Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing

XIAO CHENG, University of New South Wales, Australia
JIAWEI REN, University of New South Wales, Australia
YULEI SUI, University of New South Wales, Australia

Typestate analysis is a commonly used static technique to identify software vulnerabilities by assessing if a sequence of operations violates temporal safety specifications defined by a finite state automaton. Path-sensitive typestate analysis (PSTA) offers a more precise solution by eliminating false alarms stemming from infeasible paths. To improve the efficiency of path-sensitive analysis, previous efforts have incorporated sparse techniques, with a focus on analyzing the path feasibility of def-use chains. However, they cannot be directly applied to detect typestate vulnerabilities requiring temporal information within the control flow graph like use-to-use information.

In this paper, we introduce FGS, a Fast Graph Simplification approach designed for PSTA by retaining multi-point temporal information while harnessing the advantages of sparse analysis. We propose a new multi-point slicing technique that captures the temporal and spatial correlations within the program. By doing so, it optimizes the program by only preserving the necessary program dependencies, resulting in a sparser structure for precision-preserving PSTA. Our graph simplification approach, as a fast preprocessing step, offers several benefits for existing PSTA algorithms. These include a more concise yet precision-preserving graph structure, decreased numbers of variables and constraints within execution states, and simplified path feasibility checking. As a result, the overall efficiency of the PSTA algorithm exhibits significant improvement.

We evaluated FGS using NIST benchmarks and ten real-world large-scale projects to detect four types of vulnerabilities, including memory leaks, double-frees, use-after-frees, and null dereferences. On average, when comparing FGS against ESP (baseline PSTA), FGS reduces 89% of nodes, 86% of edges, and 88% of calling context of the input graphs, obtaining a speedup of 116× and a memory usage reduction of 93% on the large projects evaluated. Our experimental results also demonstrate that FGS outperforms six open-source tools (IKOS, ClangSA, Saber, Cppcheck, Infer, and Sparrow) on the NIST benchmarks, which comprises 846 programs. Specifically, FGS achieves significantly higher precision, with improvements of up to 171% (42% on average), and detects a greater number of true positives, with enhancements of up to 245% (52% on average). Moreover, among the ten large-scale projects, FGS successfully found 105 real bugs with a precision rate of 82%. In contrast, our baseline tools not only missed over 42% of the real bugs but also yielded an average precision rate of just 13%.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; **Model checking**.

Additional Key Words and Phrases: Graph simplification, multi-point slicing, path-sensitive typestate analysis

ACM Reference Format:
Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing. *Proc. ACM Softw. Eng.* 1, FSE, Article 23 (July 2024), 23 pages. https://doi.org/10.1145/3643749

Authors' addresses: Xiao Cheng, University of New South Wales, Sydney, Australia, xiao.cheng@unsw.edu.au; Jiawei Ren, University of New South Wales, Sydney, Australia, jiawei.ren@student.unsw.edu.au; Yulei Sui, University of New South Wales, Sydney, Australia, y.sui@unsw.edu.au.

1 INTRODUCTION

Typestate analysis [30, 68] aims at determining whether a sequence of program operations violates a safety specification given a finite state automaton. This approach has shown success in identifying a spectrum of vulnerabilities, including use-after-frees/double-frees [72, 76], memory/resource leakages [45, 50], access controls [59], and concurrency bugs [32, 38]. Path-sensitive typestate analysis (PSTA) [30, 33, 50] elevates the precision of its path-insensitive counterpart by capturing inter-branch correlations and reducing false alarms arising from infeasible paths. PSTA typically requires maintaining an execution state that captures the values of program variables and path constraints, and it validates path feasibility when encountering branches.

Existing efforts and limitations. ESP [30, 33] is a representative instance of PSTA that scales to large programs. Unlike full path-sensitive analyses that compute a solution through a meet-over-all-path approach, ESP takes a partially path-sensitive approach to avoid examining a potentially unbounded number of program paths. It employs a symbolic state, incorporating both an execution state and an automaton's "property state" (typestate) [30] as a data-flow fact. At a control-flow joint point, ESP merges execution states with identical typestates, yielding a single symbolic state and thus achieving a maximal-fixed-point solution with program paths sensitive to typestate preserved. Subsequently, several approaches [37, 44, 50, 80] propose more effective analysis based on optimizing the typestate transitions. Their main focus is to improve the precision of alias analysis and eliminate spurious typestate transitions on non-aliased objects. For instance, Fink et al. [37] propose incorporating typestate analysis with alias analysis and concentrating precise alias analysis only on relevant typestate properties. Additionally, there are hybrid typestate analysis approaches attempting to combine static typestate analysis with the dynamic residual runtime monitor [15, 16, 35] or fuzzing [72] to detect typestate vulnerabilities.

To the best of our knowledge, all previous efforts in PSTA primarily focused on enhancing the precision of alias analysis or exploring opportunities for integrating dynamic analysis techniques. Nonetheless, there remains considerable potential for optimizing PSTA's efficiency, which is a hard but relatively unexplored research area. Recently, several approaches propose sparse path-sensitive analyses [64, 65, 71] to improve the efficiency of path-sensitive analysis. The key idea is to propagate data flow facts along def-use chains and skip unnecessary control flows. For example, Fusion [65] proposes a fused approach to validate path feasibility for def-use chains efficiently. However, the def-use chains lack information on the temporal execution order within the control flow graph, such as use-to-use information. Hence, the existing sparse approaches cannot be directly applied to detect vulnerabilities associated with typestate concerns. Moreover, in contrast to the focus of the sparse analysis on solving a reachability problem from sources to sinks, typestate analysis often requires tracking multiple sequential program points to identify potential error states, such as detecting use-after-frees. It becomes necessary to consider the multi-point temporal and spatial correlations in the typestate analysis process.

Insights and challenges. To preserve the multi-point temporal and spatial correlations while harnessing the benefits of sparse analysis, we focus on graph simplification, a more general and practical approach by reducing the size of the control flow graph, yielding a sparser structure used by the same PSTA without modifying the underlying analysis algorithm. Our insight comes from slicing techniques [51, 61, 67], which can simplify the analysis by selectively considering the code fragment related to specific program points of interest. However, most of these slicing techniques are designed for path-insensitive analysis and are for different purposes, e.g., program debugging and comprehension. No prior works have explored slicing to improve the PSTA's scalability. The primary technical challenge lies in devising an effective and efficient multi-point slicing algorithm. Effectiveness refers to the ability of the algorithm to eliminate as many irrelevant nodes as possible,

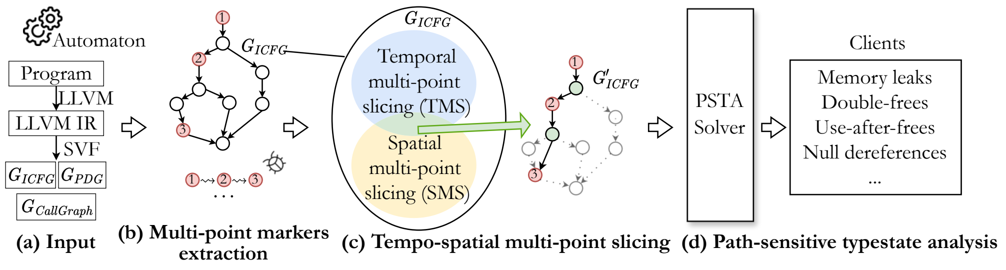


Fig. 1. Overview of our approach.

maximizing the extraction of relevant information. For example, an intuitive approach that performs a backward slicing on the sparse value-flow graph [70] does not consider the multi-point temporal and spatial properties of a typestate analysis, which significantly limits the slicing effectiveness. The ideal algorithm should also operate efficiently, serving as a pre-processing step that executes significantly faster than the PSTA algorithm itself. Furthermore, it is crucial to ensure that the soundness and precision of PSTA are not compromised after applying the slicing technique.

Our solution. We present FGS, a fast graph simplification approach for PSTA through tempo-spatial multi-point slicing that effectively compacts the interprocedural control flow graph (ICFG). Our graph simplification approach is not dependent on later typestate algorithms and can be applied across various PSTA solvers, also accommodating different alias analyses or dynamic approaches. The process initiates with a fast path-insensitive multi-point markers extraction, soundly identifying all potentially vulnerable operation sequences (aka multi-point markers). Guided by these multi-point markers, we present a new slicing technique that accounts for the temporal orders of these markers while retaining only the necessary spatial program dependencies (i.e., control and data dependencies) within the ICFG. Eliminating unnecessary ICFG nodes results in significant compaction of the control flow graph, resulting in a sparser structure with fewer edges and calling contexts. Consequently, a PSTA algorithm benefits from this graph simplification through a more compact ICFG with reduced variables and path constraints in execution states and fast SMT solving. Importantly, the soundness and precision of the PSTA algorithm remain unchanged because (1) the multi-point markers extracted from the path-insensitive analysis is an over-approximation of the sequences under path-sensitive analysis, and (2) all essential temporal and spatial information for determining the feasibility of the markers is preserved within the simplified ICFG.

Framework overview. Figure 1 provides an overview of our framework. The input of our framework is a finite-state automaton, and LLVM's intermediate representation (IR) [48], as formally defined in Section 2.1. This IR is then passed to a static analysis tool SVF [70], which produces G_{ICFG} an interprocedural control flow graph (ICFG), a program dependence graph G_{PDG} and a call graph $G_{CallGraph}$. We first extract multi-point markers (i.e., path-insensitive vulnerable operation sequences) on G_{ICFG} according to the automaton using a fast path-insensitive typestate analysis by treating all G_{ICFG} edges without considering branch conditions. The markers are used to guide our multi-point slicing module to acquire the program statements necessary for the costly path-sensitive analysis and guarantee that the vulnerable operation sequences are equivalent to the original PSTA. The multi-point slicing module consists of two key parts: a temporal multi-point slicing (TMS) and a spatial multi-point slicing (SMS). TMS acquires the statements from all the temporal paths with each containing at least one marker. SMS works on G_{PDG} to capture all the control and data dependencies of the program points in the markers. TMS and SMS work collectively to preserve the temporal and spatial correlations within the program. PSTA can then work on a more compacted graph G'_{ICFG} produced based on the tempo-spatial slice, an intersection of the temporal and spatial slices, and efficiently validate the feasibility of multi-point markers according to their temporal and spatial dependencies. The PSTA solution on G'_{ICFG} supports subsequent typestate-related clients, such as detecting memory leaks, double-frees, use-after-frees, and null dereferences.

Our major contributions are as follows:

- We present FGS, a fast graph simplification approach for path-sensitive typestate analysis utilizing tempo-spatial multi-point slicing.
- We formulate the multi-point markers extraction as a graph reachability problem based on the IFDS framework [60].
- We propose a new multi-point slicing technique that efficiently captures the temporal and spatial correlations necessary for a path-sensitive typestate analysis.
- We conduct a comprehensive evaluation of our approach's performance using 846 programs from the NIST dataset and ten real-world open-source projects. Experimental results show that our approach reduces ICFG nodes by 89% and calling context by 88% on average, resulting in a 116× speedup and 93% reduced memory usage. Our approach also outperforms our baselines, namely IKOS [17], Saber [71], Infer [43], Cppcheck [25], ClangSA [47] and Sparrow [56], with 42% higher precision (up to 171%) and more true positives (up to 245%) on NIST dataset. In ten large-scale projects, FGS identified 105 real bugs with impressive precision (82%), outperforming baseline tools.

2 PRELIMINARIES AND PROBLEM FORMULATION

We introduce the preliminary knowledge including the target language, typestate and PSTA.

2.1 LLVM-like Language

Table 1 gives the LLVM-like language [48] for conducting path-sensitive typestate analysis. The set of all variables $\mathcal{V}$ is divided into two distinct subsets: $\mathcal{O}$, including all possible abstract objects, i.e., *address-taken variables* of a pointer and $\mathcal{P}$, representing all *top-level variables*. In LLVM's language, a top-level variable, such as $\mathtt{p}, \mathtt{q}, \mathtt{r} \in \mathcal{P}$, can be defined only once. An address-taken object, such as $\mathtt{o} \in \mathcal{O}$, can be read/modified only through dereferencing top-level pointers within StoreStmt and LoadStmt. A constant value c is consistently allocated to a top-level variable as its initial assignment at ConsStmt. For AddrStmt $\mathtt{p} = \mathtt{alloc}_\mathtt{o}$, o pertains to either a stack or global variable, or it takes the form of an abstract heap object created dynamically. GepStmt serves as a representation for accessing fields within a struct object, whereby its field offset `fld` is maintained as a constant value. FGS uses a field-index-based approach to field-sensitivity similar to [13, 57]. The distinction between fields within a struct object is established through their individual and unique indices. PhiStmt is a standard SSA instruction introduced at a confluence point on the control-flow graph to facilitate the selection of variable values from different branches. Lastly, passing parameters to a callee or returning from it at a specific callsite is represented by CopyStmts. In the following sections, we may use program statements and instructions interchangeably.

Table 1. Analysis domain and LLVM-like language.

c, fld	$\in \mathcal{C}$	Constants
p, q, r	$\in \mathcal{S}$	Stack virtual registers
g	$\in \mathcal{G}$	Global pointer variables
p, q, r, g	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	Top-level Variables
o, a, $\mathtt{a}_\mathtt{f}$, a.fld, a[c]	$\in \mathcal{O}$	Abstract objects
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	Program variables
ℓ ::=		Stmt
p = c		ConsStmt
$\mathtt{p} = \mathtt{alloc}_\mathtt{o}$		AddrStmt
p = &(q→fld)		GepStmt (Field)
p = &q[c] (constant)		GepStmt (Array-C)
p = &q[v] (variable)		GepStmt (Array-V)
p = ∗q		LoadStmt
∗p = q		StoreStmt
p = q		CopyStmt
$\mathtt{p} = \mathtt{phi}(\mathtt{p}_1, \mathtt{p}_2, ...\mathtt{p}_\mathtt{n})$		PhiStmt
p = ¬q		UnaryStmt
r = p ⊙ q		BinaryStmt
$\odot \in \{+, -, *, /, \%, <<, >>, <, >, \&, \&\&, <=, >=, \equiv, \sim, \vert, \land\}$		

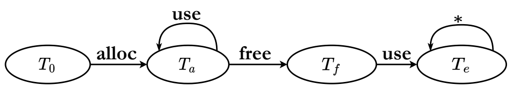


Fig. 2. τ for detecting use-after-frees.

2.2 Typestate Analysis

Typestate [68] expands the scope of standard immutable types for potential state changes by representing the different states of a given type and its state transitions through a typestate finite-state automaton (Definition 1). Typestates define valid sequences of operations that can be performed upon an instance of a given type. Typestate analysis is well-suited for specification-based bug detection, protocol analysis and so on.

Definition 1 (Typestate finite-state automaton). A typestate finite-state automaton $\tau = \langle \Sigma, \mathbb{T}, T_0, \delta, T_e \rangle$ has five tuples. The language Σ denotes the operations or instructions (e.g., function calls) that can be performed on the typestates. $\mathbb{T}$ represents all the possible typestates, and $T_0 \in \mathbb{T}$ is the initial state. The state-transition table $\delta \in (\mathbb{T} \times \Sigma) \longrightarrow \mathbb{T}$ consists of all the state-transition functions encoding the effects of operations in Σ, which maps one state to another given an instruction. T_e is the error typestate indicating a potential bug detected. The error state is a trap state with a self-loop transition.

Example 1 (Use-after-free (UAF) typestate analysis). Figure 2 shows the typestate finite-state automaton for detecting use-after-free vulnerability [28]. The language Σ consists of memory allocation (*alloc*), memory free (*free*) and memory usage (*use*). $\mathbb{T} = \{T_0, T_a, T_f, T_e\}$ includes an initial state T_0, an allocated state T_a, a freed state T_f and an error state T_e. T_0 transfers to T_a when encountering a memory allocation. T_a changes to T_f if the allocated memory is released. The error state T_e is reached when using the memory object holding a T_f state.

Definition 2 (Operation sequence). An operation sequence is a sequence of instructions $\ell_1 \rightsquigarrow \cdots \rightsquigarrow \ell_n$ on G_{ICFG}, where $\ell_1 \ldots \ell_n \in \Sigma$ are the operations in Σ of a given τ. The order of operations on the sequence should follow the temporal execution order of the target program.

Definition 3 (Vulnerable operation sequence). A vulnerable operation sequence π is an operation sequence $\ell_1 \rightsquigarrow \cdots \rightsquigarrow \ell_n$ that leads to an error typestate. An initial typestate T_0 changes into the error state T_e when sequentially performing each instruction on π. In the following sections, we may use sequence(s) as the shorter form of vulnerable operation sequence(s) for brevity.

Example 2 (Vulnerable operation sequence for UAF). Suppose that we have three consecutive program instructions ℓ_1 : `p = malloc();` ℓ_2 : `free(p);` ℓ_3 : `print(p);`. There is a vulnerable operation sequence $\ell_1 \rightsquigarrow \ell_2 \rightsquigarrow \ell_3$, which follows the temporal order of the program and leads to an error state at the end of the instructions.

2.3 Path-Sensitive Typestate Analysis

We use ESP [30], a prime instance of PSTA implemented in the IFDS framework [60], as the baseline PSTA used in our paper. As in Definition 5, ESP computes and maintains symbolic states (a combination of typestates and execution states) as data-flow facts when analyzing the target program. At a joint point in ICFG, ESP merges the symbolic states with the same typestate, while the symbolic states with different typestates are propagated and analyzed independently in a path-sensitive manner. We first introduce the symbolic state and then present the flow functions for handling program instructions and then the PSTA algorithm.

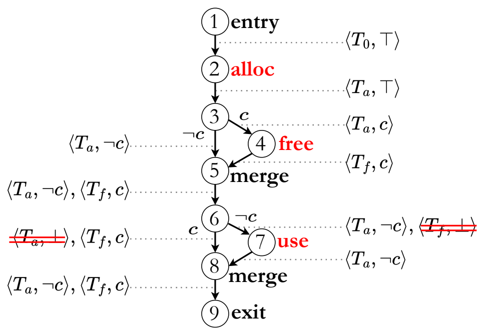
Definition 4 (Execution state). We define the execution state for a given program as a composition of a map $\mathcal{V} \longrightarrow \mathbb{A}$ from program variables $\mathcal{V}$ to an abstract domain $\mathbb{A}$, an over-approximated abstraction of the concrete domain [24], and a path constraint [18] that captures the conditions that must hold for a specific execution path to be taken.

Definition 5 (Symbolic state)**.** A symbolic state $S \in \mathbb{S}$ is a pair of typestate (Definition 1) and execution state (Definition 4). We use $\mathrm{ts}(S)$ and $\mathrm{es}(S)$ to retrieve the typestate and the execution state of the symbolic state S, respectively.

Flow functions. A flow function encodes how symbolic states transit given a program instruction ℓ. Equation 1 shows the flow functions used in ESP ($\mathsf{F_m}$, $\mathsf{F_b}$ and $\mathsf{F_o}$) and a grouping function (α), which merges the symbolic states with equivalent typestate. $\mathsf{F_m}$ handles the control-flow joint point. It first joins the symbolic states from the two incoming branches and calls α to group symbolic states. $\mathsf{F_b}$ handles the branch instruction. It checks path feasibility according to the branch condition ($cond$) for each symbolic state using an SMT solver and only keeps symbolic states with a feasible execution state. $\mathsf{F_o}$ handles the other program instructions. It updates the execution state based on the semantics of the instruction ℓ. The typestate will also be updated if the instruction is in the language, i.e., $\ell \in \Sigma$. In the end, α is applied to the updated symbolic states.

$$
\begin{aligned}
&\mathsf{F_m}(\ell, SS_1, SS_2) = \alpha(SS_1 \cup SS_2)\\
&\mathsf{F_b}(\ell, SS, cond) = \{S' \mid S' = \mathsf{f_b}(\ell, S, cond) \land S \in SS \land \mathrm{es}(S') \neq \bot\}\\
&\mathsf{F_o}(\ell, SS) = \alpha(\{\mathsf{f_o}(\ell, S) \mid S \in SS\})\\
&\alpha(SS) = \{\langle T, \sqcup_{S \in SS[T]} es(S)\rangle \mid T \in \mathbb{T} \land SS[T] \neq \emptyset\}\\
&SS[T] = \{S \mid S \in SS \land T \in \mathrm{ts}(S)\}
\end{aligned}
\tag{1}
$$

PSTA algorithm. ESP is a typical worklist algorithm that works on the G_{ICFG} by propagating each pair $\langle n, T\rangle$ consisting of an ICFG node n and typestate T along G_{ICFG}. ESP then derives a symbolic state map that maps from a control-flow edge and typestate pair to symbolic states $Info : E \times \mathbb{T} \longrightarrow \mathscr{P}(\mathbb{S})$, where $\mathscr{P}(\mathbb{S})$ denotes the powerset of $\mathbb{S}$. The algorithm terminates when a fixed point is reached, i.e., $Info$ remains unchanged. A bug is recorded if there exists an error typestate in $Info$.



① entry
$\langle T_0, \top\rangle$
② alloc
$\langle T_a, \top\rangle$
③
c
$\langle T_a, c\rangle$
$\langle T_a, \neg c\rangle$
$\neg c$
④ free
$\langle T_f, c\rangle$
⑤ merge
$\langle T_a, \neg c\rangle, \langle T_f, c\rangle$
⑥
$\neg c$
$\langle T_a, \neg c\rangle$, ~~$\langle T_f, \bot\rangle$~~
~~$\langle T_a, \bot\rangle$~~, $\langle T_f, c\rangle$
c
⑦ use
$\langle T_a, \neg c\rangle$
⑧ merge
$\langle T_a, \neg c\rangle, \langle T_f, c\rangle$
⑨ exit

Fig. 3. An example of intraprocedural PSTA.

Once an item $\langle n, T\rangle$ is selected from the worklist, the analysis will employ the corresponding flow functions based on n's type (i.e., different kinds of program instructions) in Equation 1 in order to derive symbolic states and update $Info$. For the control-flow joint point, ESP groups the symbolic states using the merge flow function $\mathsf{F_m}$. When encountering program branches, we apply $\mathsf{F_b}$ to check path feasibility. For the other nodes, we call $\mathsf{F_o}$ to update the execution state and/or typestate.

Example 3. Figure 3 shows an example of intraprocedural PSTA. A memory object is created at ②, released at ④ and used at ⑦. We analyze the program with automaton τ in Figure 2 and the flow functions in Equation 1. For the memory allocation at ②, we apply $\mathsf{F_o}$ to produce $\langle T_a, \top\rangle$ from $\langle T_0, \top\rangle$. For the conditional branch at ③, we check path feasibility and encode path constraints in the execution state according to $\mathsf{F_b}$, yielding $\langle T_a, c\rangle$ and $\langle T_a, \neg c\rangle$ at ③$\xrightarrow{c}$④ and ③$\xrightarrow{\neg c}$⑤ respectively. For the control-flow joint point ⑤, we merge its incoming symbolic states using $\mathsf{F_m}$ and produce $\{\langle T_a, \neg c\rangle, \langle T_f, c\rangle\}$. Note that, ②$\rightsquigarrow$④$\rightsquigarrow$⑦ would form a vulnerable operation sequence

(Definition 3) if using a path-insensitive typestate analyis. However, this is a false alarm because the control-flow guards ③$\xrightarrow{c}$④ and ⑥$\xrightarrow{\neg c}$⑦ contradict each other. PSTA can eliminate this false alarm because $\langle T_a, \neg c\rangle$ and $\langle T_f, c\rangle$ are analyzed independently and precisely at ⑥. $\langle T_f, c\rangle$ does not pass to ⑦ due to infeasible path given the conjunction of contradictory constraints $c \wedge \neg c$.

Interprocedural PSTA and IFDS. The interprocedural PSTA algorithm is also a worklist algorithm similar to intraprocedural PSTA. The flow functions $\mathsf{F_m}$, $\mathsf{F_b}$ and $\mathsf{F_o}$ are the same as the intraprocedural PSTA. In addition, interprocedural PSTA considers function calls by handling callsites and function exits in a context-sensitive manner.

The algorithm is implemented based on the IFDS framework [60], where the data flow facts are symbolic states. A summary map is used to summarize and cache the symbolic states generated by each function by mapping dataflow facts at the entry to the exit. At each call site, the algorithm checks whether the summary of the callee has been created, and reuses the summarized symbolic states if the summary exists. At each function exit, the algorithm generates the summary of the corresponding function.

2.4 Problem Formulation

Our objective is to boost the efficiency of the path-sensitive typestate algorithm while maintaining the same precision. This improvement can be done by focusing on simplifying G_{ICFG}, which, in turn, contributes to the reduction of execution states and alleviates the computational cost of SMT solving, which represents a significant portion of the algorithm's overhead.

Intuitively, an ideal graph simplification approach should aim to remove as many ICFG nodes as possible. However, simply eliminating nodes without considering the temporal and spatial correlations between variables can alter the sequences of operations in the typestate language. This disruption can lead to either a false alarm or a false negative, resulting in an incorrect PSTA solution. For instance, when control-flow nodes that affect branch conditions are eliminated, they can influence path feasibility and may result in spurious sequences of operations in the simplified graph. A correct graph simplification approach must ensure the equivalence of vulnerable operation sequences (Definition 3) between the original ICFG and the simplified ICFG. Hence, it is important to preserve relevant instructions (or ICFG nodes) when evaluating the path feasibility within the simplified graph. Note that our approach does not aim to retain non-vulnerable operation sequences as they do not contribute to bug detection. In the following sections, we use Π to represent path-sensitive sequences and $\hat{\Pi}$ to denote path-insensitive sequences (multi-point markers).

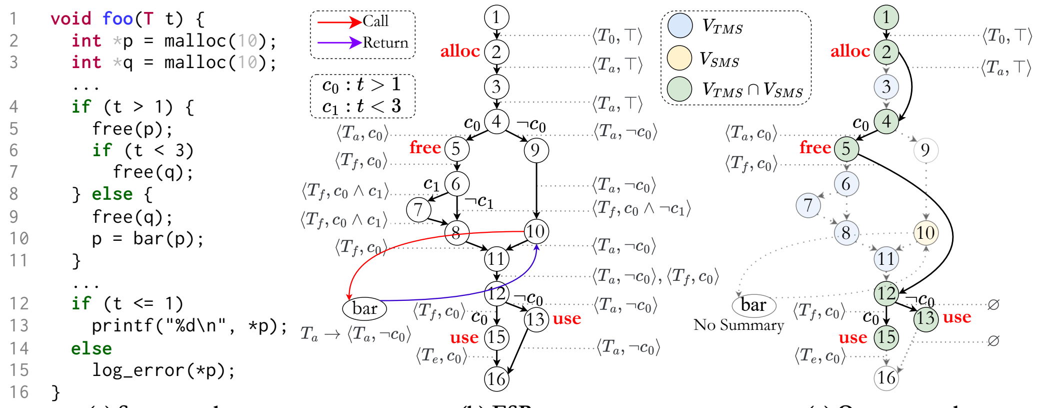
Definition 6 (Typestate reachability equivalence)**.** Let G_{ICFG} be the original interprocedural control-flow graph, and let G'_{ICFG} be the simplified graph. Given an automaton τ, let Π_τ be the vulnerable operation sequences extracted from the G_{ICFG} and Π'_τ be the sequences extracted from G'_{ICFG} under the same path-sensitive analysis. G_{ICFG} and G'_{ICFG} are typestate reachability equivalent iff $(\forall \pi \in \Pi_\tau : \pi \in \Pi'_\tau) \wedge (\forall \pi' \in \Pi'_\tau : \pi' \in \Pi_\tau) \wedge (\pi \notin \Pi_\tau \Leftrightarrow \pi \notin \Pi'_\tau)$.

We formulate our graph simplification problem as follows:

Given an interprocedural control-flow graph G_{ICFG} and an automaton τ, our approach aims to generate a smaller graph G'_{ICFG} by removing redundant nodes on G_{ICFG} via multi-point slicing and guarantee that G_{ICFG} and G'_{ICFG} are typestate reachability equivalent.

3 A MOTIVATING EXAMPLE

We present an example in Figure 4 to illustrate how FGS speedups ESP [30] a traditional PSTA by simplifying G_{ICFG} through tempo-spatial multi-point slicing using a use-after-free vulnerability.



```
1   void foo(T t) {
2     int *p = malloc(10);
3     int *q = malloc(10);
      ...
4     if (t > 1) {
5       free(p);
6       if (t < 3)
7         free(q);
8     } else {
9       free(q);
10      p = bar(p);
11    }
      ...
12    if (t <= 1)
13      printf("%d\n", *p);
14    else
15      log_error(*p);
16  }
```

Call
Return
$c_0 : t > 1$
$c_1 : t < 3$
alloc
free
use
bar
$\langle T_0, \top\rangle$
$\langle T_a, \top\rangle$
$\langle T_a, c_0\rangle$
$\langle T_f, c_0\rangle$
$\langle T_f, c_0 \wedge c_1\rangle$
$\langle T_a, \neg c_0\rangle$
$\langle T_f, c_0 \wedge \neg c_1\rangle$
$\langle T_a, \neg c_0\rangle, \langle T_f, c_0\rangle$
$T_a \rightarrow \langle T_a, \neg c_0\rangle$
$\langle T_e, c_0\rangle$
V_{TMS}
V_{SMS}
$V_{TMS} \cap V_{SMS}$
No Summary
$\varnothing$

(a) Source code **(b) ESP** **(c) Our approach**

Fig. 4. ESP vs PSTA on the simplified ICFG. The code fragment contains a use-after-free bug at ℓ_{15} for the memory object p allocated at ℓ_2.

The UAF-related source code snippet is depicted in Figure 4(a). Our analysis focuses on the allocated object p at ②. The UAF bug is triggered at the memory usage `*p` at ⑮, which is allocated with `malloc` at ② but is released with `free` at ⑤ before being accessed at ⑮. Note that, a path-insensitive typestate analysis would report another (false) use-after-free bug at ⑬. This produces two multi-point markers $\hat{\Pi}$: {②⇝⑤⇝⑬, ②⇝⑤⇝⑮}. The first sequence is infeasible under a path-sensitive analysis because the branch condition ④$\xrightarrow{c_0}$⑤ and ⑫$\xrightarrow{\neg c_0}$⑬ contradict each other. Hence there is only one true vulnerable operation sequence, that is Π = {②⇝⑤⇝⑮}.

Sparse analysis, ESP and their limitations. The current sparse path-sensitive analysis cannot distinguish between use-after-free and free-after-use because the value-flow graph used in the sparse path-sensitive analysis does not have information on the temporal order from ⑤ to ⑬ and to ⑮ (use-to-use relation). Hence, reporting "free-after-use" as a vulnerability can cause false positives. In contrast, ESP can track the use-to-use temporal order from ⑤ to ⑬ and to ⑮. The ESP approach in Figure 4(b) involves traversing G_{ICFG} and deriving symbolic states at each program point. In comparison, an ideal simplified graph G'_{ICFG} is shown in Figure 4(c), which is significantly smaller than G_{ICFG}. Meanwhile, G'_{ICFG} is typestate reachability equivalent (Definition 6) as G_{ICFG} because the path-sensitive sequences are equivalent, i.e., $\Pi \equiv \Pi'$ (both are {②⇝⑤⇝⑮}). Note that, removing any nodes on G'_{ICFG} would alter the sequences (e.g., removing ②, ⑤, ⑬ or ⑮) or influence the path feasibility (e.g., removing ① ,④ or ⑫).

FGS approach. Our graph simplification captures temporal and spatial correlations by preserving the temporal information while leveraging sparse analysis to keep necessary spatial and temporal dependencies. This can boost the efficiency of PSTA while not compromising the analysis precision. In Figure 4(b), we contrast the ESP performed on the original ICFG, introduced in Section 2.3, with the PSTA on the simplified ICFG via multi-point slicing illustrated in Figure 4(c). FGS first runs a fast yet cheap path-insensitive analysis to extract $\hat{\Pi}$: {②⇝⑤⇝⑬,②⇝⑤⇝⑮}. Guided by $\hat{\Pi}$, a multi-point slicing is conducted to capture temporal and spatial dependencies of $\hat{\Pi}$. These dependencies produced by slicing facilitate PSTA to efficiently validate the feasibility of $\hat{\Pi}$ and yield equivalent sequences as ESP Π = {②⇝⑤⇝⑮} because ②⇝⑤⇝⑬ is infeasible (④$\xrightarrow{c_0}$⑤ and ⑫$\xrightarrow{\neg c_0}$⑬ contradict each other). Temporal multi-point slicing (TMS) extracts all execution paths with each containing at least one vulnerable operation sequence. ⑨, ⑩ and ⑯ can be excluded on the temporal slice because no program executions containing at least one $\hat{\pi} \in \hat{\Pi}$ would pass ⑨, ⑩ and ⑯. The absence of TMS would require feasibility checking at ④→⑨, resulting in additional overhead. Furthermore, it also requires analyzing the function bar at the callsite ⑩. Spatial multi-point slicing (SMS) identifies control and data dependencies related to the sequence. ③, ⑥, ⑦, ⑧, and ⑪ can be excluded on the spatial slice because they have no control or data dependence relation with the nodes on $\hat{\Pi}$. The absence of SMS would need to compute the execution state and feasibility checking at ⑥→⑦ and ⑥→⑧. The path constraint would include the redundant branch condition c_1 at ⑥. The tempo-spatial slice, an intersection of the temporal and spatial slices, consists of only seven nodes, ①, ②, ④, ⑤, ⑫, ⑬ and ⑮.

Benefits. A smaller graph G'_{ICFG} can contribute to fewer symbolic state propagations during PSTA. While ESP yields 18 symbolic states on the original ICFG and yields just 6 on the simplified ICFG. This translates to significant computational savings, particularly in terms of function summaries at irrelevant callsites (e.g., ⑩) and faster convergence to a fixed-point in the PSTA algorithm, due to the simplified graph structure with fewer control-flow joint nodes. Furthermore, execution states become more compact as irrelevant variables and path constraints are eliminated. Our graph simplification approach not only improves time and space during symbolic state propagation but also reduces the cost of SMT solving, given the simplified path constraints.

4 FGS APPROACH

In this section, we provide details for extracting multi-point markers $\hat{\Pi}$ in Section 4.1, followed by the tempo-spatial multi-point slicing approach in Section 4.2. We discuss the correctness and complexity of each component, and several properties to state FGS's correctness and the efficiency improvement it delivers.

4.1 Multi-Point Markers Extraction

Intuitively, we can either enumerate the instructions in the typestate language Σ or utilize depth/bread-first search on G_{ICFG} to extract multi-point markers $\hat{\Pi}$ (path-insensitive vulnerable operation sequences). However, enumerating instructions may yield unrealizable sequences that do not exist on G_{ICFG}, while depth/bread-first search may result in an exponential blowup of program paths [51]. This is almost as expensive as solving all path feasibility and clearly undermines the goal of improving scalability as a pre-analysis. We propose $\hat{\Pi}$-extraction by identifying multi-point markers in a context-sensitive but path-insensitive manner by solving an IFDS data-flow problem [30], where the data flow facts are in the form of symbolic chains $\mathbb{Q}$ (Definition 7). Symbolic chains contain information of the multi-point markers. Hence, we collect symbolic chains using the fast IFDS algorithm to obtain multi-point markers for later multi-point slicing. In the following sections, we will give the definition of symbolic chains and present the flow functions specifically designed to extract symbolic chains.

Definition 7 (Symbolic chains $\mathbb{Q}$). A symbolic chain $Q \in \mathbb{Q}$ is a composition of a typestate (Definition 1) and operation sequences (Definition 2). We use $\mathrm{ts}(Q)$ and $\mathrm{os}(Q)$ to represent the typestate and the operation sequences within the symbolic chain Q respectively. We use QS to represent a set of symbolic chains. The path-insensitive $\hat{\Pi}$-extraction can be obtained within the symbolic chains coupled with an error typestate, i.e., $\hat{\Pi} = \bigcup_{\langle T_e, P\rangle \in \mathbb{Q}} P$.

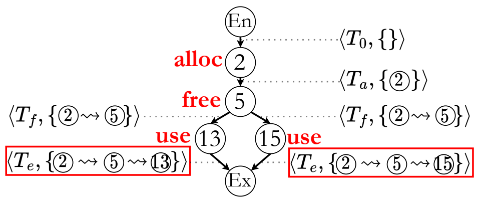
Flow functions for symbolic chains. Equation 2 shows the flow functions designed for symbolic chains. F_m joins the symbolic chains from the predecessors of a control flow joint point. When encountering program branches, F_b does not perform expensive path feasibility checks and simply transfers the input chains QS to the output. F_o updates the typestate and/or the operation sequences within the symbolic chains QS for a given control flow node.

$$\begin{aligned}\mathsf{F_m}(\ell, QS_1, QS_2) &= \alpha_{os}(QS_1 \cup QS_2)\\ \mathsf{F_b}(\ell, QS) &= QS\\ \mathsf{F_o}(n, QS) &= \alpha_{os}(\{\mathsf{f_o}(n, Q) \mid Q \in QS\})\end{aligned} \tag{2}$$

The grouping function α_{os} merges the operation sequences with equivalent typestate, which is defined as follows:

$$\alpha_{os}(QS) = \{\langle T, \bigcup_{Q \in QS[T]} \mathrm{os}(Q)\rangle \mid T \in \mathbb{T} \wedge QS[T] \neq \varnothing\} \tag{3}$$

where $QS[T] = \{Q \mid Q \in QS \wedge T \in \mathrm{ts}(Q)\}$.



En
$\langle T_0, \{\}\rangle$
alloc
2
$\langle T_a, \{②\}\rangle$
free
5
$\langle T_f, \{② \rightsquigarrow ⑤\}\rangle$
$\langle T_f, \{② \rightsquigarrow ⑤\}\rangle$
use
13
15
use
$\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑬\}\rangle$
$\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑮\}\rangle$
Ex

Fig. 5. An example of $\hat{\Pi}$-extraction.

Note that, prior to collecting symbolic chains, we condense G_{ICFG} by only retaining those nodes in Σ that involve objects aliased with the object of interest. This effectively trims the size of G_{ICFG}, thereby enhancing the overall efficiency of the symbolic chains collection algorithm.

Example 4. Figure 5 illustrates how to collect symbolic chains ($\langle T_e$, $\{② \rightsquigarrow ⑤ \rightsquigarrow ⑬\}\rangle$ and $\langle T_e$, $\{② \rightsquigarrow ⑤ \rightsquigarrow ⑮\}\rangle$) by revisiting our code example in Figure 4. The condensed G_{ICFG} becomes very straightforward because it exclusively encompasses nodes in Σ that are related to the object p, i.e., ②, ⑤, ⑬ and ⑮. The initial symbolic chain is $\langle T_0, \{\}\rangle$. The typestate T_0 changes to T_a at ② so ② is appended to the symbolic chain to generate $\langle T_a, \{②\}\rangle$. The symbolic chain passing through ⑤, ⑬ and ⑮ also observe typestate transition, producing $\langle T_f$, $\{② \rightsquigarrow ⑤\}\rangle$, $\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑬\}\rangle$ and $\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑮\}\rangle$ respectively. We check the symbolic chains at each error triggering point (⑬ and ⑮ in our example) and collect all the multi-point markers $② \rightsquigarrow ⑤ \rightsquigarrow ⑬$ and $② \rightsquigarrow ⑤ \rightsquigarrow ⑮$ residing in the symbolic chains coupled with an error typestate, i.e., $\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑬\}\rangle$ and $\langle T_e, \{② \rightsquigarrow ⑤ \rightsquigarrow ⑮\}\rangle$. These markers are used to guide later multi-point slicing.

Lemma 4.1 (Correctness of $\hat{\Pi}$-extraction). *Given a G_{ICFG} and an automaton τ, $\hat{\Pi}$-extraction based on the symbolic chains $\mathbb{Q}$ obtains all multi-point markers on G_{ICFG}.*

Proof Sketch. The symbolic chains are collected using a conservative path-insensitive typestate analysis built on the IFDS [30], and the flow functions in Equation 2 strictly follow Definitions 2 and 3 to derive the operation sequences in $\mathbb{Q}$. Therefore, it can identify all possible vulnerable operation sequences (multi-point markers). □

Complexity. The time complexity is $O(|\hat{\Pi}||E^-||\mathbb{T}|^3)$, where $|E^-| < |E|$ is the number of edges in the condensed G_{ICFG} and $|\mathbb{T}|$ is a constant representing the size of all the typestates. $O(|E^-||\mathbb{T}|^3)$ is the time complexity of IFDS [60] and each edge stores at most $|\hat{\Pi}|$ markers.

4.2 Tempo-Spatial Multi-Point Slicing

Given the multi-point information in the path-insensitive $\hat{\Pi}$, tempo-spatial multi-point slicing effectively collects only necessary temporal and spatial program dependencies essential for PSTA. The expensive PSTA can then concentrate on the partial and relevant program slice to derive path-sensitive Π. Next, we detail the multi-point slicing technique, comprising a temporal multi-point slicing TMS (Section 4.2.1) which captures the temporal orders of $\hat{\Pi}$; and a spatial multi-point slicing SMS (Section 4.2.2) which includes control and data-dependencies within $\hat{\Pi}$.

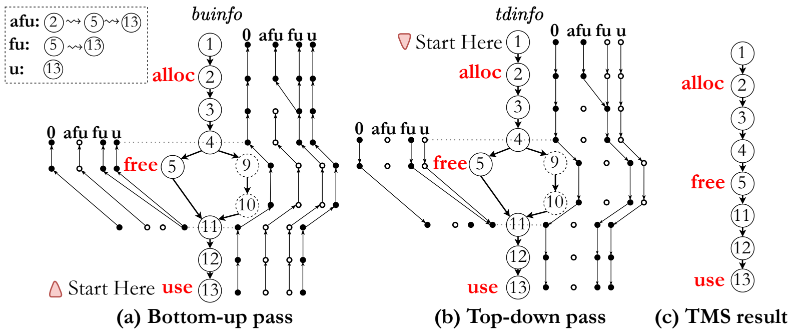


Fig. 6. An example of temporal multi-point slicing.

4.2.1 *Temporal multi-point slicing (TMS).* TMS aims to collect the program instructions from all realizable control flows with each containing at least one $\hat{\pi} \in \hat{\Pi}$ in the temporal execution order inherent in $\hat{\pi}$. Every control flow edge is treated non-deterministically, which ensures that any program instructions not included within the temporal slice are definitively excluded from the scope of runtime executions that pass through at least one $\hat{\pi} \in \hat{\Pi}$. Consequently, these program instructions not in the temporal slice are skipped during PSTA.

TMS starts with a bottom-up pass and finishes with a top-down pass. Both passes are performed (fully) context-sensitively based on the IFDS [51, 60]. The bottom-up pass starts at the error-triggering point and computes backwards dataflow facts, denoted as *buInfo*, for each node. The element in *buInfo* characterizes the partially anticipable suffixes associated with the marker. The top-down pass starts at program entry and computes forwards dataflow facts, denoted as *tdInfo*, for each node n, which signifies the set of suffixes to represent implicitly the fact that their corresponding prefixes are partially available at node n. In determining whether a node n is incorporated into the temporal slice V_{TMS}, a pivotal criterion is applied: the two sets of data flow facts at node n must possess shared data flow facts, i.e., $buInfo(n) \cap tdInfo(n) \neq \emptyset$.

Example 5. Figure 6 illustrates TMS on the multi-point marker ②⇝⑤⇝⑬ by revisiting the example in Figure 4 (⑥, ⑦, ⑧, ⑮ and ⑯ are omitted for simplicity). As shown in the TMS result in Figure 6(c), ⑨ and ⑩ are excluded because no runtime executions containing ②⇝⑤⇝⑬ would pass ⑨ or ⑩. There are a total of three non-empty dataflow facts: $\{\texttt{afu}, \texttt{fu}, \texttt{u}\}$, where a, f and u represent ②, ⑤ and ⑬ respectively. The solid dots in Figure 6(a) and Figure 6(b) represent reachable dataflow facts in *buinfo* and *tdinfo*, respectively. The bottom-up pass shown in Figure 6(a) starts at ⑬ and backward traverses the exploded G_{ICFG}. The suffixes u, fu and afu are partially anticipable at ⑬, ⑤ and ② respectively. Thus, we have $buInfo(13) = \{\texttt{u}\}$, $buInfo(5) = \{\texttt{u}, \texttt{fu}\}$ and $buInfo(2) = \{\texttt{u}, \texttt{fu}, \texttt{afu}\}$. The top-down pass in Figure 6(b) starts at the program entry ①. The prefix a appears at ②. Thus, we have $tdInfo(3) = \{\texttt{fu}\}$, indicating that the prefix a is partially available at ③. The temporal slice is obtained by intersecting *buInfo* and *tdInfo* per node. ⑨ and ⑩ are excluded because they have no intersected dataflow facts.

Lemma 4.2 (Correctness of TMS). *Given a G_{ICFG} and multi-point markers $\hat{\Pi}$, after graph simplification via TMS, the simplified ICFG preserves all $\hat{\pi} \in \hat{\Pi}$.*

Proof Sketch. For each multi-point marker in $\hat{\Pi}$, TMS is able to collect the instructions on all possible control flows that contain the marker. Therefore, all $\hat{\pi} \in \hat{\Pi}$ are preserved within the simplified graph. □

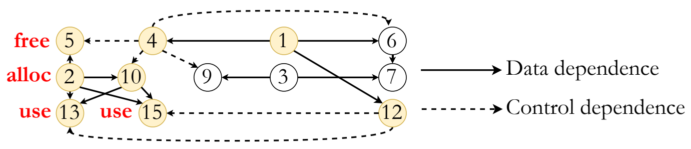


Fig. 7. An example of spatial multi-point slicing.

Complexity. The cost of TMS for one sequence is the complexity of the IFDS [60], which is $O(|E||\mathbb{T}|^3)$, where $|E|$ is the number of edges in G_{ICFG} and $|\mathbb{T}|$ is a constant representing the size of all the typestates. Therefore, the total complexity is $O(|\hat{\Pi}||E||\mathbb{T}|^3)$.

4.2.2 Spatial multi-point slicing (SMS). SMS works on the program dependence graph, as defined in Definition 8, aiming to acquire the control and data dependencies of the program points in $\hat{\Pi}$, which are essential to determine the feasibility of $\hat{\Pi}$. The program instructions that are not control or data dependent on $\hat{\Pi}$ are irrelevant and excluded.

Definition 8 (Program dependence graph G_{PDG})**.** G_{PDG} is a directed graph that captures the control and data dependencies of a program. We use E_c and E_d to represent control and data dependencies, respectively. For a control dependence edge $n_i \longrightarrow n_j \in E_c$, n_j is reachable iff the control-flow guard at n_i is satisfied. A data dependence edge $n_i^{'} \longrightarrow n_j^{'} \in E_d$ means that the variable definition at $n_i^{'}$ is used at $n_j^{'}$. The control dependence edges set E_c can be established in linear time by using an augmented postdominator tree (APT) [58]. The data dependence E_d is built upon the partial SSA form by considering program control flows and a pre-computed points-to result [41, 70].

$$[\texttt{INIT}]\,\frac{\pi \in \hat{\Pi} \quad n \in \pi}{\{n\} \subseteq V_{\text{SMS}}} \qquad [\texttt{CONTROL}]\,\frac{n \in V_{\text{SMS}} \quad (n_c, n) \in E_c}{\{n_c\} \subseteq V_{\text{SMS}}} \qquad [\texttt{DATA}]\,\frac{n \in V_{\text{SMS}} \quad (n_d, n) \in E_d}{\{n_d\} \subseteq V_{\text{SMS}}}$$

Fig. 8. Inference rules for spatial multi-point slicing.

Figure 8 formalizes the inference rules of SMS including `INIT`, `CONTROL` and `DATA`. The `INIT` rule initializes V_{SMS} using the instructions in $\hat{\Pi}$. The `CONTROL` rule aims to retrieve all the control dependencies of the instructions in $\hat{\Pi}$, while the `DATA` rule aims to collect all the data dependencies. `CONTROL` and `DATA` rules work collectively until no new instructions are discovered.

Example 6. Figure 7 shows a running example of the spatial multi-point slicing using the code example from Figure 4. Data and control dependencies are annotated with solid and dotted lines respectively. For example, ⑩ is data dependent on ② because the variable p used at ⑩ is defined at ②. ⑤ is control dependent on ④ because the condition $t > 1$ at ④ determines the execution of ⑤. We perform spatial multi-point slicing by recursively traversing backward against the control and data dependencies starting from the nodes in $\hat{\Pi}$, i.e., ②, ⑤, ⑬ and ⑮. As a result, the spatial slice consists of ①, ②, ④, ⑤, ⑩, ⑫, ⑬ and ⑮.

Lemma 4.3 (Correctness of SMS). *Given a G_{ICFG} and multi-point markers $\hat{\Pi}$, after graph simplification via SMS, the simplified ICFG preserves all the control and data dependencies of the instructions within $\hat{\Pi}$.*

Proof Sketch. The G_{PDG} (Definition 8) is sound because both control and data dependencies are soundly built. Therefore, SMS is correct because it is an inclusion-based algorithm by including all possible control and data dependencies related to $\hat{\Pi}$ according to the inference rule in Figure 8. □

Complexity. The cost of SMS is $O(|V| + |E|)$, which is the cost of a standard depth-first traversal algorithm (each node can only be visited once). V and E are the nodes and edges of G_{ICFG}.

Table 2. The statistics of the open-source projects. *#LOI* denotes the number of lines of LLVM instructions. *#Method* and *#Call* are the numbers of functions and method calls. *#Ptr* and *#Obj* represent the quantities of pointer variables and memory objects. $|V|$ and $|E|$ indicate the numbers of ICFG nodes and ICFG edges.

Project	*#LOI*	*#Method*	*#Call*	*#Ptr*	*#Obj*	$\|V\|$	$\|E\|$
YAJL	20,592	151	561	10,197	208	9,253	9,922
gzip	33,058	195	459	19,264	457	16,889	16,582
MP4v2	39,178	601	610	15,925	1,991	15,595	16,733
bzip2	48,181	116	250	28,710	263	26,220	25,912
darknet	159,205	985	9,776	136,510	2,550	136,094	147,852
nasm	186,935	652	7,435	121,836	3,736	79,330	81,638
tmux	446,626	1,967	22,369	187,315	3,879	162,879	178,924
Teeworlds	529,737	2,306	28,267	292,621	5,754	251,356	246,029
NanoMQ	788,967	3,235	47,646	379,798	30,838	358,312	443,670
redis	1,363,507	6,314	68,664	708,251	13,958	589,019	704,356
Total	3,615,986	165,22	186,037	1,900,427	63,634	1,644,947	1,871,618

4.2.3 *Putting it all together.*

Given the temporal slice V_{TMS} and spatial slice V_{SMS}, we compact the original G_{ICFG} into a much smaller graph G'_{ICFG} by retaining only the nodes in the intersected slice $V_{\text{TMS}} \cap V_{\text{SMS}}$.

Theorem 4.4 (Correctness of FGS). *Given an interprocedural control flow graph G_{ICFG} and an automaton τ, FGS is able to generate a smaller graph G'_{ICFG} which ensures that G_{ICFG} and G'_{ICFG} are typestate reachability equivalent.*

Proof Sketch. According to Lemma 4.1, all the multi-point markers $\hat{\Pi}_\tau$ in G_{ICFG} are preserved within G'_{ICFG}, i.e., $(\forall \hat{\pi} \in \hat{\Pi}_\tau : \hat{\pi} \in \hat{\Pi}'_\tau) \wedge (\forall \hat{\pi}' \in \hat{\Pi}'_\tau : \hat{\pi}' \in \hat{\Pi}_\tau) \wedge (\hat{\pi} \notin \hat{\Pi}_\tau \Leftrightarrow \hat{\pi} \notin \hat{\Pi}'_\tau)$. Concerning the feasibility of $\hat{\Pi}$, our multi-point slicing is able to include all the required program dependencies used to determine the feasibility of $\hat{\Pi}$. This is because our approach, which computes the intersection of TMS and SMS, can correctly include all the control and data dependencies residing in all possible temporal executions with each containing at least one $\hat{\pi} \in \hat{\Pi}$, given the correctness of TMS (Lemma 4.2) and SMS (Lemma 4.3). Therefore, the feasible sequences Π_τ and Π'_τ under path-sensitive analysis are equivalent, i.e., $(\forall \pi \in \Pi_\tau : \pi \in \Pi'_\tau) \wedge (\forall \pi' \in \Pi'_\tau : \pi' \in \Pi_\tau) \wedge (\pi \notin \Pi_\tau \Leftrightarrow \pi \notin \Pi'_\tau)$. This demonstrates that G_{ICFG} and G'_{ICFG} are typestate reachability equivalent. □

Speedup by FGS. The PSTA algorithm, as described in [30], exhibits a time complexity of $O(H(K + J + Q)|\mathbb{T}|(|E||\mathbb{T}| + CS|\mathbb{T}|^2))$. Here, each element can become less precise at most H times, K corresponds to a highly complex function for SMT solving, J and Q are the cost of the join operation and the equality operation on execution states, respectively. $\mathbb{T}$ signifies the domain of typestate, E stands for G_{ICFG} edges, and CS indicates the number of call sites in the program. FGS can improve the efficiency by simplifying G_{ICFG}, i.e., reducing sizes of $|E|$ and CS, which contributes to the reduction of execution states including H, J and Q and reducing the cost K of SMT solving.

5 EVALUATION

In this section, we perform an ablation analysis to gain deeper insights into how the multi-point slicing of FGS influences its overall performance. Moreover, we also show the effectiveness of FGS for analyzing real-world programs and its practicality for detecting vulnerabilities including memory leaks, use-after-frees, double-frees, and null dereferences. We assess the effectiveness of FGS by comparing it with six popular open-source tools: IKOS [17], ClangSA (Clang Static Analyzer) [53], Saber [71], Cppcheck [25], Infer [43] and Sparrow [56].

5.1 Datasets and Implementation

Datasets. We evaluate FGS on (1) a benchmark comprising 846 vulnerabilities from NIST [55], which includes memory leaks, double-frees, use-after-frees and null dereferences. (2) ten open-source C/C++ projects across a variety of different domains: `YAJL` [10] (JSON parsing library), `gzip` [3] (data compression program), `MP4v2` [4] (MP4 file library), `bzip2` [1] (data compressor), `darknet` [2] (neural network framework), `nasm` [6] (assembler), `tmux` [9] (terminal multiplexer), `Teeworlds` [8] (online multiplayer game), `NanoMQ` [5] (MQTT broker for IoT edge platform) and `redis` [7] (in-memory database). These projects encompass a diverse range of applications, allowing us to assess FGS's effectiveness in real-world scenarios.

Implementation. The experiments were conducted on an Ubuntu 18.04 server with an eight-core 2.60GHz Intel Xeon CPU and 128 GB memory. We constructed the interprocedural control and value flow graphs based on LLVM-IR [48, 70] (using LLVM version 14.0.0). The LLVM-IR represents program functions as global variables, which are further modeled as address-taken variables. The abstract domain in the execution state (Definition 4) is a combined domain of memory addresses for pointer analysis and the constant propagation used in `ESP` [30] for numerical analysis. The abstract value representation is implemented using Z3 expressions [31]. The callgraph of a program is built on a pre-computed Andersen's points-to results [11] to resolve indirect calls. For the baselines, namely IKOS, ClangSA, Saber, Cppcheck, Infer, and Sparrow, we utilize their open-source implementations with default settings to detect memory leaks, double-frees, use-after-frees and null dereferences.

5.2 Research Questions

Our evaluation aims to answer the following research questions:

RQ1 **How do different components impact the overall performance of FGS?** We want to investigate how different slicing methods influence the effectiveness and efficiency of FGS.
RQ2 **Does FGS outperform popular static tools for bug detection?** We aim to explore whether FGS can detect more bugs with lower false alarm rates than the state-of-the-art on detecting existing bugs using the NIST benchmark with ground truths.
RQ3 **Can FGS find bugs with lower false positives efficiently in real-world projects?** We would like to examine the effectiveness (in terms of true and false positives) and efficiency (in terms of running time and memory usage) of FGS on real-world popular applications.

5.3 Impact of Graph Simplification and Ablation Analysis (RQ1)

In this section, we focus on examining the impacts of various components on the performance of FGS. Firstly, we compare FGS with a path-insensitive typestate analysis on the NIST dataset to understand the precision improvement by path sensitivity. Next, we show the time proportion of different phases of FGS. Subsequently, we present the graph simplification result to understand the statistics of graph simplification achieved by FGS. Finally, we conduct an ablation analysis to comprehend the influences of different slicing methods.

Comparison with path-insensitive typestate analysis. For path-insensitive typestate analysis, we consider all G_{ICFG} edges as executable in a non-deterministic manner. While path-insensitive typestate analysis is capable of uncovering all the bugs detected by FGS, it also tends to produce over 50% more false positives in comparison to FGS. This underscores the significance of employing path-sensitive analysis in enhancing the precision of typestate analysis.

Proportions of analysis time. Figure 9 shows the time proportions of different phases in FGS, including graph generation, multi-point markers extraction, TMS, SMS and main PSTA phase on

Table 3. Graph simplification result. $|V|$, $|V'|$, $|V_{\mathrm{TMS}}|$ and $|V_{\mathrm{SMS}}|$ represent the number of nodes in G_{ICFG}, G'_{ICFG}, temporal slice and spatial slice, respectively. $\#Call$ and $\#Call'$ represent the number of calling contexts of G_{ICFG} and G'_{ICFG}. $|E|$ and $|E'|$ represent the number of edges in G_{ICFG} and G'_{ICFG}.

Project	$\|V\|$	$\|V'\|$	$\|V_{\mathrm{TMS}}\|$	$\|V_{\mathrm{SMS}}\|$	$\#Call$	$\#Call'$	$\|E\|$	$\|E'\|$
`darknet`	136,094	1,791	5,523	1,928	9,776	93	147,852	1,802
`nasm`	79,330	24,946	38,081	26,604	7,435	2,317	81,638	26,034
`tmux`	162,879	2,671	4,273	3,693	22,369	205	178,924	2,810
`Teeworlds`	251,356	565	1,380	1,875	28,267	40	246,029	578
`NanoMQ`	358,312	62,543	102,118	118,663	47,646	5,801	443,670	61,696
`redis`	589,019	87,446	102,416	111,041	68,664	17,844	704,356	240,956

Table 4. Ablation analysis results. The "–" in the Time columns indicates a running time of more than 48 hours. FGS-TMS and FGS-SMS represent the versions of FGS using only temporal slicing and spatial slicing respectively. FGS-Base represent the version of FGS without slicing.

Project	FGS		FGS-TMS		FGS-SMS		FGS-Base	
	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)	Time (secs)	Mem (MB)
`darknet`	750	2,104	2,542	2,785	817	2,784	81,422	34,244
`nasm`	894	2,482	1,681	4,132	940	3,413	111,750	31,781
`tmux`	1,932	5,251	5,782	9,064	3,102	7,223	–	–
`Teeworlds`	407	4,320	1,424	5,014	1,700	6,062	–	–
`NanoMQ`	8,722	10,176	25,890	13,600	29,100	18,424	–	–
`redis`	14,266	58,231	23,146	78,131	31,103	98,064	–	–

`darknet`, `nasm`, `tmux`, `Teeworlds`, `NanoMQ` and `redis`. It is evident that the main phase accounts for the largest proportion of the overall runtime. Our graph simplification techniques, including $\hat{\Pi}$-extraction, TMS and SMS, collectively consume less than 10% of the total execution time. This demonstrates that our graph simplification introduces relatively minor overhead when compared to the time expended in the main analysis phase.

Graph simplification statistics. Table 3 shows the statistics detailing the number of nodes, edges, and calling contexts before and after graph simplification. On average, FGS reduces the number of nodes by 89%, edges by 86%, and calling contexts by 88%. These results demonstrate the effectiveness of our graph simplification approach in eliminating a substantial number of irrelevant nodes. Furthermore, both TMS and SMS contribute to creating more simplified graphs. The intersection of these two techniques produces an even more concise graph.

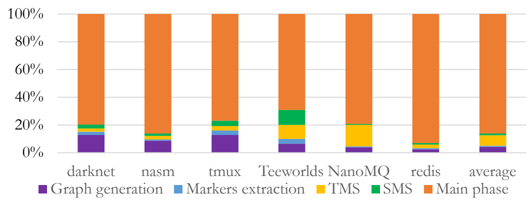


Fig. 9. The proportions of different phases of FGS.

Ablation analysis. In our ablation analysis, as detailed in Table 4, we compared the runtime and memory costs of various configurations of FGS. These configurations include FGS with only temporal multi-point slicing (FGS-TMS), FGS with only spatial multi-point slicing (FGS-SMS), and FGS without any slicing (FGS-Base). FGS-Base fails to complete the analysis within a 48-hour time budget when applied to projects like `tmux`, `Teeworlds`, `NanoMQ`, and `redis`. FGS, compared to FGS-Base, exhibited an average speedup of 116× and achieved a notable reduction in memory usage by 93%. Additionally, when comparing FGS with FGS-TMS and FGS-SMS, we observed notable improvements in both runtime and memory consumption. For instance, on the `Teeworlds` project, FGS was three times faster than FGS-TMS and reduced memory consumption by approximately 29% compared to FGS-SMS. These outcomes align with the benchmark statistics presented in Table 3,

Table 5. Comparing true positives (#*TP*) and false positives (#*FP*) with six tools using the NIST benchmark. The "–" means that the detection of specific vulnerabilities is not supported by the corresponding tools.

Category	IKOS		ClangSA		Saber		Cppcheck		Infer		Sparrow		FGS		Ground Truth
	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	#*TP*	#*FP*	
Memory leak	–	–	128	112	200	126	0	0	126	162	–	–	**228**	**0**	228
Double-free	228	18	156	20	204	20	84	144	–	–	–	–	**228**	**0**	228
Use-after-free	–	–	40	0	–	–	0	0	0	0	–	–	**138**	**0**	138
Null dereference	234	18	216	24	234	18	108	18	134	82	228	18	**252**	**0**	252
Total	462	36	540	156	638	164	192	162	260	244	228	18	**846**	**0**	846

further emphasizing the efficiency enhancements introduced by the composition of temporal and spatial multi-point slicing in FGS.

5.4 Analysis Results using NIST Benchmark (RQ2)

Table 5 compares the performance of six state-of-the-art open-source static analysis tools (IKOS, ClangSA, Saber, Cppcheck, Infer and Sparrow) on the pre-labeled NIST benchmark. We show true positives and false positives of FGS and our baselines on detecting four vulnerability categories: memory leak (CWE-401) [26], double-free (CWE-415) [27], use-after-free (CWE-416) [28] and null dereference (CWE-476) [29]. The last row of the table displays the number of labeled bug ground truth for each category.

Comparison results. FGS effectively identifies all four types of vulnerabilities, setting it apart from IKOS and Sparrow, which lack the ability to detect memory leaks. Additionally, IKOS, Saber, and Sparrow do not provide support for use-after-free detection. Infer and Sparrow also fall short in identifying double-frees. This highlights FGS's capability to detect a wide range of vulnerabilities by leveraging typestate analysis. Furthermore, FGS achieves the best performance by precisely identifying all the bugs, achieving a 100% true positive rate, and reporting no false alarms, outperforming all six baseline tools. For memory leak detection, our baseline tools exhibit an average precision of merely 53%, whereas FGS consistently records no false memory leaks. With regard to double-free detection, FGS demonstrates its ability to reduce an average of 50 false positives out of the 228 true bugs identified by our baseline tools. In terms of use-after-frees, although our baselines also report no false alarms, they can only identify a limited number of bugs. For example, ClangSA only uncovers 40 out of the 138 use-after-free bugs, while the other tools fail to detect any. Regarding null dereference detection, our baseline detectors, on average, report a lower precision of 86% compared to FGS.

Result Analysis. In Figure 10, we present three code scenarios, illustrating the factors contributing to FGS's superior performance compared to the other tools on the NIST dataset. FGS benefits from its precision-preserving path-sensitive analysis upon simplified ICFG. Figure 10(a) shows an example to demonstrate FGS's ability to capture branch correlations. The code fragment is safe, but IKOS, ClangSA, Saber and Cppcheck report a false double-free alarm at ℓ_{10}. At ℓ_2, the variable `cond` derives its value from the function `rand()%2`, which consistently returns either true or false. FGS can precisely distinguish the conflicting conditions $\ell_3 \xrightarrow{cond} \ell_5$ and $\ell_9 \xrightarrow{\neg cond} \ell_{10}$. Consequently, it deduces that the invocation of the `free()` function at both ℓ_5 and ℓ_{10} cannot occur sequentially. Figure 10(b) presents another instance exemplifying FGS's prowess in capturing branch correlations. Despite the code fragment being safe, our baseline detectors (ClangSA, Saber and Infer) incorrectly report a false alarm concerning a memory leak. At ℓ_1, the variable `var` receives an initial value generated randomly through `rand()`, represented as c. As the execution proceeds to ℓ_6, `var` undergoes a self-increment operation, i.e., `var` changes to $c + 1$. FGS can precisely distinguish the conflicting

```
1   char* data;
2   bool cond = rand() % 2;
3   if(cond) {
4     data = malloc(...);
5     free(data);
6   } else {
7     data = malloc(...);
8   }
9   if(cond) { ; }
10  else { free(data); }
```

(a) Capturing branch correlations.

```
1   int var = rand();
2   int* data;
3   if(var == 5) {
4     data = malloc(...);
5   } else { ; } // do nothing
6   ++var;
7   ...
8   if(var != 5) {
9     free(data);
10  } else { ; } // do nothing
```

(b) Capturing branch correlations.

```
1   int* data = malloc(...);
2   int* res;
3   ...
4   if(rand() % 2) {
5     res = data;
6   } else {
7     free(data);
8     res = 0;
9   }
10  free(res);
```

(c) Path-sensitive aliasing.

Fig. 10. Code scenarios to explain the precision of FGS.

Table 6. Comparing FGS with six open-source tools 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	IKOS				ClangSA				Saber				Cppcheck				Infer				Sparrow				FGS			
	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)
YAJL	4	15	2895	4822	0	0	4	111	3	22	2	206	1	5	1	13	2	15	13	133	3	86	6	59	5	0	2	168
gzip	4	4	3114	4949	0	1	27	151	0	4	18	179	1	3	89	35	1	17	36	177	1	22	14	89	4	0	18	835
MP4v2	2	1	3684	6215	0	0	11	145	3	24	3	380	0	6	56	38	4	28	496	426	1	20	214	231	5	0	2	344
bzip2	0	0	3690	6809	0	6	16	181	0	2	18	179	0	0	3	17	0	37	53	271	0	0	77	148	1	0	9	280
darknet	19	75	5216	8622	11	39	75	301	20	300	245	1145	2	24	11	55	12	104	1185	612	25	10	951	954	30	7	750	2104
nasm	2	8	5007	9951	2	7	180	515	2	102	572	2258	0	1	1	76	1	16	621	919	2	9	942	1132	3	1	894	2482
tmux	4	29	11325	38366	6	12	409	799	4	160	597	3882	0	0	61	39	2	34	693	637	3	12	1036	1894	5	1	1932	5251
Teeworlds	8	8	13569	40368	0	0	83	654	10	50	88	1877	1	4	2	54	6	48	267	449	5	24	1593	2984	12	2	407	4320
NanoMQ	17	29	9344	63068	0	0	52	555	10	426	1421	7613	5	54	111	40	18	74	910	555	6	354	1642	3125	31	11	8722	10176
redis	–	–	–	–	0	23	502	1499	7	141	8775	16752	0	1	637	123	1	51	2699	1655	1	149	2654	9211	9	1	14266	58231
Total	60	169	57844	183170	19	88	1359	4911	59	1231	11739	34471	10	98	972	490	47	424	6973	5834	47	686	9129	19827	105	23	27002	84191

conditions $\ell_3 \xrightarrow{c\equiv 5} \ell_4$ and $\ell_8 \xrightarrow{c\equiv 4} \ell_{10}$. Consequently, it effectively concludes that no memory leaks along the branch $\ell_8 \xrightarrow{c\equiv 4} \ell_{10}$. Figure 10(c) provides an example that undergoes FGS's ability in path-sensitive aliasing. There are no double-frees in this code snippet. However, IKOS, ClangSA and Cppcheck raise a false positive at ℓ_{10}. In contrast, FGS precisely discerns that the value attributed to `res` at ℓ_{10} from ℓ_8 does not share an aliasing relationship with `data` at ℓ_1. This distinction arises due to FGS's capacity to independently address the aliasing relations across different paths.

5.5 Bugs in Real-World Projects (RQ3)

Table 6 compares the true positives, false positives, running time and memory costs of FGS with six baseline tools (IKOS, ClangSA, Saber, Cppcheck, Infer and Sparrow) across ten real-world popular applications. Overall, FGS achieves the best performance on reporting 128 bugs, including 105 true bugs and 23 false positives after a rigorous manual examination.

Comparison results and analysis. FGS can find all the bugs reported by our baseline detectors with 61% more true positives on average. Meanwhile, FGS exhibits a higher precision at about 82%, compared to a precision of merely 12% for our baselines. FGS finds more bugs because it effectively handles some hard code features, such as interprocedural analysis, loop handling, and accurate external API modeling. This gives better results than other tools in uncovering more bugs. The key factor behind FGS's lower false positive rate lies in its scalable yet precision-preserving path-sensitive analysis on top of simplified ICFG. This is particularly clear when comparing FGS with a path-insensitive typestate analysis, which employs similar vulnerability checkers but lacks path-sensitivity. Our observations reveal that the path-insensitive approach yields significantly more false positives (Section 5.3). Furthermore, FGS showcases commendable efficiency in terms of

```
1   char *fgetl(FILE *fp) {
2     ...
3     char *line = malloc(...);
4     if(!fgets(...)){
5       free(line);
6       return 0;
7     }...
8     return line;
9   }
10  char **read_tokens(...) {
11    free(fgetl(fp));
12  }
```

(a) A safe free in darknet.

```
1   void RateConvert(int id) {
2     CSample *p = &m_aSamples[id];
3     ...
4     int N = ...;
5     short *data = mem_alloc(...);
6     ...
7     for(int i = 0; i < N; i++) {
8         ...
9         data[i] = p->m_pData[f];
10    }
11    ...
12  }
```

(b) A null dereference in Teeworlds.

```
1   void u_ranges(TYPE *frs, ...) {
2     for(TYPE *fr=frs.head();...) {
3         if (fr->s != s) continue;
4         free(fr);
5         ...
6     }
7   }
8   void centre(TYPE *frs, ...) {
9     u_ranges(frs, ...);
10    ...
11    u_ranges(frs, ...);
12  }
```

(c) A use-after-free in tmux.

Fig. 11. A false positive eliminated by FGS and two bugs found by FGS simplified from real-world projects.

both runtime and memory usage, managing to complete analyses for all the projects. It is worth noting that although FGS may demand more time for code analysis compared to ClangSA, Saber, Cppcheck, Infer, and Sparrow, its superior ability to uncover a multitude of many types of real bugs, coupled with its substantially reduced false positive rate, positions it as a highly valuable tool. Thus, while there might be a modest increase in analysis time, the tangible benefits derived from its bug detection capabilities far outweigh the incurred time overhead.

Case studies. Figure 11 depicts three real-world scenarios to demonstrate the effectiveness of FGS in detecting software vulnerabilities. For illustration purposes, we only show the essential parts relevant to the vulnerability. Figure 11(a) shows a safe code snippet extracted from the darknet project, wherein FGS precisely eliminates a false double-free alarm reported by our baselines (IKOS, ClangSA and Cppcheck). At ℓ_{11}, the program releases the memory returned by the function fgetl. Although the heap object line allocated at ℓ_3 is released at ℓ_5, the return variable at ℓ_6 is not aliased with line along this path. Therefore, the free operation at ℓ_{11} is safe. Figure 11(b) presents a null dereference detected by FGS in the Teeworlds project, which is missed by Cppcheck. The problematic scenario involves the pointer data allocated at ℓ_5, which potentially holds a null value and is subsequently dereferenced at instruction ℓ_9. Figure 11(c) shows a use-after-free bug simplified from tmux. The function centre calls the function u_ranges twice at ℓ_9 and ℓ_{11}. The initial call releases the memory pointed by fr at ℓ_4. During the second call, at ℓ_3, the field access of fr yields a use-after-free bug, as the memory has already been deallocated in the prior call.

6 DISCUSSIONS

This section discusses the impact of SMT solving on PSTA in the analysis of extremely large programs, as well as the potential alternative uses of FGS.

SMT solving. The effectiveness and scalability of PSTA is affected by the precision and overhead of SMT solving. It is possible to apply a heavy-weight theorem prover to increase the precision of feasibility checking. On the other hand, it is an interesting topic to trade precision of SMT solving for efficiency when analyzing extremely large programs. Importantly, FGS is not dependent on the type of SMT solving and guarantees that the simplified and original graphs are typestate reachability equivalent (Definition 6).

Other technique uses. FGS is designed as a fast graph simplification analysis tool and can serve as a preprocessing step for any PSTA algorithms to boost their performance, thereby facilitating efficient end-to-end bug detection, such as detecting double-frees, use-after-frees, and null dereferences, as outlined by the typestate finite state automaton (Definition 1) in PSTA. FGS produces a compact code graph representation, containing only essential temporal and/or spatial information for later analysis. This enables the main analysis task to be focused on specific program points of interest without sacrificing precision. We believe the concept and methodology also apply to other clients, like symbolic execution [18], abstract interpretation [73] and code embedding [19, 21, 22, 69].

7 RELATED WORK

Typestate analysis. Typestate analysis [39, 68] is a widely-used program analysis technique to detect semantically undefined vulnerable execution sequences and improve software reliability [32, 38, 46, 54, 59]. There are several static approaches [14, 33, 37, 44, 45, 50, 74, 80] trying to improve the analysis precision using more precise alias analysis. For example, Li et al. [50] use an access-path-based approach to compute must-aliases. Our approach works on an orthogonal topic that speeds up PSTA, which can be easily adapted to various PSTA algorithms with diverse alias analyses. There is also another branch of hybrid typestate analysis approaches aiming to use static typestate analysis to guide dynamic analysis [15, 16, 35, 72, 78, 79]. For example, Wang et al. [72] present a typestate-guided fuzzer to detect UAF vulnerabilities. For those dynamic approaches, FGS can also work as a complement by enhancing the static typestate analysis.

Path-sensitive analysis. Path-sensitivity determines the feasibility of program paths, which is an essential property for enhancing the precision of static analysis [12, 34, 42, 64, 71]. However, it is challenging to achieve a meet-over-path solution because of the potential unbounded number of paths to be analyzed and the high computational overhead of path conditions. There are several studies proposed to reduce the trade-offs of path condition solving using condition simplification [75], summarization [77] or refinement [12, 23]. However, these works still suffer from scalability when analyzing large programs [65]. Recently, some approaches [63–66] introduce sparse analysis to path-sensitivity. However, these approaches are not applicable to typestate analyses that require tracking multiple program points by following the temporal order.

Graph simplification for static analysis. Graph simplification techniques have been widely used across various program analyses. For pointer analysis, a diverse set of techniques, as evidenced by Manuel et al. [36], Hardekopf and Calvin [40], and Rountev [62], have been employed to streamline constraint graphs utilized in the inclusion-based pointer analysis. For example, Hardekopf and Calvin [40] primarily focus on establishing relations such as pointer-equivalence and location-equivalence among variables, leading to the consolidation of equivalent nodes and consequent graph simplification. In the domain of CFL-reachability, Li et al. [52] propose an approach centered on eliminating graph edges that do not contribute to any InterDyck-paths. Meanwhile, Lei et al. [49] introduce criteria for identifying foldable node pairs with an RSM-based graph-folding algorithm. In contrast, our graph-simplification approach aims at boosting the efficiency of path-sensitive typestate analysis, distinguishing it from these aforementioned studies.

8 CONCLUSION

This paper introduces FGS, a novel approach to boosting the efficiency of path-sensitive typestate analysis (PSTA), focusing on accelerating the path-sensitive algorithm. FGS introduces a novel tempo-spatial multi-point slicing technique, which effectively reduces computational complexity and memory costs in PSTA by simplifying the ICFG while retaining vital temporal and spatial information. We have evaluated FGS using the NIST benchmark and ten real-world projects. Our experimental results demonstrate that our approach, on average, reduces the number of ICFG nodes by 89% and the calling context by 88%, leading to a 116× speedup and a 93% reduction in memory usage. Furthermore, FGS outperforms six popular static detectors in memory leak, double-free, use-after-free and null dereference detection. FGS achieves notably higher precision (up to 171%) and detects more true positives (up to 245%) on the NIST dataset. Across the ten large-scale projects, FGS identifies 105 real bugs with remarkable precision (82%), surpassing our baseline tools.

DATA AVAILABILITY STATEMENT

We have made our implementation publicly available at [20].

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. bzip2 and libbzip2. https://sourceware.org/bzip2
[2] 2023. Darknet - Open Source Neural Networks in C. https://github.com/pjreddie/darknet
[3] 2023. GNU Gzip. https://www.gnu.org/software/gzip
[4] 2023. MP4v2 - A C/C++ library to create, modify and read MP4 files. https://github.com/enzo1982/mp4v2/
[5] 2023. NanoMQ - An ultra-lightweight and blazing-fast MQTT broker for IoT edge. https://github.com/emqx/nanomq
[6] 2023. NASM, the Netwide Assembler. https://github.com/netwide-assembler/nasm/
[7] 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/
[8] 2023. Teeworlds - A retro multiplayer shooter. https://teeworlds.com/
[9] 2023. Tmux - tmux source code. https://github.com/tmux/tmux
[10] 2023. YAJL - A fast streaming JSON parsing library in C. https://github.com/lloyd/yajl
[11] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Cophenhagen. https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf
[12] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08).* ACM. https://doi.org/10.1145/1368088.1368118
[13] George Balatsouras and Yannis Smaragdakis. 2016. Structure-sensitive points-to analysis for C and C++. In *Static Analysis: 23rd International Symposium (SAS '16).* Springer. https://doi.org/10.1007/978-3-662-53413-7_5
[14] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular Typestate Checking of Aliased Objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07).* ACM. https://doi.org/10.1145/1297027.1297050
[15] Eric Bodden. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE '12).* ACM. https://doi.org/10.1145/1806799.1806805
[16] Eric Bodden, Patrick Lam, and Laurie Hendren. 2012. Partially Evaluating Finite-State Runtime Monitors Ahead of Time. *ACM Trans. Program. Lang. Syst.* (2012). https://doi.org/10.1145/2220365.2220366
[17] 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 (SEFM '14).* Springer, 271–277. https://doi.org/10.1007/978-3-319-10431-7_20
[18] 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. https://dl.acm.org/doi/10.5555/1855741.1855756
[19] Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wang, Zheng Zheng, and Yulei Sui. 2022. How About Bug-Triggering Paths? - Understanding and Characterizing Learning-Based Vulnerability Detectors. *TDSC* (2022). https://doi.org/10.1109/TDSC.2022.3192419
[20] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi- Point Slicing (Artifact). https://doi.org/10.5281/zenodo.11077099
[21] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *TOSEM* (2021). https://doi.org/10.1145/3436877
[22] 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. https://doi.org/10.1145/3533767.3534371
[23] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. 2013. BLITZ: Compositional Bounded Model Checking for Real-World Programs. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13).* IEEE Press. https://doi.org/10.1109/ASE.2013.6693074
[24] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77).* ACM. https://doi.org/10.1145/512950.512973

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

[26] CWE-401 2023. CWE-401: Missing Release of Memory after Effective Lifetime. https://cwe.mitre.org/data/definitions/401.html.

[27] CWE-415 2023. CWE-415: Double Free. https://cwe.mitre.org/data/definitions/415.html.

[28] CWE-416 2023. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.

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

[30] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation (PLDI '02)*. ACM. https://doi.org/10.1145/512529.512538

[31] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag. https://dl.acm.org/doi/10.5555/1792734.1792766

[32] Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM. https://doi.org/10.1145/378795.378811

[33] Dinakar Dhurjati, Manuvir Das, and Yue Yang. 2006. Path-Sensitive Dataflow Analysis with Iterative Refinement. In *Proceedings of the 13th International Conference on Static Analysis (SAS '06)*. Springer-Verlag. https://doi.org/10.1007/11823230_27

[34] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, Complete and Scalable Path-Sensitive Analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM. https://doi.org/10.1145/1375581.1375615

[35] Matthew B. Dwyer and Rahul Purandare. 2007. Residual Dynamic Typestate Analysis Exploiting Static Analysis: Results to Reformulate and Reduce the Cost of Dynamic Analysis. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM. https://doi.org/10.1145/1321631.1321651

[36] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM. https://doi.org/10.1145/277650.277667

[37] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective typestate verification in the presence of aliasing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, 133–144. https://doi.org/10.1145/1146238.1146254

[38] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2011. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM. https://doi.org/10.1145/1950365.1950394

[39] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* (2014). https://doi.org/10.1145/2629609

[40] Ben Hardekopf and Calvin Lin. 2007. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *Proceedings of the 14th International Conference on Static Analysis (SAS '07)*. Springer-Verlag. https://doi.org/10.1007/978-3-540-74061-2_17

[41] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO '11)*. IEEE, 289–298. https://doi.org/10.1109/CGO.2011.5764696

[42] William R. Harris, Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. 2010. Program Analysis via Satisfiability modulo Path Programs. *SIGPLAN Not.* (2010). https://doi.org/10.1145/1707801.1706309

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

[44] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. *Papaya: Global Typestate Analysis of Aliased Objects.* ACM. https://doi.org/10.1145/3479394.3479414

[45] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and Modular Resource Leak Verification *(ESEC/FSE 2021)*. ACM. https://doi.org/10.1145/3468264.3468576

[46] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2022. Accumulation Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP '22)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2022.10

[47] Ted Kremenek. 2008. Finding software bugs with the clang static analyzer. *Apple Inc* (2008), 2008–08. https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

[48] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO '04)*. https://doi.org/10.1109/CGO.2004.1281665

[49] Yuxiang Lei, Yulei Sui, Shin Hwei Tan, and Qirun Zhang. 2023. Recursive State Machine Guided Graph Folding for Context-Free Language Reachability. *Proc. ACM Program. Lang.*, Article 119 (2023). https://doi.org/10.1145/3591233

[50] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-Sensitive and Alias-Aware Typestate Analysis for Detecting OS Bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM. https://doi.org/10.1145/3503222.3507770

[51] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP'16)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2016.15

[52] Yuanbo Li, Qirun Zhang, and Thomas Reps. 2020. Fast Graph Simplification for Interleaved Dyck-Reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM. https://doi.org/10.1145/3385412.3386021

[53] LLVM Community. 2021. Clang Static Analyzer. https://clang-analyzer.llvm.org/.

[54] Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like Analysis of Multiple Interacting Objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM. https://doi.org/10.1145/1449764.1449792

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

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

[57] D.J. Pearce, P.H.J. Kelly, and C. Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM TOPLAS* (2007). https://doi.org/10.1145/1290520.1290524

[58] Keshav Pingali and Gianfranco Bilardi. 1995. APT: A data structure for optimal control dependence computation. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI '95)*. ACM. https://doi.org/10.1145/207110.207114

[59] Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. 2019. AuthCheck: Program-State Analysis for Access-Control Vulnerabilities. In *Formal Methods. FM 2019 International Workshops (FM '19)*. Springer-Verlag. https://doi.org/10.1007/978-3-030-54997-8_34

[60] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM. https://doi.org/10.1145/199448.199462

[61] Thomas Reps and Genevieve Rosay. 1995. Precise Interprocedural Chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '95)*. ACM. https://doi.org/10.1145/222124.222138

[62] Atanas Rountev and Satish Chandra. 2000. Off-Line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM. https://doi.org/10.1145/349299.349310

[63] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. 2020. Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM. https://doi.org/10.1145/3377811.3380346

[64] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM. https://doi.org/10.1145/3192366.3192418

[65] 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 (PLDI '21)*. ACM. https://doi.org/10.1145/3453483.3454086

[66] Qingkai Shi and Charles Zhang. 2020. Pipelining Bottom-up Data Flow Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, 835–847. https://doi.org/10.1145/3377811.3380425

[67] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM. https://doi.org/10.1145/1250734.1250748

[68] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12 (1986). https://doi.org/10.1109/TSE.1986.6312929

[69] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *OOPSLA* (2020). https://doi.org/10.1145/3428301

[70] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. ACM. https://doi.org/10.1145/2892208.2892235

[71] 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. https://doi.org/10.1145/2338965.2336784

[72] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd*

International Conference on Software Engineering (ICSE '20). ACM. https://doi.org/10.1145/3377811.3380386

[73] Cheng Xiao, Wang Jiawei, and Sui Yulei. 2024. Precise Sparse Abstract Execution via Cross-Domain Interaction. In *46th International Conference on Software Engineering (ICSE '2024)*. ACM/IEEE. https://doi.org/10.1145/3597503.3639220

[74] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. 2014. ARC++: Effective Typestate and Lifetime Dependency Analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM. https://doi.org/10.1145/2610384.2610395

[75] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, 351–363. https://doi.org/10.1145/1040305.1040334

[76] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*. ACM. https://doi.org/10.1145/3134600.3134620

[77] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Proceedings of the 35nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM. https://doi.org/10.1145/1328438.1328467

[78] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic Verification of Regular Properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM. https://doi.org/10.1145/3180155.3180227

[79] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press. https://doi.org/10.1109/ICSE.2015.80

[80] 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 (EuroSys '19)*. ACM. https://doi.org/10.1145/3302424.3303972

Received 2023-09-29; accepted 2024-01-23