mathcal{V}$ to the abstract domain $\mathbb{A}$.

Definition 3 (Abstract Trace $\sigma : \mathbb{L} \times \mathcal{V} \to \mathbb{A}$**).** Abstract trace is a set of abstract states with each qualified by a program point. $\sigma_L(x)$ returns x's value at a particular program point $L \in \mathbb{L}$ from $\mathbb{A}$, where L can be a program point immediately before ($\overline{\ell}$) or after ($\underline{\ell}$) program statement ℓ.

2.4 Sparse Abstract Execution

Sparse abstract execution propagates abstract states based on sound yet over-approximated data dependence chains. It only maintains the abstract values of variables where necessary (the use sites of these variables) during the analysis, reducing the size of states at each program point and saving memory costs. The data dependencies are the def-use chains on the value flow graph, which is initially built upon the partial SSA form by considering program control flows and pre-computed points-to results [10, 37]. Top-level variables are in LLVM's SSA form and their def-use chains are directly obtained. Address-taken variables are obtained by building the interprocedural memory SSA form using Andersen's points-to results [10]. We use $\ell \overset{o}{\hookrightarrow} \ell'$ to represent the value-flow of an address-taken object $o \in O$ which is defined at ℓ and used at ℓ'.

3 MOTIVATING EXAMPLE

Figure 3 illustrates the key ideas of CSA by revisiting the example in Figure 1. We aim to compare and contrast the traditional sparse analysis using the Cartesian product in Figure 3(a), with a precise analysis by tracking the correlation between the memory address and interval domains in Figure 3(b). Definition 4 gives notations for the interval and memory address domains.

Definition 4 (Interval and Memory Address Domains)**.** The *interval domain* [22] represents a set of integers that fall between two given endpoints, which is equipped with a lattice $\mathit{Interval}\ \langle \mathbb{I}, \sqsubseteq, \sqcap, \sqcup, \bot, [-\infty, +\infty] \rangle$, where $\mathbb{I} = \{[a, b] \mid a, b \in \mathbb{Z} \cup [-\infty, +\infty]\} \cup \{\bot\}$ ($\mathbb{Z}$ denotes all integers) is the set of all intervals. The abstraction for memory addresses is a set of discrete values. The lattice for *MemAddress* domain is $\langle \mathbb{O}, \subseteq, \cap, \cup, \varnothing, \top \rangle$, where $\mathbb{O} = \mathcal{P}(O)$ is the powerset of the set O representing all allocated memory addresses.

Analysis over combined domains using Cartesian product. In this case, the results of pointer analysis and interval analysis are simply combined with the Cartesian product ($\sigma \in \mathbb{L} \times \mathcal{V} \to \mathit{Interval} \times \mathit{MemAddress}$) without cross-domain refinement. As shown in Figure 3(a), the abstract value of the pointer variable `loc1` at program point $\overline{\ell_5}$ is represented by a pair $\sigma_{\overline{\ell_5}}(\texttt{loc1}) = \langle \top, \{\&\texttt{ids[0]}, \&\texttt{ids[1]}, ..., \&\texttt{ids[4]}\} \rangle$ (some approaches [38, 49] aggregate `&ids[0]`, `&ids[1]`, ..., `&ids[4]` to one object) where the first element $\top$ is a conservative value of this pointer by interval analysis. The second element is a set of memory addresses pointed by `loc1` computed by an external pointer analysis over the memory address domain. Because the pointer analysis does not have interval information, `loc1` conservatively points to all the elements of `ids` including a spurious target `&ids[4]`, which establishes a false data dependence $\ell_5 \overset{\&\texttt{ids[4]}}{\hookrightarrow} \ell_7$. This false dependence propagates value $\sigma_{\underline{\ell_5}}(\&\texttt{ids[4]}) = \langle [5, 5], \top \rangle$ to $*\texttt{loc2}$ at $\overline{\ell_7}$ ($\top$ here represents a conservative points-to result as a constant integer can be interpreted as a memory address value in LLVM). Consequently, $*\texttt{loc2}$ used as an index to access `tab` has an imprecise value $[0, 5]$ which may exceed `tab`'s boundary, causing a false alarm.

Analysis over combined domains using correlation tracking. Unlike the Cartesian product, the cross-domain analysis uses σ only to maintain abstract trace for top-level variables $\mathcal{P}$ (Section 2.2). The trace is then reduced to $\sigma \in \mathcal{P} \to \mathit{Interval} \times \mathit{MemAddress}$. To keep track of the values of memory objects O, the analysis needs to capture the correlations across abstract domains between *MemAddress* and *Intervals* when analyzing pointer-related statements including GepStmt, LoadStmt and StoreStmt. The correlation is captured by establishing an implication that maps each memory object to its implied memory address or interval value. For example, at $\underline{\ell_1}$ in Figure 3(b), we create an implication from each memory object in `ids` to the initial value, that is $\&\texttt{ids[0]} \mapsto \langle [0, 0], \top \rangle ... \&\texttt{ids[4]} \mapsto \langle [0, 0], \top \rangle$. Each of these implications represents that an object if accessed at $\underline{\ell_1}$ has a value $[0, 0]$, i.e., the implication result for the object is $[0, \overline{0}]$. The implication $\&\texttt{ids[4]} \mapsto \langle [0, 0], \top \rangle$ is then propagated to ℓ_7 via the data dependence $\ell_1 \overset{\&\texttt{ids[4]}}{\hookrightarrow} \ell_7$. At $\overline{\ell_5}$, unlike the analysis performed on a single memory address domain, the cross-domain analysis considers the interval value $\sigma(\texttt{slot}) = \langle [0, 3], \top \rangle$, and derives a precise memory address value of `loc1`, $\texttt{pts}(\texttt{loc1}) = \{\&\texttt{ids[0]}, \&\texttt{ids[1]}, ..., \&\texttt{ids[3]}\}$. This precise points-to result in turn helps remove the spurious dependence $\ell_5 \overset{\&\texttt{ids[4]}}{\hookrightarrow} \ell_7$. Consequently, it contributes to a more precise interval value $\langle [0, 0], \top \rangle$

$$[\textsc{ValueFlow}]\ \frac{\ell' \overset{o}{\hookrightarrow} \ell}{\delta_{\overline{\ell}}(o) \supseteq \delta_{\underline{\ell'}}(o)} \qquad [\textsc{ConsStmt}]\ \frac{\ell : \mathsf{p} = \mathsf{c}}{\sigma(\mathsf{p}) := \langle \alpha(\{\mathsf{c}\}), \top \rangle} \qquad [\textsc{CopyStmt}]\ \frac{\ell : \mathsf{p} = \mathsf{q}}{\sigma(\mathsf{p}) := \sigma(\mathsf{q})}$$

$$[\textsc{PhiStmt}]\ \frac{\ell : \mathsf{r} = \mathsf{phi}(\mathsf{p}_1, \mathsf{p}_2, ..., \mathsf{p}_n)}{\sigma(\mathsf{r}) := \bigsqcup_{i=1}^{n} \sigma(\mathsf{p}_i)} \qquad [\textsc{UnaryStmt}]\ \frac{\ell : \mathsf{p} = \neg \mathsf{q}}{\sigma(\mathsf{p}) := \neg^{\#} \sigma(\mathsf{q})} \qquad [\textsc{BinaryStmt}]\ \frac{\ell : \mathsf{r} = \mathsf{p} \odot \mathsf{q}}{\sigma(\mathsf{r}) := \sigma(\mathsf{p}) \odot^{\#} \sigma(\mathsf{q})}$$

$$[\textsc{AddrStmt}]\ \frac{\ell : \mathsf{p} = \mathtt{alloc}_{\mathsf{o_i}}}{\sigma(\mathsf{p}) := \langle \top, \{o_i\} \rangle} \qquad [\textsc{GepStmt}]\ \frac{\ell : \mathsf{p} = \&(\mathsf{q} \rightarrow \mathsf{i})\ \textit{or}\ \ell : \mathsf{p} = \&(\mathsf{q}[\mathsf{i}])}{\sigma(\mathsf{p}) := \bigsqcup_{o \in \gamma(\sigma(\mathsf{q}))} \bigsqcup_{j \in \gamma(\sigma(\mathsf{i}))} \langle \top, \{\mathsf{o.fld}_j\} \rangle}$$

$$[\textsc{LoadStmt}]\ \frac{\ell : \mathsf{p} = *\mathsf{q}}{\sigma(\mathsf{p}) := \bigsqcup_{o \in \{o \mid (o \mapsto \_) \in \delta_{\overline{\ell}}\}} \delta_{\overline{\ell}}(o)} \qquad [\textsc{StoreStmt}]\ \frac{\ell : *\mathsf{p} = \mathsf{q}}{\delta_{\underline{\ell}} \supseteq (\{o \mapsto \sigma(\mathsf{q}) | o \in \gamma(\sigma(\mathsf{p}))\} \sqcup \delta_{\overline{\ell}} \setminus \mathsf{kill}(\ell))}$$

$$\mathsf{kill}(\ell : *\mathsf{p} = \mathsf{q}) := \begin{cases} \{\mathsf{o} \mapsto \_ \mid \mathsf{o} \in \gamma(\sigma(\mathsf{p}))\} & \text{if } \sigma(\mathsf{p}) \equiv \langle \top, \{\mathsf{o}\} \rangle \wedge \mathsf{o} \text{ is singleton} \\ \{\mathsf{o} \mapsto \_ \mid \mathsf{o} \in O\} & \text{if } \sigma(\mathsf{p}) \equiv \langle \top, \varnothing \rangle \\ \varnothing & \text{otherwise} \end{cases}$$

Figure 4: Analysis rules for sparse abstract execution by tracking correlations across domains. $a \supseteq b$ represents $a := a \sqcup b$. $\mathsf{f}^{\#}$ is the abstract operator concerning the concrete operator f. $\gamma(\sigma(\mathsf{p}))$ concretes the abstract value $\sigma(\mathsf{p})$ based on Definition 1 and returns the memory address set pointed by p or the integers residing inside the interval value of variable p. $\mathsf{o.fld_j}$ represents the j^{th} field of the base object o.

of $*$loc2 at ℓ_7. As a result, $*$loc2 can only have the value [0, 0] copied from &ids[4] at $\underline{\ell_1}$ via value-flow $\ell_1 \overset{\&ids[4]}{\hookrightarrow} \ell_7$. Hence the table access tab[$*$loc2] is safe, no false alarm is reported.

4 APPROACH

Section 4.1 introduces the details of correlation tracking across domains (with analysis rules in Figure 4). Section 4.2 discusses our implication-equivalent approach (with analysis rules in Figure 5).

4.1 Correlation Tracking Across Domains

We present precise correlation tracking through domain interaction by using reduced cardinal power [22] that efficiently maps the abstract value from one domain to the other.

Definition 5 (Implication and implication result)**.** An *implication* $a_1 \mapsto a_2$ in reduced cardinal power maps $a_1 \in \mathbb{A}_1$ to $a_2 \in \mathbb{A}_2$, where $\mathbb{A}_1$ is *MemAddress* and $\mathbb{A}_2$ is *Interval*×*MemAddress*. The implication signifies that if memory address a_1 is accessed at a pointer dereference, then a_2 is the value held at that address. The *implication result* of a_1 is a_2.

In our analysis, $\mathbb{A}_1$ represents only the *MemAddress* domain (Definition 4) because the implication is only used for tracing values of memory objects, while for top-level variables (Section 2.2), we still use abstract trace (Definition 3). Domain $\mathbb{A}_2$ represents *Interval*×*MemAddress* given that each implication result can be either an interval or a memory address. This is because a memory address stores either a numerical value or other memory addresses.

Definition 6 (**Abstract power trace** $\delta : \mathbb{L} \rightarrow (\mathbb{A}_1 \rightarrow \mathbb{A}_2)$)**.** We define an *abstract power trace* δ to track the implications at each program point. We use $\delta_L = \{a_1 \mapsto a_2 \mid a_1 \in \mathbb{A}_1 \wedge a_2 \in \mathbb{A}_2\}$ to represent a set of implications at program point L. $\delta_L(a_1)$ returns the implication result of a_1 at program point L.

Note that implications (or correlations) are derived online via cross-domain interaction while existing sparse techniques [37, 49, 60] use pre-computed points-to results without online refinement. Unlike existing analyzers [37, 49, 60] that aggregate the memory objects within arrays, the implications in δ_L support fine-grained array-sensitive correlation tracking. This representation also supports tracking a group of objects (Section 4.2).

Analysis rules. Figure 4 presents the abstract execution rules using reduced cardinal power. For sparse analysis, the abstract power states are propagated through value flows via the ValueFlow rule, which depends on the pointer information derived on the fly. For the pointer-free statements (ConsStmt, CopyStmt, PhiStmt, UnaryStmt, BinaryStmt), we update the value of the left-hand-side variable based on the right-hand-side value and relevant operators. For instance, given a BinaryStmt $\ell : \mathsf{r} = \mathsf{p} + \mathsf{q}$ and $\sigma(\mathsf{p}) := \langle [1, 5], \top \rangle, \sigma(\mathsf{q}) := \langle [4, 5], \top \rangle$, we derive $\sigma(\mathsf{r}) := \langle [5, 10], \top \rangle$.

We show pointer-related analysis rules, in particular, GepStmt, LoadStmt and StoreStmt, to conduct online value-flow refinement by capturing correlation between domains.

The GepStmt rule updates the value in the memory address domain with the information from the interval domain. For GepStmt where the offset is a variable, a more precise field- and array-sensitive analysis can capture the correlation between the two domains by considering the numerical value of the offset to refine the resulting memory addresses of an object's fields. Meanwhile, the refined pointer result remains conservative because the interval value of the offset variable is over-approximated. In contrast, without knowing the numeric value of the offset variable, the standalone pointer analysis returns all possible fields or aggregates all fields to one object to ensure soundness, which can be too conservative and compromise precision.

When processing a LoadStmt $\ell : \mathsf{p} = *\mathsf{q}$, we refine the derived value of p based on the precise memory addresses of q and the correlation from the abstract power trace $\delta_{\overline{\ell}}$. We first obtain the memory addresses of q via $\sigma(\mathsf{q})$ from the memory address domain. For each memory address o that q refers to, we retrieve the interval value or memory addresses related to o based on the implications stored in $\delta_{\overline{\ell}}$. The final result for $\sigma(\mathsf{p})$ is the join of all implied values of the memory addresses.

For a StoreStmt $\ell : *\mathsf{p} = \mathsf{q}$, we construct the implication based on $\sigma(\mathsf{p})$ and $\sigma(\mathsf{q})$. We first obtain p's points-to targets via $\sigma(\mathsf{p})$ from the memory address domain. For each memory address o that p refers to, we map o to the abstract value of q in the power state, i.e., $\mathsf{o} \mapsto \sigma(\mathsf{q})$. We also employ strong updates [41] for singletons, which contain all address-taken objects except the local variables in recursion, arrays or heap objects, to enhance precision. The singleton information is determined by the pre-analysis [60].

$$[\textsc{ValueFlow}]\ \frac{\ell' \overset{rf}{\hookrightarrow} \ell}{\delta_{\overline{\ell}} := \delta_{\overline{\ell}} \boxed{\bullet} \{rf \mapsto \delta_{\underline{\ell'}}(rf)\}} \qquad [\textsc{AddrStmt}]\ \frac{\ell : \mathsf{p} = \mathsf{alloc}_{\mathsf{o_i}}}{\sigma(\mathsf{p}) := \langle \top, rf_{\{\mathsf{o_i}\}} \rangle}$$

$$[\textsc{GepStmt}]\ \frac{\ell : \mathsf{p} = \&(\mathsf{q} \rightarrow \mathsf{i})\ \textit{or}\ \ell : \mathsf{p} = \&(\mathsf{q}[\mathsf{i}])}{\sigma(\mathsf{p}) := \bigsqcup_{\mathsf{o} \in dr(\gamma(\sigma(\mathsf{q})))} \bigsqcup_{j \in \gamma(\sigma(\mathsf{i}))} \langle \top, rf_{\{\mathsf{o.fld}_j\}} \rangle} \qquad [\textsc{LoadStmt}]\ \frac{\ell : \mathsf{p} = *\mathsf{q}}{\sigma(\mathsf{p}) := \bigsqcup_{rf \in \{rf \mid (rf \mapsto \_) \in \delta_{\overline{\ell}} \wedge\ rf \cap \sigma(q) \neq \emptyset\}} \delta_{\overline{\ell}}(rf)}$$

$$[\textsc{StoreStmt}]\ \frac{\ell : *\mathsf{p} = \mathsf{q}}{\delta_{\underline{\ell}} := \delta_{\underline{\ell}} \boxed{\bullet} \{\gamma(\sigma(\mathsf{p})) \mapsto \sigma(\mathsf{q})\} \boxed{\bullet} \delta_{\overline{\ell}} \setminus \mathsf{kill}(\ell)}$$

$$\mathsf{kill}(\ell : *\mathsf{p} = \mathsf{q}) := \begin{cases} \{rf_{\{\mathsf{o}\}} \mapsto \_ \mid \mathsf{o} \in \gamma(\sigma(\mathsf{p}))\} & \text{if } \sigma(\mathsf{p}) \equiv \langle \top, \{rf_{\{\mathsf{o}\}}\} \rangle \wedge \mathsf{o} \text{ is singleton} \\ \{rf_{\{\mathsf{o}\}} \mapsto \_ \mid \mathsf{o} \in O\} & \text{if } \sigma(\mathsf{p}) \equiv \langle \top, \emptyset \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 5: Analysis rules for sparse abstract execution with equivalent correlation tracking across domains. The rules for pointer-free statements (ConsStmt, CopyStmt, PhiStmt, UnaryStmt, BinaryStmt) are the same as those in Figure 4.

Join of two power states. The join of two power states, $\delta_{L_1} \sqcup \delta_{L_2}$, is utilized in Rule StoreStmt in Figure 4. The join is defined as follows: for each memory address o where the two states intersect, a join operation is performed to combine their implied values, i.e., $\delta_{L_1}(\mathsf{o}) \sqcup \delta_{L_2}(\mathsf{o})$. The remaining elements in these states are left unchanged and are merged into the resulting abstract power state.

4.2 Equivalent Correlation Tracking

The correlation tracking in Section 4.1 builds implications for individual objects. This single-object correlation tracking approach may introduce redundant join operations for memory addresses holding equivalent implication results, which may be very costly. Furthermore, the single-object approach can consume much more memory for redundantly storing abstract values. To avoid redundant computation and storage, we introduce an implication-equivalent tracking approach, which merges the memory objects which have the same implication results, so that we could analyze and store these implication-equivalent objects only once without sacrificing precision or soundness.

Figure 6 gives an example to compare and contrast the single-object approach and implication-equivalent correlation tracking approach. There are three pointers `loc1` (pointing to o_1 and o_2), `loc2` (pointing to o_3 and o_4) and `loc3` (pointing to o_1, o_2, o_3 and o_4) at ℓ_1, ℓ_2 and ℓ_3 respectively. At ℓ_1, the constant 1 is stored at the addresses that `loc1` points to. At ℓ_2, 5 is stored at the addresses that `loc2` points to. Finally, at ℓ_3, the states from ℓ_1 and ℓ_2 are merged, and r is assigned the value stored at the memory address pointed to by `loc3`. The derived abstract states and power states are shown in Figure 6. In the following paragraphs, we introduce implication-equivalent memory addresses and compare and contrast the difference between single-object (introduced in Section 4.1) and implication-equivalent tracking approaches.

Identifying implication-equivalent memory addresses. The abstract power trace, as defined in Definition 6, may include multiple memory objects that yield equivalent implication results. An example of this scenario can be observed in Figure 6, where o_1 and o_2 are considered implication-equivalent because o_1 and o_2 are both pointed by `loc1` implying the same value $\langle [1,1], \top \rangle$ at the StoreStmt ℓ_1. Consequently, analyzing o_1 and o_2 requires only a single implication statement: $\{o_1, o_2\} \mapsto \langle [1,1], \top \rangle$. This approach can reduce redundancy when building implications.

In Figure 6, we compare the abstract states derived from the single-object and the implication-equivalent correlation tracking.

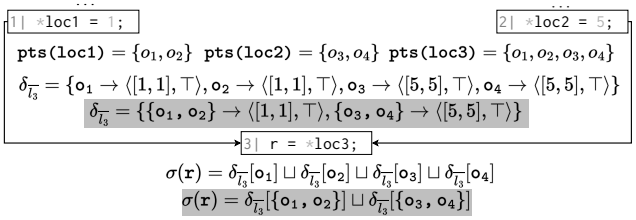


Figure 6: Abstract traces by single-object and implication-equivalent correlation tracking. The implication-equivalent power is highlighted in grey.

For the single-object version at $\underline{\ell_1}$, we build an implication from a group of memory addresses (o_1 and o_2) referred by p to the value $\langle [1,1], \top \rangle$ independently. For the implication-equivalent version, we combine o_1 and o_2 by building an implication from the set $\{\mathsf{o}_1, \mathsf{o}_2\}$ to its value:

- Single-object: $\delta_{\underline{\ell_1}} = \{\mathsf{o}_1 \mapsto \langle [1,1], \top \rangle, \mathsf{o}_2 \mapsto \langle [1,1], \top \rangle\}$
- **Implication-equivalent**: $\delta_{\underline{\ell_1}} = \{\{\mathsf{o}_1, \mathsf{o}_2\} \mapsto \langle [1,1], \top \rangle\}$

The size of the implication-equivalent abstract power state is only half the number of the single-object states because $\langle [1,1], \top \rangle$ is stored only once. The propagation of the state is also saved when performing abstract execution.

At the value flow joint point $\overline{\ell_3}$, $\delta_{\underline{\ell_1}}$ and $\delta_{\underline{\ell_2}}$ are merged, yielding the following states:

- Single-object: $\delta_{\overline{\ell_3}} = \{\mathsf{o}_1 \mapsto \langle [1,1], \top \rangle, \mathsf{o}_2 \mapsto \langle [1,1], \top \rangle, \mathsf{o}_3 \mapsto \langle [5,5], \top \rangle, \mathsf{o}_4 \mapsto \langle [5,5], \top \rangle\}$
- **Implication-equivalent**: $\delta_{\overline{\ell_3}} = \{\{\mathsf{o}_1, \mathsf{o}_2\} \mapsto \langle [1,1], \top \rangle, \{\mathsf{o}_3, \mathsf{o}_4\} \mapsto \langle [5,5], \top \rangle\}$

Like the states at $\underline{\ell_1}$ and $\underline{\ell_2}$, the implication-equivalent power state $\delta_{\overline{\ell_3}}$ costs only half memory space.

For the LoadStmt at ℓ_3, the single-object method joins all the values implied by these four memory addresses (o_1, o_2, o_3 and o_4) pointed by p, while the implication-equivalent version joins all the values implied by the two grouped memory address sets $\{\mathsf{o}_1, \mathsf{o}_2\}$ and $\{\mathsf{o}_3, \mathsf{o}_4\}$:

- Single-object: $\delta_{\overline{l_3}}(\mathsf{o}_1) \sqcup \delta_{\overline{l_3}}(\mathsf{o}_2) \sqcup \delta_{\overline{l_3}}(\mathsf{o}_3) \sqcup \delta_{\overline{l_3}}(\mathsf{o}_4)$
- **Implication-equivalent**: $\delta_{\overline{l_3}}(\{\mathsf{o}_1, \mathsf{o}_2\}) \sqcup \delta_{\overline{l_3}}(\{\mathsf{o}_3, \mathsf{o}_4\})$

Only one join operation is needed for the implication-equivalent power, while it takes three join operations by the single-object one.

Algorithm 1: Implication-equivalent join operator of two abstract power states.

```
1  Function ⍟(δ1, δ2):
2      Initialize δ with an empty map;
3      for (rf1 ↦ a1) ∈ δ1 do
4          for (rf2 ↦ a2) ∈ δ2 do
5              δ(rf1 ∩ rf2) := a1 ⊔ a2
6      for (rf1 ↦ a1) ∈ δ1 do
7          δ(rf1 \ ⋃{rf2|(rf2↦_)∈δ2}) := a1
8      for (rf2 ↦ a2) ∈ δ2 do
9          δ(rf2 \ ⋃{rf1|(rf1↦_)∈δ1}) := a2
10     return δ
```

Representing memory address sets. Computing and maintaining memory address sets (mem-sets) at different program points (flow-sensitivity) can introduce many duplicates. The example in Figure 6 shows that the mem-set $\{o_1, o_2\}$ can appear at different program points multiple times (e.g., $\underline{\ell_1}$, $\overline{\ell_3}$). We further adopt the idea of hash consing [13, 34, 36] from the functional programming community to represent the mem-set. Each mem-set is stored in a global pool with a unique reference. Whenever a mem-set is generated during the analysis, a reference to the equivalent mem-set in the pool is returned if the mem-set already exists, otherwise, a new mem-set is added to the pool, and a reference to this newly added one is returned. We use rf_{mems} to denote the reference to the mem-set *mems*. For example, the reference of the mem-set $\{o_1, o_2\}$ is $rf_{\{o_1,o_2\}}$. Dereferencing a mem-set reference $dr(rf_{mems})$ obtains the original *mems*. For the same mem-set references $rf_{mems} \equiv rf_{mems'}$, their dereferenced mem-sets are also the same, i.e., $dr(rf_{mems}) \equiv dr(rf_{mems'})$. For equivalent mem-sets, only one of them is stored in the global pool. Let us revisit the example in Figure 6. After hash consing, we have $\delta_{\overline{\ell_3}} = \{rf_{\{o_1,o_2\}} \mapsto [1,1], rf_{\{o_3,o_4\}} \mapsto [5,5]\}$.

Implication-equivalent join operator. Before describing the rules for equivalent correlation tracking, we first outline how to merge two abstract power states (join of states) in Algorithm 1. The mem-sets in the newly produced δ are disjointed. As such, we join the abstract values $a_1 \sqcup a_2$ implied by the shared memory addresses $rf_1 \cap rf_2$ between δ_1 and δ_2 (Lines 3-5). For the relative complement parts (Lines 6-9), we build an implication from the complement mem-set to the original abstract values. The implementation of the implication-equivalent join operator is described in Section 5.1.

Implication-equivalent analysis rules. Figure 5 gives the analysis rules for implication-equivalent correlation tracking. The equivalent memory objects are grouped and propagated together for the implication-equivalent ValueFlow rule. The pointer-free rules (ConsStmt, CopyStmt, PhiStmt, UnaryStmt, BinaryStmt) remain unchanged from Figure 4. For the pointer-related rules, instead of analyzing each memory object individually, implication-equivalent objects are grouped and analyzed collectively. By grouping implication-equivalent memory objects, the analysis becomes more efficient without compromising precision.

Table 2: The statistics of the open-source projects. #LOI denotes the number of lines of LLVM instructions. #Method, #Call and #Obj are the numbers of functions, method calls and memory objects, respectively. |V| and |E| are the numbers of ICFG nodes and ICFG edges.

Project	#LOI	#Method	#Call	#Obj	|V|	|E|
paste	8,416	53	758	510	9,395	9,922
md5sum	11,483	63	881	606	12,494	13,064
YAJL	20,592	151	561	208	9,253	9,922
MP4v2	39,178	601	610	1,991	15,595	16,733
RIOT	54,597	579	1,614	951	20,176	20,843
darknet	159,205	985	9,776	2,550	136,094	147,852
tmux	446,626	1,967	22,369	3,879	162,879	178,924
Teeworlds	529,737	2,306	28,267	5,754	251,356	246,029
NanoMQ	788,967	3,235	47,646	30,838	358,312	443,670
redis	1,363,507	6,314	68,664	13,958	589,019	704,356
Total	3,422,308	16,254	181,146	61,245	1,564,573	1,791,315

5 EVALUATION

In this section, we aim to show the effectiveness of CSA for analyzing real-world programs and its practicality for bug detection, i.e., detecting buffer overflows and null dereferences. We evaluate the performance of CSA by comparing with five state-of-the-art open-source tools, Infer [38], Cppcheck [27], IKOS [16], Sparrow [49] and KLEE [17]. Moreover, we conduct an ablation analysis to gain a deeper understanding of how the cross-domain refinement and implication-equivalent approach influence the overall performance.

5.1 Datasets and Implementation

Datasets. We evaluate CSA using (1) a benchmark comprising 7774 programs from NIST [46], which includes its null dereferences and buffer overflow vulnerabilities, and (2) 10 popular open-source C/C++ projects (with their statistics in Table 2) across various application domains: `paste` [17] (file merger), `md5sum` [17] (file verifier), `YAJL` [8] (JSON parsing library), `MP4v2` [2] (MP4 file library), `RIOT` [5] (IoT operating system), `darknet`[1] (neural network framework), `tmux` [7] (terminal multiplexer), `Teeworlds` [6] (online multiplayer game), `NanoMQ` [3] (MQTT broker for IoT edge platform) and `redis` [4] (in-memory database).

Implementation. The experiments are conducted on an Ubuntu 18.04 server with an eight-core 2.60GHz Intel Xeon CPU and 128 GB memory. The interprocedural control and value flow graphs are built upon LLVM-IR (with LLVM version 14.0.0). The LLVM-IR represents program functions as global variables, further modeled as address-taken variables. The callgraph is built on-the-fly by using the cross-domain pointer information to resolve indirect calls, which precisely capture the value-flows across program procedures. Program loops/recursive calls are identified through weak topological ordering (WTO) [15], and they are handled conservatively by applying widening on the heads of the WTO. The abstract value representation is implemented using Z3 expressions [45]. For the implication-equivalent join operator, we leverage the fast set reference union/meet technique [13] when performing set union/meet. For the baselines Infer, Cppcheck, IKOS, Sparrow and KLEE, we directly use their open-source implementations and their default settings for detecting buffer overflows and null dereferences.

Table 3: Comparing with five tools and CSA-CP (a variant of CSA without cross-domain interaction, Section 5.5.1) using the NIST benchmark, with true positive rate (#TPR) and precision rate (#PCR) in percentage (%).

Tool	Buffer overflow		Null dereference		Total	
	#TPR (%)	#PCR (%)	#TPR (%)	#PCR (%)	#TPR (%)	#PCR (%)
Infer	19.23	70.57	53.17	50.19	20.20	68.48
Cppcheck	2.72	100.00	42.86	85.71	3.87	95.00
KLEE	67.78	98.81	91.27	93.12	68.45	98.58
IKOS	49.76	45.83	92.86	92.86	50.99	47.07
Sparrow	44.64	32.49	90.48	52.78	45.95	33.21
CSA-CP	73.84	42.62	100.00	42.64	74.58	42.65
CSA	**73.84**	**84.11**	**100.00**	**100.00**	**74.58**	**84.63**
BugNum	8589		252		8841	

```
int buf[5] = {0};
...
buf[idx] = 5; // idx: [0, 2]
...
int dest[5] = {0};
int c = dest[buf[4]];
```

(a) Handling arrays.

```
char buf[10];
*pa = 5; // pa -> o1
...
*pb = 10; // pb -> o2
...
send(buf[*pc]); // pc -> o1
```

(b) Handling pointers.

```
int idx;
fscanf(stdin, "%d", &idx);
int buf[10] = { 0 };
if (idx >= 0) {
    buf[idx] = 1;
}
```

(c) Handling library functions.

```
int data;
char buf[len] = "";
if (fgets(buf, len, stdin))
{
    data = atoi(buf);
}
```

(d) Handling library functions.

Figure 7: Examples extracted from NIST dataset.

5.2 Research Questions

Our evaluation aims to answer the following research questions:

- RQ1 **Is CSA effective in detecting existing bugs?** We aim to investigate whether CSA can achieve a better performance than the state-of-the-art on detecting existing bugs.
- RQ2 **Can CSA find bugs with a low false positive rate in real-world projects?** We would like to examine the effectiveness and efficiency of CSA using real-world popular applications.
- RQ3 **What is the influence of different components in our framework?** We aim to understand RQ3.1: the precision improvement of cross-domain refinement; and RQ3.2: efficiency improvement in terms of time and memory using equivalent correlation tracking.

5.3 NIST Benchmark (RQ1)

Table 3 shows the results of the true and false positives of CSA and our five baseline detectors on detecting buffer overflows [28–30] and null dereferences [31] from the pre-labeled NIST benchmark. The last row of the table displays the number of labeled bug ground truth for each category.

Comparison results. Overall, CSA outperforms the static tools Infer, IKOS and Sparrow with an average 35.04% higher precision and a 35.53% higher true positive rate. Cppcheck has a higher precision than CSA but detects only 3.87% of bugs, while CSA finds over 19 times more bugs (74.58%). KLEE, a dynamic tool, generates few false alarms, but CSA detects more bugs than KLEE.

```
char data[100];
memset(data, 'A', 50-1);
data[50-1] = '\0';
len = strlen(data);
char dest[50] = "";
for (i = 0; i < len; i++)
    dest[i] = data[i];
```

(a) Memory operation functions.

```
int arr[8];
...
int s = input();
int idx = 0
if(s == 5) idx = idx + 5;
if(s == 3) idx = idx + 3;
arr[idx] = 1;
```

(b) Branch correlations.

Figure 8: False positives reported by CSA.

Result analysis. We demonstrate several code scenarios in Figure 7 to explain the better performance of CSA. The primary reason is that CSA conducts pointer and interval analyses collaboratively. Figure 7(a) shows an example, similar to the one in Figure 1. The array access at ℓ_6 is safe because the offset `buf[4]` does not exceed the size of the array `dest`. CSA can precisely determine the interval value of `buf[4]` as the initial value $[0, 0]$ stored at ℓ_1, not affected by the value $[5, 5]$ stored at ℓ_3. This is because CSA removes the spurious data dependence $\ell_3 \xhookrightarrow{\&buf[4]} \ell_6$, preventing $[5, 5]$ at ℓ_3 from propagating to ℓ_6. Our baselines report a false alarm here, indicating that `buf[4]` is of value $[0, 5]$ because of the inaccuracy of the pre-pointer analysis of the array. Figure 7(b) presents another example where the constant 5 is stored in the location pointed by `pa` at ℓ_2, and the constant 10 is stored in `pb` at ℓ_4. At ℓ_6, the array `buf` of size 10 is accessed using the dereferenced value of `pc`. CSA can precisely distinguish that only `pc` and `pa` are aliased (both point to o_1). Therefore, the dereferenced value of `pc` is $[5, 5]$. However, the offline pointer analysis used in our baselines considers `pc` to be aliased with both `pa` and `pb`. As a result, the dereferenced value of `pc` is $[5, 10]$, leading to a false positive at ℓ_6.

Moreover, CSA can handle the side effects of standard library functions (e.g., `fscanf`, `fgets`, `snprintf`) precisely. For example, in Figure 7(c), the variable `idx` is used as an offset to write to the fixed-length array `buf` at ℓ_5. However, the value of `idx` is specified by the external I/O function `fscanf` at ℓ_2, causing a potential buffer overflow vulnerability. To address this issue, CSA stores an unbounded value in `idx`, indicating that `idx` can represent any integers. This helps CSA report the buffer overflow warning at ℓ_5 because the value of `idx` can be larger than that of `buf`. In comparison, all our baselines except KLEE overlook this vulnerability. Another example is shown in Figure 7(d), where the size of the array `buf` at ℓ_2 is determined by the variable `len`. At ℓ_3, the program calls `fgets` to fill `buf` with user inputs, and the second parameter of `fgets` specifies the maximum input size. CSA can prove the safe usage of the `fgets` function because the size of the inputs is bounded by the array size `len`. However, all our baseline detectors report a false alarm here in `fgets` function calls.

False positives of CSA. We further inspect the false positives reported by CSA and identified two reasons behind them, as shown in examples in Figure 8. Firstly, CSA trades precision for efficiency when handling some string-related built-in functions, such as string length calculation function `strlen` in Figure 8(a). Instead of iterating through each field to locate the '\0' stored in `data[50 − 1]` at ℓ_3, CSA returns the allocated memory size of `data`, which is 100. As a result, the value of `len` is over-approximated, causing a false positive at ℓ_7. This limitation can be overcome by developing efficient string analysis with better approximations. False positives

Table 4: Comparing CSA with five open-source tools and CSA-CP using ten popular applications. #TP and #FP are true positive and false positive, respectively. Time (secs), Mem (MB) are running time and memory costs. The – in the Time columns indicates a running time of more than 4h. The – in the Mem columns indicates a cost of more than 100 Gigabytes.

Project	Infer				Cppcheck				IKOS				KLEE				Sparrow				CSA-CP				CSA			
	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem	Report		Time	Mem
	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)	#TP	#FP	(secs)	(MB)
paste	1	15	7	61	0	17	1	9	3	21	512	1126	4	0	2911	1711	4	35	3	51	3	19	5	92	3	0	9	106
md5sum	2	21	8	80	0	18	1	11	2	35	986	1684	3	0	2824	1642	2	22	2	48	4	26	15	121	4	1	8	110
YAJL	0	17	9	110	0	14	1	12	1	1625	2895	4822	4	16	14400	17333	3	86	6	59	3	35	7	172	3	0	5	102
MP4v2	1	28	313	335	1	26	38	38	1	956	3684	6215	2	3	14400	21358	1	236	214	231	1	25	58	269	1	0	13	384
RIOT	3	29	111	155	2	19	2	22	2	1325	5216	8622	5	2	14400	23654	2	651	315	421	8	38	102	366	8	6	27	346
darknet	25	134	837	282	16	214	10	55	14	1265	9531	23954	25	8	14400	40015	10	842	826	984	21	199	3483	1982	21	10	3507	1875
tmux	5	142	522	909	3	156	30	39	4	1632	11325	38366	2	1	14400	70826	3	1256	1036	1894	12	360	1182	6343	12	10	824	5052
Teeworlds	10	169	684	934	4	187	2	54	2	529	13569	40368	2	1	14400	71865	10	1512	1593	2984	15	244	2754	3485	15	8	2886	2598
NanoMQ	23	154	654	305	10	147	94	38	–	–	–	–	5	2	14400	91465	6	1241	1642	3125	30	292	1801	7063	30	8	1143	6551
redis	6	137	1292	10484	8	136	516	123	–	–	–	–	3	2	14400	101475	5	1152	2654	9211	14	275	8629	4421	14	8	6553	3870
Total	76	846	4437	13655	44	934	695	401	29	7388	47718	125157	55	35	120935	441344	46	7033	8291	19008	111	1513	18036	24314	111	51	14975	20994

```
void process_msg() {
    PNode *hd = get_node_arr();
    PNode *n1 = hd + inc1;
    recv(n1->len, ...);
    ...
    PNode *n2 = n1 + step;
    enquiry_msg_queue(n2);
}
void enquiry_msg_queue(PNode *n) {
    memcpy(sh->buf, ptr, n->len);
    ...
}
```

(a) A safe buffer access in NanoMQ.

```
void CSound::RateConvert(int SampleID) {
    CSample *pSample = &m_aSamples[SampleID];
    ...
    int NumFrames = ...;
    short *pNewData = (short *) mem_alloc(...);
    ...
    for(int i = 0; i < NumFrames; i++)
        ...
        if(pSample->m_Channels == 1)
            pNewData[i] = pSample->m_pData[f];
    ...
}
```

(b) A null dereference bug in Teeworlds.

```
void valid_captcha(..., char *f) {
    char **labels = get_labels(f);
    network *net = load_network(...);
    list *plist = get_paths(...);
    ...
    for(int i = 0; I< plist->size; ++i)
        ...
        for(int j = 0; j < 13; ++j)
            ...
            if(strstr(...,labels[j]))
    ...
}
```

(c) A buffer overflow bug in darknet.

Figure 9: A false positive scenario eliminated by CSA and two bugs found by CSA in real-world projects.

also occur due to untracked branch correlations as CSA's analysis is path-insensitive over a non-relational domain. In Figure 8(b), the branch conditions at ℓ_5 and ℓ_6 contradict each other. However, CSA's interval analysis fails to distinguish between the conflicting conditions. As a result, the additions at ℓ_5 and ℓ_6 are executed, causing `idx` to accumulate to 8, surpassing the array's size. Introducing relational domains in the abstract states can eliminate this type of false positive.

5.4 Bugs in Real-World Projects (RQ2)

Table 4 presents the experimental results for true positives, false positives, running time, and memory costs of CSA compared to the five existing tools (Infer, Cppcheck, IKOS, KLEE and Sparrow) on real-world popular applications. Overall, CSA and the other five tools have reported 3673 bugs. We successfully identified 281 real bugs after a rigorous manual examination of its bug report. Out of these bugs, 62 have been fixed by the developers.

Comparison results and analysis. CSA achieved the best result by correctly identifying 111 real bugs with the highest precision of 68.51%. In comparison, the baseline detectors missed more than 82.21% of the real bugs. For example, CSA detects 46.05% more bugs than Infer. Additionally, on average, the baseline detectors (Infer, Cppcheck, IKOS and Sparrow) exhibit a precision rate of only 14.98%, less than one-fourth of the precision rate of CSA. CSA even achieves 12.11% more precision rate than KLEE.

CSA detects more bugs because it effectively handles hard code features, such as interprocedural analysis, loop handling, and accurate external API modeling. This gives it an edge over other tools in uncovering more bugs. The key factor behind CSA's lower false positive rate lies in the precision improvement achieved through cross-domain refinement (Section 5.5.1). In terms of efficiency, IKOS demands high computational resources and fails to finish the analysis within a four-hour limit when analyzing `NanoMQ` and `redis`. The reason lies in its non-sparse solution, as it maintains and accumulates abstract states throughout the control flow graph. Although KLEE records fewer false alarms, it discovers fewer bugs and incurs much higher computational overhead as it analyzes all possible paths through a program symbolically with constraint solving. In contrast, CSA statically approximates all possible runtime states, resulting in a much better analysis coverage. CSA may require more time for code analysis than Infer, Sparrow and Cppcheck but discovers numerous real bugs with a significantly higher precision. As a result, we believe CSA lies at the sweet spot between scalability and precision, considering the benefits of its bug detection outweigh the time overhead it incurs during analysis compared to Infer, Sparrow and Cppcheck.

Case study. Figure 9 presents three real-world code snippets to illustrate the effectiveness of CSA in finding bugs in diverse software projects. We only show the essential parts relevant to the vulnerability for illustration purposes. In Figure 9(a), we examine a code segment extracted from the `NanoMQ` project, wherein CSA successfully eliminates a false buffer overflow alarm. At ℓ_{10}, the function `enquiry_msg_queue` invokes the library function `memcpy` to copy `n->len` bytes of data from `ptr` to `sh->buf`. The variable `n->len` does not exceed the length of `sh->buf` and `ptr`; therefore, no buffer overflows occur at ℓ_{10}. However, Infer, Cppcheck and

Table 5: Comparison between CSA and CSA-NI (a version of CSA without implication-equivalent memory addresses).

Project	CSA-NI		CSA	
	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)
`tmux`	1540 (1.87×)	21016 (4.16×)	824	5052
`Teeworlds`	6176 (2.14×)	14237 (5.48×)	2886	2598
`NanoMQ`	3292 (2.88×)	48805 (7.45×)	1143	6551
`redis`	21232 (3.24×)	32314 (8.35×)	6553	3870
Geo. Mean	(2.47×)	(6.14×)		

Sparrow report a false alarm here because they treat `n2` aliased with `n1`, wherein `n1->len` holds an unbounded value at label ℓ_4. The imprecise alias relation between `n1` and `n2` is due to the external pointer analysis's inability to determine the values of the offset variable `inc1` at ℓ_3 and `step` at ℓ_6, making `n1` and `n2` both conservatively point to all possible objects. Figure 9(b) shows a null dereference found by CSA in `Teeworlds`, which is missed by Cppcheck. The pointer `pNewData` at ℓ_5 could be null and is dereferenced at ℓ_{10}. There are no null pointer checkings for ℓ_{10}. Figure 9(c) presents a buffer overflow bug found by CSA in `darknet` while KLEE fails to find this bug. The buffer `label` is initialized at ℓ_2 using a file reading function `get_labels`. The buffer size is an unbounded value depending on the input file `f`. At ℓ_{10}, the buffer `label` is accessed using the variable `j`, whose value ranges from [0, 12] indicated by the loop guard at ℓ_8. However, the buffer size can be less than 12, which may cause a buffer overflow.

5.5 Ablation Analysis (RQ3)

5.5.1 Precision Improvement of Cross-Domain Refinement (RQ3.1). For the baseline sparse analysis that performs pointer analysis and interval analysis without cross-domain refinement, we employ sparse value-flow representation in SVF [60] and implement the analysis over combined memory address and interval domains using the Cartesian product (Section 2.1). In our evaluation, we use CSA-CP to refer to this baseline sparse analysis. When comparing with CSA-CP on the NIST dataset, as shown in Table 3, both tools have the same true positive rate (74.58%). However, CSA-CP records 82.32% more false positives due to its less precise external pointer analysis and lack of domain refinement. For the real-world projects, as shown in Table 4, although CSA-CP also found 111 real bugs given the over-approximation of its pointer analysis, it reported a much higher number of false positives (1513) when compared to CSA (51) due to the overly conservative points-to results of CSA-CP's isolated pointer analysis.

5.5.2 Efficiency Improvement of Equivalent Correlation Tracking (RQ3.2). Table 5 shows the comparison results between CSA and CSA without implication-equivalent memory address (CSA-NI) on four open-source projects (`tmux`, `Teeworlds`, `NanoMQ`, and `redis`). The true and false positives of CSA and CSA-NI are identical, suggesting that the implication-equivalent approach is precision-preserving for detecting bugs. On average, equivalent correlation tracking results in a speedup of approximately 2.47×, with up to 3.24× on the `redis` project. CSA-NI also uses an average of around 6.14× more memory than CSA, with a maximum of 8.35× on the `redis` project. These results demonstrate the improvement brought by our implication-equivalent memory address sets regarding analysis time and memory usage.

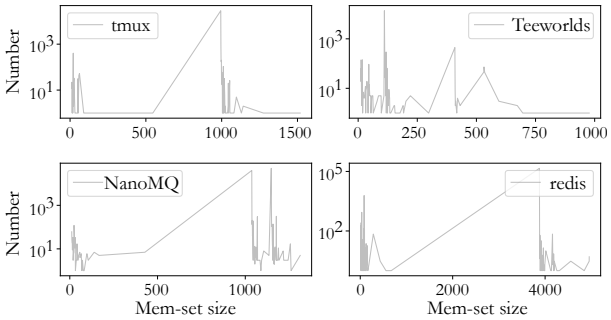


Figure 10: Distribution of mem-set sizes.

We aim to investigate further the grouping of memory address sets to better understand the benefits of implication-equivalent memory address sets. Therefore, we explore the distribution of mem-set (memory address set) sizes collected in the abstract states. Figure 10 illustrates the distribution statistics, revealing that numerous pointers can point to a large number of objects. Consider the example of `redis`, where the average size of its memory address sets is 799.22, with the largest set containing 4955 elements. This indicates that a considerable number of objects can be grouped and analyzed together. The implication-equivalent approach eliminates the redundant value join and copy of the memory addresses in the same group. Furthermore, the analysis reveals an interesting observation that the speedup and memory savings brought by implication-equivalent memory address sets are positively correlated with the size of the memory address sets. To illustrate, `redis` shows the most significant performance improvement, with a speedup of 3.24×, compared to merely 1.87× for `tmux`. Additionally, `redis` demonstrates a memory saving of 8.35×, compared to only 4.16× for `tmux`. These results align with the mem-set size statistics: `redis` has an average mem-set size of 799.22, while `tmux` has only 163.71 on average.

6 RELATED WORK

Our discussion focuses on the aspects of the work that are closely related to CSA, specifically, the utilization of abstract execution, combined abstract domain and sparse static analysis.

Abstract Execution. Abstract execution or interpretation [22, 25] statically reasons about program runtime states. A wide range of static analysis approaches is built upon abstract interpretation, e.g., constant propagation [16, 33], booleans [61, 62], intervals [16, 23, 50, 51], dual numbers [40] and relational domains [16, 56, 57, 64, 67, 69], to ensure over-approximation and analysis termination. These approaches conduct pointer analysis and abstract execution independently and assume that all elements of an array were regarded as the alias of each other (array-insensitive) [56, 57, 61]. Loops and recursive data structures/calls are unrolled a fixed number of times, which may under-approximate the runtime behavior of the program, while we use WTO [15] and widening to over-approximate the abstract states within program loops. Weiss et al. [66] propose a database-backed analysis based on graph algorithms, which is orthogonal to our approach.

Combined Abstract Domain. A variety of abstract domains have been developed over these years, e.g., Powerset domains [11], Interval [22], Polyhedral [26], Octagon [44, 58], which can be combined to fit specific analyses. For example, ASTRÉE [23] uses combined domains to improve analysis precision while guaranteeing soundness. A straightforward combination is a Cartesian product [9, 43], where the combined concretization may not be injective. A generic refinement method is a lower closure operator [35] on the lattices of multiple abstract domains to refine the abstract values, e.g., reduced product [21, 24], which aims to obtain a more precise abstraction concerning the Cartesian product's abstract values without sacrificing the abstraction's soundness. Another refinement method is the reduced cardinal power [22] which takes into account relations between different domains. Like the reduced product, its primary objective is to refine the value in the lattice of the Cartesian product domain [21], while CSA is the first to introduce reduced cardinal power in sparse abstract execution for efficient correlation tracking between the memory address and interval domains.

Sparse Static Analysis. Sparse program analysis prevents the expensive data-flow propagation across the control flow graph. It greatly relies on the construction of data dependencies. Static single assignment [32] explicitly captures the def-use chains and provides an effective representation for the data dependence analysis, thereby facilitating sparse program analysis. Sparse program analysis can provide speedup for a wide range of applications, such as constant propagation [53, 65], pointer analysis [14, 37, 63, 68], bug detection [20, 57, 61, 62] and code embedding [18, 19, 59]. Madsen and Møller [42] proposed a special sparse analysis for JavaScript programs. Oh et al. [47, 48] are the first to propose sparse abstract interpretation. However, the above sparse analyses are all based on a pre-pointer analysis over individual domains, while our analysis is more precise with cross-domain interactions.

7 CONCLUSION

This paper introduces CSA a new sparse abstract execution approach that works on multiple abstract domains through correlation tracking. CSA performs online refinement during sparse analysis and achieves more precise analysis results than the traditional sparse analysis that handles each domain separately. CSA outperforms five open-source tools on the NIST benchmark set and ten open-source projects in two assertion-based checking client applications, buffer overflow and null dereference detection. In the real-world projects, CSA identified 111 bugs with a precision rate of 68.51%, surpassing Infer by detecting 46.05% more bugs and outperforming KLEE with a 12.11% higher precision rate. When compared to the version lacking cross-domain interaction, CSA decreases false positives by 82.32% on NIST dataset and by 96.63% on real-world projects. Additionally, CSA exhibited an average speedup of 2.47× and reduced memory usage by 6.14× when utilizing equivalent correlation tracking.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their reviews and suggestions. This research is supported by Australian Research Grants DP210101348 and FT220100391, and by a generous Aspire Gift Grant from Google.

REFERENCES

[1] 2023. Darknet - Open Source Neural Networks in C. https://github.com/pjreddie/darknet
[2] 2023. MP4v2 - A C/C++ library to create, modify and read MP4 files. https://github.com/enzo1982/mp4v2/
[3] 2023. NanoMQ - An ultra-lightweight and blazing-fast MQTT broker for IoT edge. https://github.com/emqx/nanomq
[4] 2023. Redis - The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker. https://github.com/redis/redis/
[5] 2023. RIOT - The friendly OS for IoT. https://github.com/RIOT-OS/RIOT
[6] 2023. Teeworlds - A retro multiplayer shooter. https://teeworlds.com/
[7] 2023. Tmux - tmux source code. https://github.com/tmux/tmux
[8] 2023. YAJL - A fast streaming JSON parsing library in C. https://github.com/lloyd/yajl
[9] Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2020. Abstract interpretation, symbolic execution and constraints. In *Recent Developments in the Design and Implementation of Programming Languages*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[10] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen* (1994).
[11] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2006. Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer* 8, 4 (01 Aug 2006), 449–466.
[12] George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *SAS '16*.
[13] Mohamad Barbar and Yulei Sui. 2021. Hash Consed Points-To Sets. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 25–48.
[14] 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 '21)*. IEEE Computer Society, USA, 222–235.
[15] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–141.
[16] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. 2014. IKOS: A Framework for Static Analysis Based on Abstract Interpretation. In *Software Engineering and Formal Methods*, Dimitra Giannakopoulou and Gwen Salaün (Eds.). Springer International Publishing, Cham, 271–277.
[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 209–224.
[18] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 38 (2021), 33 pages.
[19] Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-Sensitive Code Embedding via Contrastive Learning for Software Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. ACM.
[20] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. 2007. Practical Memory Leak Detection Using Guarded Value-Flow Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery.
[21] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. 2013. A survey on product operators in abstract interpretation. *arXiv preprint arXiv:1309.5146* (2013).
[22] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252.
[23] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2007. Combination of Abstractions in the ASTRÉE Static Analyzer. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, Mitsu Okada and Ichiro Satoh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–300.
[24] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2011. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *Foundations of Software Science and Computational Structures*, Martin Hofmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 456–472.
[25] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: Abstract^2 Interpretation. *Proc. ACM Program. Lang.* 3, POPL, Article 42 (jan 2019), 31 pages.
[26] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. Association for Computing Machinery, New York, NY, USA, 84–96.

[27] Cppcheck. 2021. Cppcheck: A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/.

[28] CWE-121 2023. CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html.

[29] CWE-122 2023. CWE-122: Heap-based Buffer Overflow. https://cwe.mitre.org/data/definitions/122.html.

[30] CWE-126 2023. CWE-126: Buffer Over-read. https://cwe.mitre.org/data/definitions/126.html.

[31] CWE-476 2023. CWE-476: NULL Pointer Dereference. https://cwe.mitre.org/data/definitions/476.html.

[32] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (1991), 451–490.

[33] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 57–68.

[34] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-Safe Modular Hash-Consing. In *Proceedings of the 2006 Workshop on ML* (Portland, Oregon, USA) *(ML '06)*. Association for Computing Machinery, New York, NY, USA, 12–19.

[35] Roberto Giacobazzi and Francesco Ranzato. 1997. Refining and compressing abstract domains. In *Automata, Languages and Programming*, Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 771–781.

[36] Jean Goubault. 1994. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. *Journées Francophones des Langages Applicatifs (JFLA'93)* (1994), 222–238.

[37] B. Hardekopf and C. Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. *CGO '11* (2011), 289–298.

[38] Infer. 2021. Facebook Infer: a tool to detect bugs in Java and C/C++/Objective-C code. https://fbinfer.com/.

[39] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86.

[40] Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. 2022. A Dual Number Abstraction for Static Analysis of Clarke Jacobians. *Proc. ACM Program. Lang.* 6, POPL, Article 56 (jan 2022), 30 pages.

[41] Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 3–16.

[42] Magnus Madsen and Anders Møller. 2014. Sparse Dataflow Analysis with Pointers and Reachability. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 201–218.

[43] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems*, Mooly Sagiv (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–20.

[44] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (01 Mar 2006), 31–100.

[45] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[46] NIST 2023. NIST datasets. https://samate.nist.gov/SARD/test-suites/116.

[47] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global Sparse Analysis Framework. *ACM Trans. Program. Lang. Syst.* 36, 3, Article 8 (sep 2014), 44 pages.

[48] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 229–238.

[49] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. The Sparrow static analyzer. https://opam.ocaml.org/packages/sparrow/.

[50] Komal Pathade and Uday P. Khedker. 2018. Computing Partially Path-Sensitive MFP Solutions in Data Flow Analyses. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 37–47.

[51] Komal Pathade and Uday P. Khedker. 2019. Path Sensitive MFP Solutions in Presence of Intersecting Infeasible Control Flow Path Segments. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 159–169.

[52] D.J. Pearce, P.H.J. Kelly, and C. Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM TOPLAS* 30, 1 (2007), 4–es.

[53] John H. Reif and Harry R. Lewis. 1977. Symbolic Evaluation and the Global Value Graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 104–118.

[54] P. Cousot. 2005. Abstract interpretation. (Feb.–May 2005). MIT course 16.399, http://web.mit.edu/16.399/www/.

[55] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

[56] 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. *SIGPLAN Not.* 53, 4 (jun 2018), 693–706.

[57] Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2021. Path-Sensitive Sparse Analysis without Path Conditions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 930–943.

[58] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2015. Making Numerical Program Analysis Fast. *SIGPLAN Not.* 50, 6 (jun 2015), 303–313.

[59] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 233 (Nov. 2020), 27 pages.

[60] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC)*. ACM, New York, NY, USA, 265–266.

[61] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 254–264.

[62] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Software Eng (TSE '14)*. 40, 2 (2014), 107–122.

[63] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA. *Programming Languages and Systems (APLAS '11)* (2011), 155–171.

[64] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. 2023. Detecting Blocking Errors in Go Programs Using Localized Abstract Interpretation. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 32, 12 pages.

[65] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210.

[66] Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. 2015. Database-Backed Program Analysis for Scalable Error Propagation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 586–597.

[67] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program Analysis via Efficient Symbolic Abstraction. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 118 (oct 2021), 32 pages.

[68] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 218–229.

[69] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. 2019. Grapple: A Graph System for Static Finite-State Property Checking of Large-Scale Systems Code. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 38, 17 pages.