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.

CCS CONCEPTS
- Software and its engineering → Automated static analysis;
- Theory of computation → Abstraction.

KEYWORDS

Abstract execution, sparse analysis, cross-domain interaction

ACM Reference Format:

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 [57, 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,
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 ℓ2, it aims to print an item tab[1*loc2] at ℓ4 through a location index 1*loc2 retrieved from an index array ids. Despite the table access at ℓ4 being safe, both SPARROW [49] and INFERN [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 ℓ4 gets propagated to *loc2 at ℓ4 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 ℓ4 only points to ids’s first four elements (&ids[0]...&ids[3]) given constraint randn()%4 at ℓ5, while loc2 at ℓ4 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 ℓ1.

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 sqlot. For example, SPARROW [49] aggregates all memory addresses of ids into a single abstract location when performing pointer analysis. Without the interval result itv(slots) = [0, 3] at ℓ4, 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 ℓ7 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 itv(slots) from the interval domain. As a result, loc1 and loc2 are precisely identified as must-not-aliases, eliminating the spurious def-use chain between ℓ5 and ℓ7. Meanwhile, the pointer analysis on the memory address domain refines the interval value itv(*loc2) at ℓ7. Thus, at ℓ7, *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].
Precise Sparse Abstract Execution via Cross-Domain Interaction

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

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 a 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 \( A \) are abstract values that approximate a set of concrete values, i.e., the values that a variable can take in the concrete domain \( C \) during program execution (e.g., integers, floats and strings). The abstract domain \( A \) is an over-approximated abstraction of \( C \) with a concretization function \( y \in A \rightarrow C \) based on a partial order \( \preceq \) over \( A \) such that \( \forall a, a' \in A, a \preceq a' \iff y(a) \subseteq y(a') \). The partial order relations of an abstract domain \( A \) form a lattice \( \mathcal{L} = (A, \preceq, \perp, \top) \), where \( \perp \) and \( \top \) are the meet and join operations, and \( \perp \) and \( \top \) are unique least and greatest elements of \( A \).

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 \( \times \) (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 \( (a_1 \times \ldots \times a_n) \) 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 \( (a_1, \ldots, a_n) \) be an abstract value in \( A_1 \times \ldots \times A_n \), and \( C_1 = y_1(a_1), \ldots, C_n = y_n(a_n) \) be their corresponding concrete values. The concretized value of \( (a_1, \ldots, a_n) \) is the intersection of \( C_1 \) to \( C_n \), i.e., \( y((a_1, \ldots, a_n)) = C_1 \cap \cdots \cap C_n \).

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 \( V \) is separated into two subsets, \( O \) that contains all possible abstract objects, i.e., address-taken variables of a pointer and \( P \) that contains all top-level variables. In LLVM’s language, a top-level variable \( p, q, r \in P \), including stack virtual registers and global variables, can only be defined once. An address-taken object \( o \in O \) can be read/modified only through dereferencing top-level pointers at \( \text{LoadStmt}/\text{StoreStmt} \). A constant value \( c \) is always first assigned to a top-level variable at \( \text{ConsStmt} \). For \( \text{AddrStmt} \), \( p = \text{alloca} \) is a stack or global variable or a dynamically created abstract heap object. \( \text{GepStmt} \) models the field accesses of a struct object with its field offset \( f1d \) 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., \( p = \&a[v] \)) is resolved during abstract execution. \( \text{PhiStmt} \) 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 \( \text{CopyStmt} \).
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 $\sigma$ (Definition 3). At a high level, abstract execution follows the control flow graph and updates $\sigma$ by analyzing each statement until a fixed point is reached.

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

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

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 flow 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 $\ell \rightarrow \ell'$ to represent the value-flow of an address-taken object $o \in O$ which is defined at $\ell$ and used at $\ell'$.

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 $\mathbb{Z}$ represents a set of integers that fall between two given endpoints, which is equipped with a lattice $\text{Interval}(\mathbb{Z})$. The lattice for memory addresses is a set of discrete values. The lattice for MemAddress domain is $\langle \mathbb{O} \cup \mathbb{Q} \cup \mathbb{S} \cup \mathbb{N} \cup \mathbb{D} \cup \mathbb{M} \cup \mathbb{T} \rangle$, where $\mathbb{O} = \mathbb{P}(O)$ 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 ($\sigma = \mathbb{L} \times \mathcal{V}$) without cross-domain refinement. As shown in Figure 3(a), the abstract value of the pointer variable $\ell_{oc1}$ at program point $\ell_1$ is represented by a pair $\sigma_{\ell_{oc1}}(\ell_1) = (\mathbb{I}, \{\ell_{ids}[0], \ell_{ids}[1], ..., \ell_{ids}[4]\})$ (some approaches [38, 49] aggregate $\ell_{ids}[0], \ell_{ids}[1], ..., \ell_{ids}[4]$ to one object) where the first element $\mathbb{I}$ is a conservative value of this pointer by interval analysis. The second element is a set of memory addresses pointed by $\ell_{oc1}$ computed by an external pointer analysis over the memory address domain. Because the pointer analysis does not have interval information, $\ell_{oc1}$ conservatively points to all the elements of $\ell_{ids}$ including a spurious target $\ell_{ids}[4]$, which establishes a false data dependence $\ell_{ids}[4] \rightarrow \ell_{loc2}$. This false dependence propagates value $\ell_{ids}[4]$ to $\ell_{loc2}$ at $\ell_2$ ($\ell_2$ here represents a conservative points-to result as a constant integer can be interpreted as a memory address value in LLVM). Consequently, $\ell_{oc1}$ 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 $\sigma$ only to maintain abstract trace for top-level variables $\mathcal{P}$ (Section 2.2). The trace is then reduced to $\sigma = \mathcal{P} \rightarrow \text{Interval} \times \text{MemAddress}$. To keep track of the values of memory objects $O$, the analysis needs to capture the correlations across abstract domains between $\text{MemAddress}$ and $\text{Intervals}$ 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 $\ell_1$ in Figure 3(b), we create an implication from each memory object in $\ell_{ids}$ to the initial value, that is $\ell_{ids}[0] \rightarrow \mathbb{I} \mathbb{O} \mathbb{N} \mathbb{Q} \mathbb{D} \mathbb{M} \mathbb{T}$, and derives a precise memory address value of $\ell_{oc1}, pts(\ell_{oc1}) = \{\ell_{ids}[0], \ell_{ids}[1], ..., \ell_{ids}[3]\}$. This precise points-to result in turn helps remove the spurious dependence $\ell_{ids}[4] \rightarrow \ell_{loc2}$. Consequently, it contributes to a more precise interval value $\langle [0, 0], \mathbb{T} \rangle$. 

Figure 3: Sparse analysis on the memory address and interval domains by revisiting the example in Figure 1.
We present precise correlation tracking through domain interaction objects within arrays, the implications in $\delta$ unlike existing analyzers [37, 49, 60] that aggregate the memory use pre-computed points-to results without online refinement.

The implication result of $\ell$ represents a set of implications at program point $L$. We use $\phi[p1, p2, \ldots, pn]$ to represent the abstract operator concerning the concrete operator $f$. $\gamma(p)$ concretely the abstract value $\sigma(p)$ based on Definition 1 and returns the memory address set pointed by $p$ or the integers residing inside the interval value of variable $p$. $a.fld_j$ represents the $j^{th}$ field of the base object $a$.

of $+\text{loc}2$ at $f_1$. As a result, $+\text{loc}2$ can only have the value $[0, 0]$ copied from $\&\text{id}[4]$ at $f_1$ via value-flow $f_1 \xrightarrow{\&\text{id}[4]} f_2$. Hence the table access $\&\text{tab}[+\text{loc}2]$ 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 $a_1 \rightarrow a_2$ in reduced cardinal power maps $a_1 \in A_1$ to $a_2 \in A_2$, where $A_1$ is MemAddress and $A_2$ is Interval x 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 $a_1 \rightarrow a_2$.

In our analysis, $A_1$ 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 $A_2$ represents Interval x 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 $\delta : L \rightarrow (A_1 \rightarrow A_2)$). We define an abstract power trace $\delta$ to track the implications at each program point. We use $\delta_f(L) = \{a_1 \rightarrow a_2 | a_1 \in A_1 \land a_2 \in A_2\}$ to represent a set of implications at program point $L$. $\delta_f(a_1)$ 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 $\delta_f$ 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 (CONSSTM, COPYSTM, PHISTM, UNARYSTM, BINARYSTM), we update the value of the left-hand-side variable based on the right-hand-side value and relevant operators. For instance, given a BINARYSTM $f : r = p + q$ and $\sigma(p) = \{[1, 5], T\}, \sigma(q) = \{[4, 5], T\}$, we derive $\sigma(r) = \{[5, 10], T\}$. We show pointer-related analysis rules, in particular, GEPSTM, LOADSTM and STORESTM, to conduct online value-flow refinement by capturing correlation between domains.

The GEPSTM rule updates the value in the memory address domain with the information from the interval domain. For GEPSTM 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 LOADSTM $f : p = q$, we refine the derived value of $p$ based on the precise memory addresses of $q$ and the correlation from the abstract power trace $\delta_f$. We first obtain the memory addresses of $q$ via $\sigma(q)$ from the memory address domain. For each memory address $\sigma(q)$ that $r$ refers to, we retrieve the interval value or memory addresses related to $\sigma$ based on the implications stored in $\delta_f$. The final result for $\sigma(p)$ is the join of all implied values of the memory addresses.

For a STORESTM $f : r = q$, we construct the implication based on $\sigma(p)$ and $\sigma(q)$. We first obtain $p$’s points-to targets via $\sigma(p)$ from the memory address domain. For each memory address $\sigma(q)$ that $p$ refers to, we map $o$ to the abstract value of $q$ in the power state, i.e., $o \mapsto \sigma(q)$. 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].

Figure 4: Analysis rules for sparse abstract execution by tracking correlations across domains. $a \sqsupseteq b$ represents $a = a \cup b$. $f^X$ is the abstract operator concerning the concrete operator $f$. $\gamma(p)$ concretely the abstract value $\sigma(p)$ based on Definition 1 and returns the memory address set pointed by $p$ or the integers residing inside the interval value of variable $p$. $a.fld_j$ represents the $j^{th}$ field of the base object $a$. 
Figure 5: Analysis rules for sparse abstract execution with equivalent correlation tracking across domains. The rules for pointer-free statements (ConstStmt, 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., $\delta_{l_1}(o) \sqcup \delta_{l_2}(o)$. 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 $\langle 1, 1, T \rangle$ at the StoreStmt $l_1$. Consequently, analyzing $o_1$ and $o_2$ requires only a single implication statement: $\langle o_1, o_2 \rangle \mapsto \langle 1, 1, T \rangle$. 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.
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., \( f_1, f_2 \)). 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 \( \delta \) to denote the reference to the mem-set \( \text{mems} \).

For example, the reference of the mem-set \( \{ o_1, o_2 \} \) is \( \delta \). Dereferencing a mem-set reference \( \delta \) obtains the original \( \text{mems} \). For the same mem-set references \( \delta \), their dereferenced mem-sets are also the same, i.e., \( \delta \equiv \delta \). 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 \( \delta \equiv \{ o_1, o_2 \} \rightarrow \{ [1, 1], [o_1, o_2] \} \rightarrow \{ 5, 5 \} \).

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 \( \delta \) are disjoint. As such, we join the abstract values \( a_1 \sqcup a_2 \) implied by the shared memory addresses \( \delta f_1 \sqcap \delta f_2 \) (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 (CONSSTM, COPYSTM, PHISTM, UNARYSTM, BINARYSTM) 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.

| Project     | #LOI | #Method | #Call | #Obj | |V| | |E| |
|-------------|------|---------|-------|------|-------|-------|
| paste       | 8,416| 53      | 758   | 510  | 9,395 | 9,922 |
| ndskmm      | 11,483| 63     | 881   | 606  | 12,494| 13,064|
| YAJL        | 20,592| 151    | 561   | 208  | 8,253 | 9,922 |
| MP4vz       | 39,178| 9,395  | 3,235 | 1,991| 11,483| 443,670|
| RIOT        | 54,597| 606    | 1,614 | 951  | 13,064| 22,369|
| darknet     | 159,208| 985   | 9,776 | 2,550| 138,094| 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| 1,364,573| 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


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 (%).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Buffer overflow</th>
<th>Null dereference</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#TPR (%)</td>
<td>#PCR (%)</td>
<td>#TPR (%)</td>
</tr>
<tr>
<td>INFER</td>
<td>19.23</td>
<td>70.57</td>
<td>20.20</td>
</tr>
<tr>
<td>Cppcheck</td>
<td>2.72</td>
<td>100.00</td>
<td>20.20</td>
</tr>
<tr>
<td>KLEE</td>
<td>67.78</td>
<td>98.81</td>
<td>68.45</td>
</tr>
<tr>
<td>IKOS</td>
<td>49.76</td>
<td>43.83</td>
<td>47.07</td>
</tr>
<tr>
<td>Sparrow</td>
<td>44.64</td>
<td>32.49</td>
<td>45.95</td>
</tr>
<tr>
<td>CSA-CP</td>
<td>73.84</td>
<td>42.62</td>
<td>74.58</td>
</tr>
<tr>
<td>CSA</td>
<td>73.84</td>
<td>84.11</td>
<td>84.63</td>
</tr>
</tbody>
</table>

| BugNum     | 8589            | 252             | 8841           |

(a) Handling arrays. (b) Handling pointers.

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.

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 $f_5$ is safe because the offset buf[4] does not exceed the size of the array dest. CSA can precisely determine the value of $buf[4]$ as the initial value $[0, 0]$ stored at $f_5$, not affected by the value $[5, 5]$ stored at $f_6$. This is because CSA removes the spurious data dependence $f_5 \rightarrow f_6$, preventing $[5, 5]$ at $f_6$ from propagating to $f_6$. 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 $f_5$, and the constant 10 is stored in $pb$ at $f_6$. At $f_6$, the array buffer 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 $f_6$.

Moreover, CSA can handle the side effects of standard library functions (e.g., sscanf, fgets, snprintf) precisely. For example, in Figure 7(c), the variable idx is used as an offset to write to the fixed-length array buffer at $f_5$. However, the value of $idx$ is specified by the external I/O function fscanf at $f_5$, 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 $f_5$ because the value of $idx$ can be larger than that of buffer. In comparison, all our baselines except KLEE overlook this vulnerability. Another example is shown in Figure 7(d), where the size of the array buffer at $f_5$ is determined by the variable len. At $f_6$, the program calls fgets to fill buffer 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 $f_6$, 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 $f_6$. 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 Mem columns indicates a cost of more than 100 Gigabytes.

<table>
<thead>
<tr>
<th>Project</th>
<th>Infer</th>
<th>Cppcheck</th>
<th>IKOS</th>
<th>KLEE</th>
<th>Sparrow</th>
<th>CSA-CP</th>
<th>CSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>paste</td>
<td>1 10</td>
<td>7 61</td>
<td>0 17</td>
<td>1 9</td>
<td>2 35</td>
<td>966</td>
<td>1684</td>
</tr>
<tr>
<td>mSUM</td>
<td>0 2</td>
<td>2 8</td>
<td>8 0</td>
<td>1 1</td>
<td>2 2 35</td>
<td>966</td>
<td>1684</td>
</tr>
<tr>
<td>YAL</td>
<td>1 17</td>
<td>9 110</td>
<td>0</td>
<td>14 1</td>
<td>2 1625</td>
<td>8622</td>
<td>3125</td>
</tr>
<tr>
<td>MP4v2</td>
<td>1 28</td>
<td>1 313</td>
<td>3 1 1</td>
<td>1 26</td>
<td>38</td>
<td>38 1 9</td>
<td>596</td>
</tr>
<tr>
<td>RIOT</td>
<td>3 2</td>
<td>9</td>
<td>115</td>
<td>2 22 1 2 1325</td>
<td>2862</td>
<td>1125</td>
<td>38368</td>
</tr>
<tr>
<td>darknet</td>
<td>25 134</td>
<td>837</td>
<td>282</td>
<td>16</td>
<td>2144</td>
<td>10 55</td>
<td>14</td>
</tr>
<tr>
<td>lnx</td>
<td>15 142</td>
<td>522</td>
<td>989</td>
<td>3</td>
<td>156</td>
<td>30</td>
<td>39 1 4</td>
</tr>
<tr>
<td>Teeworlds</td>
<td>10 169</td>
<td>684</td>
<td>934</td>
<td>4</td>
<td>187</td>
<td>2</td>
<td>54 1 2</td>
</tr>
<tr>
<td>NanoQ</td>
<td>23 154</td>
<td>654</td>
<td>395</td>
<td>8</td>
<td>147</td>
<td>94</td>
<td>38</td>
</tr>
<tr>
<td>redis</td>
<td>6 137</td>
<td>1292</td>
<td>10484</td>
<td>8</td>
<td>136</td>
<td>516</td>
<td>123</td>
</tr>
</tbody>
</table>

Total: 76 846 4447 73655 44 934 695 401 29 7388 47718 125157 55 35 120935 441344 46 7033 8291 19008 111 1538 18036 24141 111 51 14975 20994

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 ability and precision, considering the benefits of its bug detection ability and precision, considering the benefits of its bug detection ability and precision.
Table 5: Comparison between CSA and CSA-NI (a version of CSA without implication-equivalent memory addresses).

<table>
<thead>
<tr>
<th>Project</th>
<th>CSA-NI Time (secs)</th>
<th>CSA Time (secs)</th>
<th>CSA Mem (MB)</th>
<th>CSA-NI Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmux</td>
<td>1540 (1.87x)</td>
<td>824</td>
<td>21016 (4.16x)</td>
<td>5052</td>
</tr>
<tr>
<td>Teeworlds</td>
<td>6176 (2.14x)</td>
<td>2886</td>
<td>14237 (5.48x)</td>
<td>2598</td>
</tr>
<tr>
<td>NanoMQ</td>
<td>3292 (2.58x)</td>
<td>1143</td>
<td>48805 (7.45x)</td>
<td>6551</td>
</tr>
<tr>
<td>redis</td>
<td>21232 (3.24x)</td>
<td>6533</td>
<td>32314 (8.35x)</td>
<td>3870</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>(2.47x)</td>
<td>(6.14x)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPARROW report a false alarm here because they treat n2 aliased with n1, wherein n1->len holds an unbounded value at label \( \ell_4 \). 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 \( \ell_5 \) and step at \( \ell_6 \), 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 \( \ell_5 \) could be null and is dereferenced at \( \ell_0 \). There are no null pointer checkings for \( \ell_0 \). Figure 9(c) presents a buffer overflow bug found by CSA in darknet while KLEE fails to find this bug. The buffer label1 is initialized at \( \ell_2 \) using a file reading function get_labels. The buffer size is an unbounded value depending on the input file f. At \( \ell_0 \), the buffer label1 is accessed using the variable j, whose value ranges from [0, 12] indicated by the loop guard at \( \ell_6 \). 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 2.47x, with up to 3.24x on the redis project. CSA-NI also uses an average of around 6.14x more memory than CSA, with a maximum of 8.35x on the redis project. These results demonstrate the improvement brought by our implication-equivalent memory address sets regarding analysis time and memory usage.

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 3.24x, compared to merely 1.87x for tmux. Additionally, redis demonstrates a memory saving of 8.35x, compared to only 4.16x 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, ASTREE [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 applications, buffer overflow and null dereference detection. In the ten open-source projects in two assertion-based checking client applications, buffer overflow and null dereference detection. In the ten open-source projects in two assertion-based checking client applications, buffer overflow and null dereference detection.

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
