International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2022, 12(3): 263–284, doi: 10.21655/ijsi.1673-7288.00286


International Journal of Software and Informatics

Research Article

# Counterexample-guided Spatial Flow Model Checking Methods for C Code

Yinbo Yu (于银菠), Jiajia Liu (刘家佳), Dejun Mu (慕德俊)

(School of Cyber security, Northwestern Polytechnial University, Xi'an 710072, China)
Corresponding author: Jiajia Liu, liujiajia@nwpu.edu.cn

**Abstract** Software verification has always been a popular research topic to ensure the correctness and security of software. However, due to the complex semantics and syntax of programming languages, the formal methods for verifying the correctness of programs have the problems of low accuracy and low efficiency. In particular, the state change in address space caused by pointer operations makes it difficult to guarantee the verification accuracy of existing model checking methods. By combining model checking and sparse value-flow analysis, this paper designs a spatial flow model to effectively describe the state behavior of C code at the symbolic-variable level and address-space level and proposes a model checking algorithm of CounterExample-Guided Abstraction refinement and Sparse value-flow strong update (CEGAS), which enables points-to-sensitive formal verification for C code. This paper establishes a C-code benchmark containing a variety of pointer operations and conducts comparative experiments on the basis of this benchmark. These experiments indicate that in the task of analyzing multi-class C code features, the model checking algorithm CEGAS proposed in this paper can achieve outstanding results compared with the existing model checking tools. The verification accuracy of CEGAS is 92.9%, and the average verification time of each line of code is 2.58 ms, both of which are better than those of existing verification tools.

**Keywords** software verification; model checking; spare value flow analysis; pointer analysis; vulnerability detection

**Citation** Yu YB, Liu JJ, Mu DJ. Counterexample-guided spatial flow model checking methods for C code, *International Journal of Software and Informatics*, 2022, 12(3): 263–284. http://www.ijsi.org/1673-7288/286.htm

Ensuring the correctness of software is the most critical and arduous task in today's software systems, which is one of the core factors to ensure the endogenous safety of the software systems. Manual verification of software, however, is error-prone and costly. For this reason, many methods including software analysis and testing have been proposed to improve the correctness and safety of software. In particular, formal verification of software programs has always been a focus in the field of software safety, which can prove the semantic and logical correctness of software programs mathematically and thus provide the core guarantee for software safety. At

This is the English version of Chinese article "反例引导的 C 代码空间流模型检测方法. 软件学报, 2022, 33(6): 1961–1977. doi: 10.13328/j.cnki.jos.006563"
Funding items: Guangdong Basic and Applied Basic Research Foundation of China (2021A1515110279); Basic Research Programs of Taicang, 2020 (TC2020JC03); Fundamental Research Funds for the Central Universities of China (D5000210588)
Received 2021-08-29; Revised 2021-10-16; Accepted 2022-01-10; IJSI published online 2022-09-23

present, software is mainly verified through two methods, i.e., abstract static analysis and model checking[1], where the former optimizes program analysis space by designing abstract domains, while the latter verifies safety or liveness properties of systems by structural models. Most of the existing verification tools combine these two methods to ensure scalability and accuracy of verification, such as CounterExample-Guided Abstraction Refinement (CEGAR)[2], symbolic execution[3], and proof by induction[4]. Software verification has been widely used in software engineering fields, such as for the verification of application software[5], communications and safety protocols[6, 7], device drivers and firmware, and system kernels, which has become the main method to ensure software safety.

Generally, the first step of formal verification is to model the program formally and determine the properties to be checked, and then, the state space of the model is traversed through a specific strategy to theoretically expose the paths in the program violating the checked properties[4]. However, due to the complex syntactic and semantic expression capacity of program languages, the accurate modeling of the behavior logic of software to ensure equivalence between models and program semantics has always been a key challenge in software formalization[1, 8]. To verify the logical correctness of program codes, the most direct method is to manually translate the code into a formal model and then conduct formal verification. Such a method requires high labor costs; moreover, it fails to ensure the accuracy of the manually translated model. Another solution is to design verification algorithms for high-level programming languages (such as Python and Java). With a high degree of abstraction, such languages can be easily and automatically translated into models for formal verification, and thus errors caused by manual translation can be avoided. By means of specific methods, code written in other languages can be translated into these languages for formal verification, but some underlying programming languages (such as C/C++) can hardly be described in high-level ones, due to their more complex semantic features. It is noted that most of the existing program verification tools (such as CPAChecker[9], SPIN[10], and Gazer[11]) for languages such as the C language are designed in view of state changes at the symbolic variable level of programs, which can verify software correctness efficiently by advanced techniques based on abstraction[12], interpolation[13], and counterexample-guided model refinement[2]. However, when a program contains memory operations such as pointer ones or complex data structures, these tools will not be able to or mistakenly perceive state changes at the symbolic variable level caused by state changes in memory space addresses. As a result, they will produce false verification results, failing to guarantee the validity of verification.

Taking C programs as the research object of formal verification, this study aims to tackle the failure by inaccurate verification as a result of state-space changes at the address level caused by pointer operations. The use of pointers can make code more concise, but in the dynamic execution of programs, pointers will also complicate the behavior of the programs. Improper use of pointers has become one of the main reasons for code defects. Therefore, static analysis of programs requires pointer analysis to identify objects of pointer variables at different locations. Pointer analysis, however, usually requires high-overhead data flow analysis and thus is challenged by a difficult trade-off between analysis accuracy and efficiency with the expansion of program scales. This is the main reason why most of the existing model checking algorithms do not support pointer operations. To improve the accuracy of model checking, existing methods usually employ independent pointer-analysis algorithms to obtain points-to information of program pointers in advance to assist model checking. However, high-accuracy pointer analysis (sensitive to flow and context, for example) has high computational overhead, possibly with a lot of unnecessary computational overhead, and the scalability of the analysis cannot be ensured. For insensitive pointer analysis (insensitive to flow and context, such as Anderson[14]), its scalability can be ensured, while its accuracy cannot be guaranteed.

To solve the above problems, with the idea of abstraction before verification and the combination of abstraction-based model checking and sparse value-flow analysis, this paper designs a counterexample-guided spatial flow model checking algorithm. Specifically, this algorithm constructs a formal model for a C program in linear time by adopting the method of insensitive points-to relations, which combines control flow information and sparse value-flow information of the program, that we call the spatial flow model in this paper. Then, the spatial flow model is abstracted by value abstraction, and the abstract model is checked for counterexamples. If a counterexample exists, and it is not a valid counterexample in the original model, it will be used to refine the misinformation caused by variable abstraction or insensitive points-to relations. On this basis, the information of the refined model can be gradually updated to realize analysis at both the program-symbol and address-space levels. Using counterexamples to guide model verification can effectively balance checking accuracy and efficiency.

The main contributions of this paper are as follows:

(1) A C-coded model called the sparse spatial flow model and its construction method are designed on the basis of a Sparse Value-Flow Graph (SVFG). The model can describe state changes of C programs at both symbol and address-space levels automatically and efficiently.

(2) Combining explicit-state analysis in formal methods and sparse-value analysis in software analysis, this paper proposes a counterexample-guided sparse spatial flow model checking algorithm. The algorithm uses spurious counterexamples to guide state-space analysis at both symbol and address-space levels to ensure accuracy and efficiency of verification.

(3) This paper establishes a C-program benchmark library with multiple complex syntactic and semantic features. With this library, it is proved that the designed model checking method has higher verification accuracy and efficiency than the existing methods.

In this paper, Section 1 introduces the related work. Section 2 briefly describes the motivation of this study and the designed solution. Section 3 elaborates on the counterexample-guided spatial flow model checking algorithm proposed in this paper. Section 4 compares the test results of the proposed method and the existing methods in terms of verification accuracy and efficiency. Finally, the summary is made, and research directions worthy of attention in the future are preliminarily discussed.

## 1 Related Work

Software verification is usually composed of state-space traversal and path verification. At present, the mainstream algorithms of software model checking generally transform formal verification of paths into the satisfaction of formulas[15] and then use SMT solvers to verify the feasibility of the paths. According to strategies for state-space traversal, the existing methods can be divided into three categories, i.e., explicit-state model checking, proof of induction, and predicate abstraction[15]. Specifically, the explicit-state model checking explicitly records corresponding states through state reachability graphs and traverses all possible initial paths in state space as far as possible through depth, breadth, or heuristic algorithms, as done in tools such as SPIN[10], CMC[16], and Java Path finder[17]. With the expansion of program scales, however, the state-space explosion has become the most critical challenge for model checking. For this reason, researchers set bounds on the number of expansions of all paths in a program (namely, bounded model checking) and encode paths within the bounds as SMT formulas for verification to reduce state-space explosions, as done by SMACK[18] and CBMC[19]. Bounded model checking can quickly detect logic defects of programs but cannot guarantee verification integrity. Proof of induction extends the strategy of bounded model checking by verifying some states before recursively verifying others, which is generally called k-induction, as implemented in ESBMC[20]. Predicate abstraction is used to form an over-approximated abstract model

by setting abstract domains to alleviate state-space explosion, and the most commonly used strategy is CEGAR[2]. After an abstract model is checked, if a detected counterexample is invalid in the original model, the abstract domain is refined by interpolation based on the information that makes this counterexample invalid to remove the false counterexample. Able to balance checking efficiency and accuracy effectively, CEGAR becomes the most important model checking strategy in tools such as SLAM[21], CPAChecker[9], and Theta[22]. Considering the semantic and syntactic complexity of programs and the increase in the code size, the existing model checking tools usually combine multiple state-space traversal strategies or use heuristic algorithms and reinforcement learning methods to ensure both the efficiency of verification and the comprehensiveness of model checking.

The software verification methods mentioned above are all based on formalization methods, most of which can only analyze state changes in symbolic variables, failing to directly support complex program semantics such as pointers[8]. The existing tools, such as CPAChecker and SMACK, use algorithms for pointer analysis to calculate pointer information before model checking. Pointer analysis, one of the challenges in program static analysis, attempts to compute objects that pointers point to, and correct pointer information is a key factor in detecting software vulnerabilities (such as null pointer dereferences, memory leaks, and use-after-free). In pointer analysis, multi-dimensional information including paths, flow, context, and fields will affect the tradeoff between analysis accuracy and overhead. Usually, static analysis adopts insensitive pointer analysis, such as that based on implication constraints (also known as Anderson) or unification constraints (also known as Steensgaard[23]), the two of which can guarantee analysis scalability but fail to guarantee analysis accuracy. To obtain more accurate points-to relations, such as flow-sensitive ones, pointer analysis needs to solve data flow problems iteratively, which results in huge computational overhead. For this reason, a method of sparse value-flow analysis was proposed[24–26], which transfers data-flow computation previously required at all control flow nodes to that on pre-computed sparse and approximated def-use chains to effectively reduce computational overhead. Thus, it is applicable to large-scale program analysis. By sparse value analysis, a demand-driven pointer analysis was designed for C and C++ code in Ref. [25], where after an SVFG is computed by Anderson's algorithm, information about the pointer at its location can be analyzed. With the framework of sparse value-flow analysis, Ref. [27] successfully detected use-after-free and null pointer dereference defects in millions of lines of code by designing simplified local pointer analysis and using SMT for feasibility verification of global paths. On the basis of Ref. [27], Ref. [28] proposed path-sensitive analysis based on slicing, which is able to provide null pointer dereference and taint analysis for large-scale code analysis.

Such methods of calculating points-to information through pointer analysis before model checking can ensure verification accuracy. However, the premise is that flow-sensitive and context-sensitive pointer analysis is required, which means the computational overhead will be huge, and thus the efficiency of model checking is restricted. In addition, pointer analysis may produce a lot of computational redundancy for model checking, such as analysis of pointer variables unrelated to counterexamples. Therefore, this paper achieves a more efficient model checking method for C programs by integrating model checking with sparse value-flow analysis.

## 2 Overview of Methods

### 2.1 Research motivation

Most of the existing model checking algorithms are designed in view of state changes at the symbolic variable level of programs. In the case of programs containing memory operations such as pointer ones or complex data structures, these algorithms will fail to or mistakenly

perceive state changes at the symbolic variable level caused by information changes in memory space addresses. This will cause unnecessary analysis overhead and false analysis results.

To better illustrate this problem, this section takes the C program shown in Fig. 1(a) as an example to explain defects in the existing model checking algorithms. We assume that the model of the C code is $\mathcal{M}$, and the program property to be checked is $\varphi \equiv LTL(G!call(error()))$, which means that the code will not call the function *error*() at any time; in other words, the code executed at Line 22. Model checking requires verifying whether all initial states of $\mathcal{M}$ violate $\varphi$, which can be expressed by the formula $\mathcal{M} \models \varphi$. To solve this problem, as shown in Fig. 1(b), the existing model checking algorithms first convert the code into a control flow graph, namely, $\mathcal{M}$, and then the model is transformed into a formal formula composed of multiple propositions (such as $\theta_1$ and $\theta_2$). As most of the existing algorithms are at the symbolic level, they fail to check pointer operations in the C languages and can only analyze symbolic variables and corresponding assignments (i.e., $i = 3$ and $k = 8$), and thus, the propositions $\theta_1$, $\neg\theta_2$, $\neg\theta_4$, and $\theta_5$ are true. As a result, the verification procedure concludes that $\mathcal{M} \models\varphi$ does not hold, namely that there is an execution path violating the property $\varphi$ (i.e., a counterexample): $\theta_1 \wedge \neg\theta_2 \wedge \neg\theta_4 \wedge \theta_5$. The counterexample indicates that the sample code in Fig. 1(a) will be executed according to this path and can reach the location (Line 22) where the function *error*() is called. In fact, due to pointer operations in the code, the variables $i$ and $k$ will be assigned indirectly by pointer variables. Therefore, the verification process of model checking does not perform a comprehensive verification, which cannot ensure the accuracy of results.

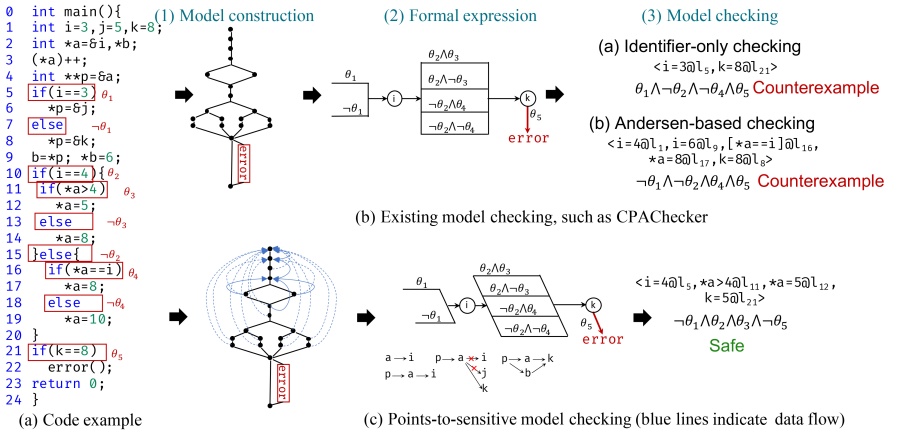


**Figure 1** Example of research motivation

For higher verification accuracy, pointer analysis is introduced into model checking algorithms to obtain points-to relations of pointers in advance or during verification, so as to correct changes due to variable assignment. Pointer analysis has many characteristics. For example, (1) if the analysis is sensitive to control flow, it is flow-sensitive; otherwise, it is flow-insensitive; (2) if the analysis requires differentiation between different function-calling contexts, it is context-sensitive; otherwise, it is context-insensitive; (3) if the analysis requires differentiation between different fields in data structures, it is field-sensitive; otherwise, it is field-insensitive. Accurate points-to relations can be computed by pointer analysis sensitive to such information. However, with the expansion of the code size, such analysis is challenged by exponentially increased computational complexity, hardly applicable to large-scale models. Therefore, the existing algorithms usually employ information-insensitive pointer analysis to reduce computational resources and improve computational speed for large-scale code

verification tasks. For example, Andersen's algorithm is a pointer analysis algorithm insensitive to both flow and context and has linear computational complexity, which is suitable for large-scale pointer analysis, but its insensitivity to flow and context results in over-approximation of computed points-to relations. In Fig. 1(b), the Anderson-based model checking algorithm computes points-to relations of pointers while verifying the model, and on the second line of the code, the pointer variable $a$ points to the variable $i$ (namely, $a \mapsto i$). Thus, on the third line, we have $i = 4$, and $\neg\theta_1 = \mathit{true}$. As points-to relations computed by Anderson's algorithm is flow-insensitive, the algorithm can compute the points-to relations on the 8th line as follows: $p \mapsto \{a\} \mapsto \{i, k\}$. By analogy, the model checking algorithm finally computes the counterexample $\neg\theta_1 \wedge \neg\theta_2 \wedge \theta_4 \wedge \theta_5$.

By actual analysis of the code example in Fig. 1(a), we can find that neither of the above two counterexamples holds. Due to the pointer $b$, the value of the pointer $a$ in the address space (namely, the address of the variable $k$) to which it points on Line 11 is 6, and thus the proposition $\theta_3$ is true. It can be learned from this that the variable $k$ is actually 5 on Line 21 of the code, and the proposition $\theta_5$ is false. Thus, the program cannot run to Line 22. As shown in Fig. 1(b), an accurate model checking algorithm should be able to simultaneously sense and detect state transition of symbolic variables and pointer-represented memory address space (i.e., a points-to-sensitive model-checking algorithm). On this basis, the feasible path $\neg\theta_1 \wedge \theta_2 \wedge \theta_4 \wedge \neg\theta_5$ can be obtained, which proves that the code will not call the function *error*().

## 2.2 Research methods

To solve the above problems, on the basis of CEGAR, this paper designes a points-to-sensitive model-checking algorithm for C programs, namely, CEGAS (CounterExample-Guided Abstraction refinement and Strong update). Fig. 2 illustrates the framework of CEGAS. In terms of modeling and verification, this section briefly explains how CEGAS realizes points-to-sensitive model checking on the premise of ensuring both checking efficiency and accuracy.

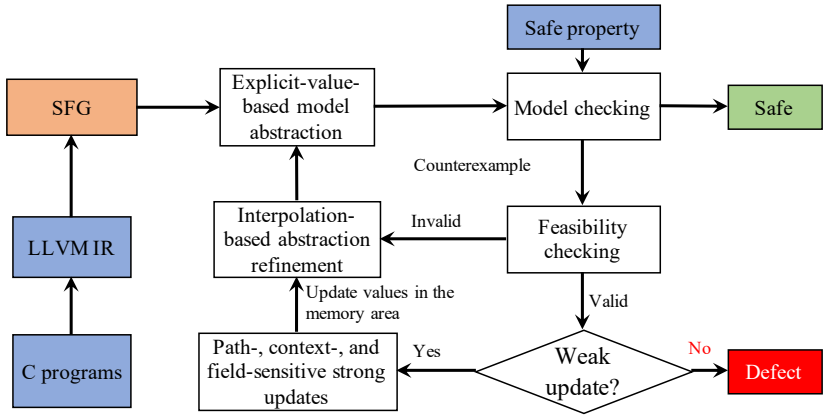


**Figure 2** Framework of CEGAS

(1) **Modeling**. The existing methods use control flow information of a C program for model checking, but control flow information lacks direct points-to information of pointers, which results in inaccurate analysis. In view of this, we propose to use a Spatial Flow Graph (SFG) to formally describe the logical behavior of a C program. Specifically, we use an SVFG to describe points-to relations of pointers while retaining control flow information. SFG can make the formal model of a C program more concise and efficient and provide sufficient information for systematic model checking. As shown in Fig. 2, the module for establishing SFGs is built

on the LLVM compiler and is executed by the well-defined LLVM IR language converted from the C program.

(2) **Verification.** For higher accuracy of model checking, the existing methods provide points-to information of pointers for model checking through independent dataflow analysis. However, it is computationally costly to obtain accurate points-to information by dataflow analysis[24, 27], while inaccurate points-to analysis (flow-/context-insensitive) fails to guarantee the accuracy of model checking. For this reason, we design an SFG-oriented model checking algorithm by combining model checking with sparse value analysis. Specifically, insensitive but fast (linear) points-to analysis is conducted to obtain over-approximated points-to information, and a coarse-grained abstract model is obtained through explicit state abstraction[4, 29]. Then, counterexample-guided model refinement is performed, namely that false counterexamples are used to guide the refinement of abstraction precision and the computation of definite points-to relations. By this SFG-based counterexample-guided method, both accuracy and efficiency of model checking can be guaranteed simultaneously.

## 3 Counterexample-guided Spatial-flow Model Checking

Formal verification of a program consists of two parts: formal description and formal verification of the program. From the two perspectives, this section introduces the counterexample-guided spatial-flow model checking algorithm proposed in this paper.

### 3.1 Definition and construction of spatial-flow model

To perceive state transition information of a program at the symbolic level and in memory space, we design an efficient formal model called SFG to describe code execution in C programs.

**Definition 1.** An SFG is composed of two sub-graphs, i.e., $\mathcal{M} = \{G_c, G_s\}$, which represent Control Flow Automation (CFA) and SVFG, respectively.

- $G_c = (L, l_0, E)$, where $L$ is the set of program locations, representing the program counter. $l_0 \in L$ is the initial program location, that is, the entry point of the program. $E \subseteq L \times S \times L$ is the set of control-flow edges, representing the operations executed during the transition from one program location to another. $S$ is the set of all operational statements in the program, and the set of all program variables is $V$.
- $G_s = (N, \mathcal{E})$, where $N \subseteq L \times S$ represents the set of definition (def) or use nodes of pointer variables; $\mathcal{E} \subseteq N \times V \times N$ represents the set of all possible def-use chains of pointer variables, which is also called value-flow edge set. For example, the edge $n_1 \xrightarrow{v} n_2$ indicates a def-use chain in which the pointer variable $v \in V$ is defined at the node $n_1$ and used at the node $n_2$, rather than all program points between $n_1$ and $n_2$.

SFG can accurately describe state transitions in a program through CFA. Considering complex memory-address changes in software (namely, points-to relations of pointer variables; in this paper, the formula $p \mapsto o$ indicates that the pointer variable $p$ points to the memory address of the object $o$), SFG further uses SVFG[30] to describe changes in states of memory addresses. SVFG only pays attention to possible def-use chains of pointer variables, without propagating points-to relations of pointers to all program points of control-flow paths, and the sparsity enables SVFG to accurately analyze memory-space information changes of large-scale programs[27, 28]. SFG, combined with CFA and SVFG, can describe the execution behavior of large-scale complex codes efficiently and accurately.

To construct the SFG of a C program, this paper first uses the open-source LLVM compiler[31] to compile the C program into the LLVM intermediate representation, namely, LLVM IR, which has a concise statement structure and instruction set. Standardizing the source code into this intermediate representation can simplify model construction. LLVM IR is a

language of partially Static Single Assignment (SSA), containing two types of variables: top-level and address-taken variables. The former is explicitly placed in the form of SSA by using the standard SSA construction algorithm, namely that such variables are assigned only once in their lifetime. Thus, top-level pointer variables have definite points-to relations (i.e., must point-to). The latter is not in the form of SSA, which needs to be accessed indirectly by top-level variables using the two instructions *load* (i.e., $p = *q$) and *store* (i.e., $*p = q$). As address-taken variables are used indirectly through *load* and are allowed to be defined indirectly by multiple *store* instructions, pointer variables of such a kind usually have indefinite points-to relations (namely, may point-to).

Pointer analysis algorithms for LLVM IR are used to identify points-to relations of address-taken pointer variables or in related expressions. Conventional pointer analysis needs to iteratively compute points-to relations satisfying Meet-Over-all Paths (MOP) by means of control flow and data flow information of a program, and in this way, the computation of correct points-to relations can be ensured. For large-scale programs, however, it is costly and non-scalable. By propagating data flow facts (namely, points-to relations of pointers) according to the data dependency of a program, sparse program analysis (also known as strong-update analysis[25]) avoids propagation at all program points in the control flow graph of the program, which ensures the scalability of the analysis. Sparse program analysis usually works in stages: first, it pre-analyzes a program and defines all def-use chains of pointer variables through weak updates, and weak updates conservatively assume that old contents at locations with may points-to relations are retained. As a result, def-use chains obtained by the pre-analysis are over-approximated. Then, sparse program analysis computes must points-to relations on the approximated def-use chains obtained by pre-analysis rather than on the whole control flow. In other words, strong updates are carried out, covering previous contents of pointer variables with new values. This is an important factor to ensure the accuracy of pointer analysis.

For SFG construction, we obtain points-to information by Andersen's pointer analysis algorithm on the basis of the LLVM compiler. Moreover, we depict may points-to relations through two weak-update functions ($\mu(a)$ and $a = \chi(a)$) and describe def-use chains containing all top-level and address-taken pointer variables in the form of inter-procedural memory SSA. In this way, SFG can be constructed considering control flow information. Fig. 3 shows an example of SFG construction. The function $\mu(a)$ indicates the use of the variable $a$, and for the instruction load $p$, $\mu(a)$ indicates that each variable $a$ (such as $i$ and $j$) pointed to by the pointer $p$ may be accessed indirectly in this instruction. The function $a = \chi(a)$ indicates def & use of the variable $a$, and for the instruction store $*p = \&i$, $a = \chi(a)$ represents that each variable $a$ pointed to by the pointer $p$ may be redefined and reused. As shown in Fig. 3(c), LLVM IR of the C program is converted into SFG for description, which provides a formal model for back-end model checking.

## 3.2 Spatial flow model checking

We now propose the counterexample-guided model-checking algorithm CEGAS for a spatial flow model. The algorithm uses detected false counterexamples to simultaneously guide the refinement of model abstraction precision and the strong update of the value-flow relationship. This can not only avoid unnecessary computational overhead (such as path-constraint solving and irrelevant-pointer analysis) but also ensure accurate analysis of state changes at the variable-symbol level and the memory-address level during model checking.

As shown in Fig. 2, CEGAS is an extension of CEGAR. CEGAS first abstracts the state transition process in the control flow information of the SFG, and then it finds out whether there is a counterexample through explicit-value analysis. If a counterexample exists, the algorithm will check in the SFG whether a weak update exists on the counterexample path that can further

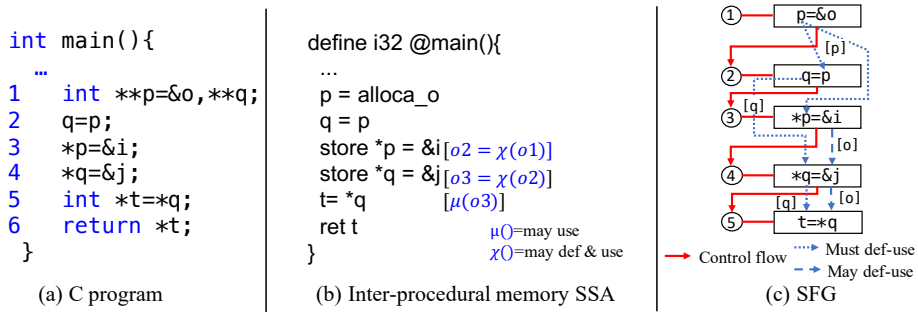


**Figure 3** An example of SFG construction from a C program

lead to a may points-to relation, and this points-to relation may cause a false counterexample. Therefore, if there is a weak update in the counterexample, strong-update computation will be executed along this path to feed back the must points-to relation to the abstract model, and then the model will be further verified. This section first explains the strong-update method of SFG under a given counterexample path and then introduces CEGAS as a whole. The details are as follows.

### 3.2.1 *Counterexample-guided path-sensitive strong update*

On a valid counterexample path detected by value analysis, a possible weak update may cause an incorrect flow computation, which indicates that the path may be spurious. For this reason, a counterexample-guided strong update method of spatial flow models is designed. The core idea of the method is as follows: according to the counterexample path detected by value analysis, the possibility of a weak update on this path is judged. If a weak update exists, the path-sensitive strong update is executed to refine changes in memory information of the model for further analysis of the validity of this path.

**Path-sensitive**. Path-sensitive pointer analysis is more accurate than flow-sensitive pointer analysis. For example, for the code example in Fig. 1(a), with flow-sensitive pointer analysis, the points-to relation $a \mapsto i$ is strongly updated to $a \mapsto j$ on Line 6 ($l_6$) of the code. Due to the lack of control flow information in sparse analysis, the algorithm will strongly update the points-to relation to $a \mapsto k$ on Line 8 ($l_8$). However, in the case of $\theta_1 = \text{true}$, the execution path cannot reach $l_8$, and thus, the points-to relation $a \mapsto k$ is wrong. In contrast, path-sensitive analysis first records the path information $\rho$, which is usually composed of a series of predicates, such as $\rho_1 = \theta_1 \wedge \theta_2 \wedge \cdots$ and $\rho_2 = \neg\theta_1 \wedge \theta_2 \wedge \cdots$. As $l_6 \in \rho_1$, and $l_8 \in \rho_2$, the points-to relation of the pointer variable $a$ can be expressed as $a \mapsto \phi(j[\rho_1], k[\rho_2])$ at Line 9, where $\phi$ refers to a conditional selection function. In other words, when the current path is $\rho_1$, we have $a \mapsto j$; otherwise, we have $a \mapsto k$.

**Field-sensitive**. Field-insensitive pointer analysis treats different fields in a structure variable as a singleton object, but different fields may point to different objects. Therefore, field-insensitive pointer analysis will not be able to perform effective strong updates on structure and array variables, thus failing to ensure the correctness of the points-to analysis results. For this reason, a simple but effective way is employed to achieve field sensitivity: the multi-level index of the instruction GetElementPtr in LLVM IR is used as the index of the fields to distinguish different fields in the same structure variable. GetElementPtr is an instruction to obtain structure or array elements in LLVM IR. For example, `%tmp = getelementptr inbounds %struct.struct_a* %S, i32 2, i32 1` means to fetch the second subfield ($S_2 \cdot e_1$) of the third element of the variable $S$ of the structure array *struct_a*. To realize this, first, we construct an object $S$ for the whole structure or array variables, and then we use index

information in GetElementPtr to build a field-sensitive object to update points-to relations.

**Constraint rules**. The strong update is a kind of backward reachability analysis on SVFG. In this paper, $\rho$ represents the path of the current analysis, and in terms of $\rho$, a series of constraint rules for backward reachability analysis in this path are designed. This paper does not consider context-sensitive analysis but maps a called function to the main function by the inline method. Thus, an operational statement upon analysis contains information on $\rho$ and $n$; for example, $\rho$, $n : p = \&o$ means that the current node $n \in N$ is in the path $\rho$. Considering weak updates, a pointer may have multiple points-to addresses and one points-to set, and thus this paper uses $p \supseteq q$ to represent the addition of the points-to set of $q$ into that of $p$.

Constraint rules of strong updates are designed in this paper, as shown in Table 1.

**Table 1** Constraint rules in counterexample-guided path- and field-sensitive strong updates

| Rule | Operational statement | Constraint | Role |
|---|---|---|---|
| ADDR | $\rho, n : p = \&o \quad n' \xrightarrow{o} n$ | $p[\rho, n] \mapsto o[\rho, n']$ | Under the condition of $\rho$, $p$ points to the memory address of $o$ at $n$ |
| COPY | $\rho, n : p = q \quad n' \xrightarrow{q} n$ | $p[\rho, n] \supseteq q[\rho, n']$ | Under the condition of $\rho$, the points-to set of $p$ contains that of $q$ |
| PHI | $\rho, n : p = \phi(q, r)$<br>$n' \xrightarrow{q} n \quad n'' \xrightarrow{r} n$ | $p[\rho, n] \supseteq_{n' \in \rho = T} q[\rho, n']$<br>$p[\rho, n] \supseteq_{n'' \in \rho = T} r[\rho, n'']$ | If $n'$ is in $\rho$, the points-to set of $p$ contains that of $q$; otherwise, it contains that of $r$. |
| GEP | $\rho, n : p = \&(q_i \to e_j)$<br>$q_i \mapsto o \quad n' \xrightarrow{o \cdot e_j} n$ | $p[\rho, n] \mapsto o \cdot e_j[\rho, n']$ | Under the condition of $\rho$, $p$ points to the memory address of $o \cdot e_j$, where $o \cdot e_j$ is the $j$th field in the $i$th element of the array $q$ |
| STORE | $\rho, n : *p = q \quad n' \xrightarrow{p} n$<br>$n''' \xrightarrow{q} n \quad n'' \xrightarrow{r} n$<br>$p[\rho, n'] \mapsto r[\rho, n'']$ | $r[\rho, n''] \supseteq q[\rho, n''']$ | Under the condition of $\rho$, the points-to set $r$ of the object pointed to by $p$ is obtained to make the points-to set of $r$ contain that of $q$ |
| LOAD | $\rho, n : p = *q$<br>$n' \xrightarrow{q} n \quad n'' \xrightarrow{r} n$<br>$q[\rho, n'] \mapsto r[\rho, n'']$ | $p[\rho, n] \supseteq r[\rho, n'']$ | Under the condition of $\rho$, the points-to set of the object $r$ pointed to by $p$ is obtained to make the points-to set of $q$ contain that of $r$ |
| SU/WU | $\rho, n : *p = \_ \quad n' \xrightarrow{o} n$<br>$o \in \mathcal{O} \backslash kill(p[\rho, n])$ | $o[\rho, n] \supseteq o[\rho, n']$ | $kill(p[\rho, n]) =$<br>$\begin{cases} o'[\rho] & \text{if } p[\rho, n] \mapsto o'[\rho] \wedge o'[\rho] \in pcSingletons \\ \mathcal{O} & \text{if } p[\rho, n] \mapsto \varnothing \\ \varnothing & \text{otherwise} \end{cases}$ |
| COMPO | $n \xrightarrow{o} n' \quad n' \xrightarrow{o} n''$ | $n \xrightarrow{o} n''$ | Transtivity of def-use chains |

- ADDR is used to define the points-to relation of $p$ by using def-use chains of $o$ to inversely obtain the memory address of $o[\rho, n']$ declared at the node $n'$. The symbol [] denotes valid sensitive information.
- COPY and PHI are two instructions for top-level variables in LLVM IR, which can yield points-to sets through must def-use chains[7], and as PHI is path-condition dependent, it is necessary to judge whether $n'$ and $n''$ belong to $\rho$. In the case of $n'$ belonging to $\rho$ (denoted by $n' \in \rho = \text{True}$), we have $p[\rho, n] \supseteq q[\rho, n']$, and thus path sensitivity is achieved.
- GEP is used to implement field-sensitive analysis in field access. According to the index of GetElementPtr, GEP constructs an object for the field, which serves as the object to

which the field pointer points.

- STORE may introduce multiple indirect def-use chains to address-taken variables. It is necessary to compute and remove current false def-use chains in an update process to ensure the correctness of points-to relations.
- LOAD is used to assign values to top-level variables indirectly from points-to relation sets of address-taken variables. Due to the existence of multiple STORE operations, points-to relations obtained by top-level variables will contain false def-use chains.
- SU/WU represents strong and weak updates in STORE $\rho, n :^* p = \_$. Strong updates $kill(p[\rho, n])$ will be executed in three cases: (1) when the pointer $p$ points to the object $o \in \mathcal{O}$ of a path-correlated singleton (*pcSingletons*), $n' \xrightarrow{o} n$ should be cut off, and the content at the original $n'$ in $o$ should be removed; the content at $n$ should be updated. (2) When the points-to set of $p$ is empty, $n' \xrightarrow{o} n$ should be cut off to avoid null pointers. (3) Weak updates will be executed in other cases. By SU/WU, we can obtain points-to information that is used to process address-taken variables in LOAD and STORE.
- COMPO represents the transitivity of def-use chains.

Given a counterexample path, CEGAS first traverses the value flow information in SFG backward along this path to determine whether a weak update exists in the path. The existence of a weak update indicates that the current points-to information contains may relations. On the basis of the above constraint rules, CEGAS analyzes the SFG and performs strong updates to remove false def-use chains caused by the weak update in the path, which provides sensitive points-to information of this counterexample for the following model checking.

### 3.2.2 *Counterexample-guided spatial flow model checking*

Model checking can be formally described as $\mathcal{M} \models \varphi$, where $\mathcal{M}$ is a Kripke-structured model, and $\varphi$ is a property to be checked. This paper mainly focuses on the safety property of a program, namely that the program will never reach an unexpected state *state*. This can be expressed as $\varphi \equiv LTL(G!state)$ through linear temporal logic. Given a Kripke-structured model, we can use many advanced model checking algorithms to check the correctness of the model[4], such as predicate analysis, explicit-value analysis, symbolic analysis, IC3, and abstraction. In this paper, explicit-value analysis based on abstraction refinement (also known as explicit-state analysis)[29] is used as the main model checking algorithm; combined with model abstraction, this analysis method can remove program variables unrelated to verification properties and only track variables necessary to refute invalid counterexample paths. In this way, the state-space explosion caused by excessive program variables and their values can be reduced to make verification more concise and efficient. Using the abstraction-refinement-based model checking strategy, CEGAS starts with null abstraction precision to iteratively refine the precision of symbolic variables and points-to information from invalid paths. The details are as follows.

Given the spatial flow model of a program, CEGAS first needs to abstract the model; the following two states are involved.

- One is the representational state $s$. It refers to a variable assignment $s := cs@l$, where $cs$: $V \to \mathbb{Z}$ indicates that an integer value is assigned to a program variable, and $l \in L$ is a program location. Similar to the existing model checking algorithms, the model checking method in this paper mainly focuses on integer variables.
- The other is the abstract state $\hat{s}$. It is expressed as: $\hat{s} = as@l$, where $as$: $V \to \mathbb{Z} \cup \{\top, \bot\}$ is an abstract variable assignment. Specifically, $\top$ is an unknown value; for example, it is assigned by an uninitialized variable or called by an external function; $\bot$ represents no value, namely, a contradicting variable assignment.

The abstract state is abstracted from the representational state according to precision. In this paper, a lazy explicit-value abstraction[12] method is adopted, which uses different precision for different abstract states in different program paths. For a specific program, precision is defined by a function $\Pi$: $L \to 2^v$, which provides a precision $\pi$ for each location of the program for the abstraction of variable assignment statements. As to a variable assignment statement, the precision $\pi$ defines a group of program variables to be analyzed and tracked for the abstraction of the variable assignment statement. At a given precision $\pi$, for the explicit-value abstraction of a variable assignment statement, its variable is determined by $\pi$. For example, $\pi = \varnothing$ indicates that no program variables will be tracked, and corresponding abstract states are null; $\pi = V$ means that all program variables are tracked, and corresponding abstract states are equal to representational ones; $\pi = \{i\}$ means that the explicit-value abstraction state of the variable assignment set $v = \{i \mapsto 1, j \mapsto 2\}$ is $v^\pi = \{i \mapsto 1\}$. Generally speaking, at a program location $l \in L$, the explicit-value abstraction of a variable assignment extracts precision through the precision function $\Pi(l)$, and then, the abstract state is computed according to precision-tracked variables. At the beginning of the verification, the precision of CEGAS is $\Pi_{\text{init}}(l) = \varnothing$. In other words, for each $l \in L$, no variable is tracked.

We define a path as a sequence $\rho := \langle (op_1@l_1), \cdots, (op_n@l_n)\rangle$ consisting of a series of operational-statement and program-location pairs, where $\gamma_\rho = \langle op_1, \cdots, op_n\rangle$ is the constraint sequence of this path. Program locations in this paper take BasicBlock of LLVM IR as the basic block, and *op* usually contains multiple operational statements. When CEGAS detects a path $\rho$ that can reach an error location $l_\varphi$ that violates the property $\varphi$, namely, a counterexample CEX, the algorithm first analyzes the feasibility of the counterexample path $\rho$ by SM Tunder full precision ($\Pi(l) = V$) (namely that the function *isFeasible*($\rho$) is used to judge whether the constraint sequence $\gamma_\rho$ can be satisfied). If it is not satisfied, the un-satisfaction may be due to low precision or the erroneous points-to information caused by weak updates, and thus, the precision should be refined. In this paper, Craig interpolation is used to generate new interpolants. It is defined as follows.

For the given formulas $\varphi^-$ and $\varphi^+$, with $\varphi^- \wedge \varphi^+$ being unsatisfiable, their Craig interpolant $\iota$ refers to a formula that satisfies the following constraints: (1) $\varphi^- \Rightarrow \iota$ is valid, namely that $\varphi^- \wedge \neg\iota$ is unsatisfiable; (2) $\iota \wedge \varphi^+ \Rightarrow$ *false*, namely that $\iota \wedge \varphi^+$ is unsatisfiable; (3) $\iota$ only contains common symbols of formulas $\varphi^-$ and $\varphi^+$, as well as the symbols of the theory itself. The definition can be extended to an ordered formula sequence $\rho = \varphi_0, \cdots, \varphi_n$, with $\bigwedge_{0\le i\le n} \varphi_i \Rightarrow$ *false*. The above interpolation method can produce a series of interpolants $\iota_0, \cdots, \iota_n$: (1) $\bigwedge_{0\le k\le i} \varphi_k \Rightarrow \iota_i$ is valid; (2) $\iota_i \wedge \bigwedge_{i\le k\le n} \varphi_k \Rightarrow$ *false*; (3) $\forall 1 \le i \le n, \varphi_{i-1} \wedge \iota_i \Rightarrow \iota_{i+1}$; (4) $\iota_i$ only contains common symbols of formulas $\bigwedge_{0\le k\le i} \varphi_k$ and $\bigwedge_{i\le k\le n} \varphi_k$, as well as symbols of the theory itself.

The above-mentioned sequential interpolants (expressed by the function *SeqInterpolant*) can be effectively calculated by SMT techniques (such as Z3[32]). The above method can only use information in the value space to generate interpolants, lacking information in address space, which results in incomplete precision refinement. For this reason, this paper takes the sequential interpolants as the basis and proposes a points-to-sensitive precision refinement algorithm (*Refine*); its pseudo-code is shown as Algorithm 1. Given an infeasible error path $\rho$, this algorithm first computes the first node that makes the constraint sequence unsatisfiable on the path $\rho$ (Lines 2–4). For the part in which path constraints are satisfiable, we use the path-sensitive strong update rules (represented by the function *PStrongUpdate*) from Table 1 to analyze the existence of a weak update in the path, and if a weak update exists, we will strongly update this part to avoid erroneous points-to information (Lines 5–6). Then, the algorithm uses the function *SeqInterpolant* to obtain the sequence $I$ of interpolants (Lines 7–10), and upon

**Algorithm 1.** Points-to information sensitive abstraction-precision refinement algorithm (*Refine*($\rho, \mathcal{M}$))

**Input:** the infeasible error path $\rho : \langle (op_1@l_1), \cdots, (op_n@l_n)\rangle$ and spatial flow model $\mathcal{M}$
**Output:** precision $\Pi$
**Variable:** interpolant sequence $I$

1. Assume the precision of each program locations $l$ is $\Pi(l) := \varnothing$; $I :=<>$
2. // Findout the first node that makes constraints on the path $\rho$ unsatisfied;
3. **for** $i := n-1$ to 0 **do**
4. **if** $\bigwedge_{0\le k\le i} op_k \Rightarrow true$ **then**
5. $\rho_I :=\rangle(op_0@l_0), \cdots, (op_i@l_i)\langle$
6. **end if**
7. **if** *hasWeakUpdatePath*($\rho_I, \mathcal{M}$) := *true* **then**
8. $\mathcal{M}$ := *PStrongUpdate*($\rho_I$); //The path is satisfied, but a weak update exists. The path-sensitive strong update is performed on the model $\mathcal{M}$.
9. **end if**
10. **for** $j := i-1$ to 0 **do**
11. $\iota_j$ := *SeqInterpolant*($\bigwedge_{0\le k\le j} op_k, \bigwedge_{j\le k\le i} op_k$);
12. **if** $\iota_j$ is empty **then**
13. **break**;
14. **end if**
15. **end for**
16. $I := \{I \cup \iota_j\}$;
17. **for** $k := j$ to $i-1$ **do**
18. $\Pi(l_{0\le t\le k}) := \{v | v \in \iota_k, \iota_k \in I\}$; //The variables used by interpolants are added to the precision of all program locations including $l_k$ backward on the path $\rho$
19. $\Pi(l_{0\le r\le k}) := \{v^p | v^p \in$ *BackReachability*$(\mathcal{M}, l_{0\le r\le k}, \iota_k)\}$; //As above, the pointer variables pointing to interpolation variables are added
20. **end for**
21. **end for**

that, the program variable used by the interpolant $\iota_k$ at the corresponding program location $l_k$ is refined to the precision of all locations backward along the path $\rho$ with $l_k$ as the starting point. Moreover, the function *BackReachability* finds out def-use chains of the variable used by the interpolant $\iota_k$ backward from the program location $l_k$ in the spatial flow model $\mathcal{M}$, and then it adds the pointer variable that points to the variable used by $\iota_k$ into the precision of the corresponding location. This algorithm is applicable to model abstraction for iterative precision refinement from the two aspects of value space and address space.

On the basis of the above refinement method, CEGAS records analyzed states and their reachability in a tree-like form by means of abstract reachability graphs. Specifically, two intermediate variables are used to record information in the analysis: the set *reached* $\subseteq E \times \Pi$ is used to record all reachable abstract states under the current precision, and the set *waitlist* $\subseteq E \times \Pi$ is used to record all unanalyzed abstract states under the current precision. $E$ refers to the set of abstract states. The pseudo-code of the CEGAS algorithm is shown as Algorithm 2, and the specific steps are as follows.

- **Step 1**: Set the current precision $\pi$ to the null precision $\pi_0$ (namely that no variables are recorded) and initialize *reached* and *waitlist* (Lines 1–2).
- **Step 2**: Execute the model checking algorithm based on explicit-value analysis and extract an unanalyzed state $b$ from *waitlist*. Under precision $\pi$, a reachable successor of $b$ is found (namely that branch conditions can be satisfied after abstraction under $\pi$), and the successor is abstracted: if the new state is not analyzed, it is merged into *reached* and *waitlist*; if the new state violates the property $\varphi$, this means that a counterexample path is found, and in this case, the current loop (Lines 4–17) is ended.

**Algorithm 2.** CEGAS

**Input:** sparse spatial flow model $\mathcal{M}$ and the property $\varphi$
**Output:** whether results are **Safe** or **Unsafe** (counterexample path)
**Variables:** precision $\pi_0$, as well as sets *reached* $\subseteq E \times \Pi$ and *waitlist* $\subseteq E \times \Pi$

1. $\pi := \pi_0 := \varnothing$;
2. *reached* $:= \{(b_0, \pi_0)\}$; *waitlist* $:= \{(b_0, \pi_0)\}$;
3. **while** true **do**
4. **while** *waitlist* $\neq \varnothing$ **do**
5. Choose and remove $(b, \pi)$ from *waitlist*
6. **for** each $b'$ with $b \to E(b', \pi)$ **do**
7. //Traverse the successor branch of $b$; $b \to E(b', \pi)$ denotes that the conditions $b$ and $b'$ are satiable under the precision $\pi$.
8. $\hat{b} :=$ *abstraction*$(b', \pi)$; //Compute the abstraction state of $b'$ by precision $\pi$
9. **if** *isCovered*$(reached, \hat{b}) \neq$ *true* **then**
10. *waitlist* $:=$ (*waitlist* $\cup\{(\hat{b}, \pi)\}$); //Add a new abstract state
11. *reached* $:=$ (*reached* $\cup\{(\hat{b}, \pi)\}$); //Add a new abstract state in the abstract reachability graph and record the path of the previous state
12. **end if**
13. **if** *isTargetState*$(b', \varphi) :=$ *true* **then** //Judge whether there is a target state in the current Basicblock
14. **break while** //Exit the current while loop
15. **end if**
16. **end for**
17. **end while**
18. **if** *waitlist* $\neq \varnothing$ **then**
19. $\rho_a :=$*extractErrorPath*(*reached*); //Extract abstract paths from the abstract reachability graph, namely, abstract counterexamples
20. **if** *isFeasible*$(\rho_a) :=$ *false* **then** //Use Z3 to prove satisfiability of the path $\rho_a$
21. $\pi := \pi \cup$ *Refine*$(\rho_a, \mathcal{M})$; //If not satisfied, it is possible the path is a false counterexample, and interpolation is performed on the current precision
22. **else if** *hasWeakUpdatePath*$(\rho_a, \mathcal{M}) :=$ *true* **then**
23. $\mathcal{M} :=$ *PStrongUpdate*$(\rho_a)$; //The path is satisfied, but a weak update exists. The path-sensitive strong update is performed on the model $\mathcal{M}$
24. $\pi := \pi \cup$ *Update*$(\rho_a, \mathcal{M})$; //Use strong-update information to remove pointer variables caused by weak-update points-to information in precision
25. **else**
26. **return** *unsafe*; //The current counterexample path is feasible, violating the checked property $\varphi$
27. *reached* $:= \{(b_0, \pi)\}$; *waitlist* $:= \{(b_0, \pi)\}$; //Update *reached* and *waitlist* to redo abstraction analysis
28. **end if**
29. **end if**
30. **else**
31. **return** *safe*;
32. **end if**
33. **end while**

- **Step 3**: Judge whether there is any remaining unanalyzed state in *waitlist*. If there is no unanalyzed state, the program is correct for the property, and therefore, the verification procedure stops (Line 31).
- **Step 4**: If *waitlist* is not empty, an abstract counterexample path $\rho_a$ is constructed from the abstract reachability graph. On this basis, the function *isFeasible* is employed to judge whether the path is valid in the original model (Lines 18–20).
- **Step 5**: If $\rho_a$ is invalid, the current path is infeasible. The precision $\pi$ is updated by the points-to-information-sensitive constrained-differential-based precision-refinement

algorithm (*Refine*) (Line 21).

- **Step 6**: If $\rho_a$ is valid, the model $\mathcal{M}$ is traversed backward along this path to judge the existence of a weak update on $\rho_a$. If there is a weak update, the path-sensitive strong update of $\mathcal{M}$ is executed using the constraint rules in Table 1 to remove the points-to information caused by weak updates. Then, the tracking variables caused by points-to relations in the precision are updated to remove the abstract states of variable assignments caused by erroneous points-to information (Lines 22–24).
- **Step 7**: If $\rho_a$ is valid, with no weak updates, the path is valid in both the value space and the address space. This indicates that there is a counterexample violating the property $\varphi$ in the program. Thus, the model checking procedure is interrupted, and the counterexample is reported (Line 26).
- **Step 8**: Update $\pi$ in *reached* and *waitlist* to the refined one and jump to Step 2 for model checking again (Line 27).

CEGAS uses null precision in the first model checking iteration mainly due to two considerations: firstly, the simplest abstract model can quickly detect states with violations of the property in the program to avoid invalid computation and analysis. For example, there is no pointer operation on a counterexample path. Secondly, as the initial SFG uses insensitive pointer analysis, there is a lot of false points-to information, which will introduce misjudgment in the model checking procedure. After that, if weak updates exist on detected paths, strong updates are carried out to provide model checking with correct path-sensitive points-to information; if no weak updates exist, points-to information on corresponding paths is correct, which means the detected paths are considered as valid.

The CEGAS model verification is further illustrated by verifying the code shown in Fig. 1(a), and the result is shown in Fig. 4. CEGAS first traverses the control flow graph of the code with null abstract precision; according to the property $\varphi$, it detects the counterexample path CEX1: $\theta_1 \wedge \theta_2 \wedge \theta_3 \wedge \theta_5$, but CEX1 is invalid in the original model. In view of this, CEGAS employs the algorithm *Refine* and finds that $\theta_1$ is the constraint that causes the invalidity of CEX1, and then, it detects the interpolant variable $i$, as well as pointer variables $a$ and $p$, to increase the current abstraction precision. In the second round of model checking, CEGAS also detects the invalid counterexample path CEX2: $\neg\theta_1 \wedge \theta_2 \wedge \theta_3 \wedge \theta_5$. By analysis, it is found that the reason for the invalidity of CEX2 is the constraint $\theta_5$, namely, the lack of precision of the variable $k$; thus CEGAS computes new interpolant variables, including the variable $k$ and pointer variables $b$, $p$, and $a$. It should be noted that pointers $p$ and $a$ are also calculated here to increase the related precision of $p$ and $a$ backward from the constraint $\theta_5$ on the path CEX2, while the $p$ and $a$ calculated in the previous step are used to record related states before the constraint $\theta_1$. In the original insensitive spatial flow model, the object $a$ pointed to by $p$ is considered to be pointed to by three variables, i.e., $i$, $j$, and $k$, simultaneously before Line 9, but a weak update is

```
(base) ➜  CEGAS git:(master) ✗ ./cmake-build-debug/bin/cegas test/example.c --property=test/error.prp
Using property file test/error.prp
Found property 1 [CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )]
Start to verify test/example.c
==========Verification Loop 0================
An CEX path:entry->if.then->if.end->if.then2->if.then4->if.end6->if.end12->if.then14->PROPERTY_ERROR->
This path is InFeasible!
Generate interpolant variables: i@main a@main p@main
==========Verification Loop 1================
An CEX path:entry->if.else->if.end->if.then2->if.then4->if.end6->if.end12->if.then14->PROPERTY_ERROR->
This path is InFeasible!
Generate interpolant variables: k@main b@main p@main a@main
==========Verification Loop 2================
The program is Safe
```

CEX1:$\theta_1 \wedge \theta_2 \wedge \theta_3 \wedge \theta_5$

CEX2:$\neg\theta_1 \wedge \theta_2 \wedge \theta_3 \wedge \theta_5$

**Figure 4** An example of CEGAS processing the C program presented in Fig. 1

detected by the algorithm *Refine*. As a result, a strong update is carried out according to the path information; in other words, after Line 9, the original points-to relations $a \mapsto i$ obtained on Line 2 and $a \mapsto j$ on Line 6 are removed, while the points-to relation $a \mapsto k$ is retained. During the third verification, CEGAS detects no path able to reach the location to call the function *error*(). Thus, it is proved that the code shown in Fig. 1(a) is safe for the property $\varphi$.

# 4 Experimental Analysis

## 4.1 Experimental design

On the basis of the LLVM (6.0.0) framework and the Z3 SMT solver, we implement the tool CEGAS to verify assertions in an input program. Specifically, we use Andersen's algorithm to establish the insensitive value flow of the program to ensure that CEGAS could build a spatial flow model of the program within linear time. As path-sensitive strong updates designed in this paper are not context-sensitive, and CEGAS maps the called function code in LLVM IR to the main function by the inline method, CEGAS only needs to analyze the spatial flow model of the main function. In addition, CEGAS as implemented in this paper mainly focuses on integer variables, not supporting floating-point ones for the moment.

To verify the effectiveness of the designed CEGAS, we establish a series of new benchmark codes considering the existing C-program verification benchmarks. SV-COMP[33] provides the benchmark library sv-benchmark (https://github.com/sosy-lab/sv-benchmarks) for verification of programs (C and Java programs), but most of the benchmark codes are mainly used for logical verification at the symbolic level. For this reason, this paper combines sv-benchmarks with Test-Suite, a benchmark library for evaluating capabilities to analyze point-to relationship of pointers in C/C++ programs (https://github.com/SVF-tools/Test-Suite), and obtains a series of C-program benchmark codes including multiple pointer operations to evaluate the verification accuracy and efficiency of model checking algorithms (open sources of the benchmark library can be found in the link: https://github.com/SFChecker/Benchmark). We design the benchmark codes by combining pointer operations with other different syntactic and semantic features of the C language, and according to the included features, these benchmark codes are divided into six groups: array, function calling (callsite), global variable (global), loop, path, and structure (struct). Specifically, variables of array codes are mainly arrays, and each code contains at least one complex variable, such as array pointers, pointer arrays, structure array pointers, and pointer structure fields, together with related instructions. The main semantic feature contained in callsite codes is the address passing of local or global pointer variables in function calls, and part of the code uses recursive functions and function pointers. Global codes are mainly used to analyze the logical correctness of global variables (such as integer variables, pointer variables, and structure pointers) during intra-procedural and inter-procedural passing. Loop codes mainly use loop instructions (such as for and while) to evaluate the capability of software verification tools to check the correctness of program state changes during loops, especially the semantics of pointer operations with loops. Similar to the code example in Fig. 1(a), path codes contain multiple nested path-condition branches, and each branch contains various pointer-operation instructions. Variables of struct codes are mainly structures, and fields of the structures include integer variables, pointer variables, substructures, and arrays to evaluate the capabilities of field-sensitive analysis. On the basis of these features, we design these benchmark library codes to verify that the reachability of SV-COMP is the main verification property. In other words, with the function calling of *__VERIFIER_error*() set at a specific location of the code, the benchmark library aims to verify whether the function *__VERIFIER_error*() is reachable from the entry point of the main function in the target code. The benchmark library can provide analysis cases at both symbol and address-space levels for evaluating the verification capabilities of model

checking algorithms.

On the basis of the above benchmark library, we compare several mainstream software-verification tools with CEGAS, including CPAChecker[9], Gazer[11], and SMACK[18]. All of these tools have achieved excellent results in SV-COMP, and similar to CEGAS, they all support model checking for the C-program LLVM IR code. CPAChecker is a C-program verifier which supports parallel checking of multiple verification algorithms and is implemented in Java. As CEGAS mainly employs explicit-value analysis, we configured CPAChecker as three different models with different modes, i.e., single explicit-value analysis (CPA-V), the combination of explicit-value and predicate analysis (CPA-VP), and the combination of k-induction, explicit-value analysis, and predicate analysis (the default configuration of CPA-KVP and CPAChecker in SV-COMP). Gazer consists of an LLVM-based front-end Gazer which is able to transform C programs into CFA (implemented in C++) and a model checking framework Theta (implemented in Java). It employs CEGAR based on the combination of explicit-value and predicate analysis for model checking C programs. SMACK (implemented in the C language) is a modular tool chain for software verification. With the help of DSA[34], it computes all pointer alias relations in LLVM IR before hand, and then it carries out model checking by a Boogie verifier after converting LLVM IR and points-to information into the intermediate verification language Boogie[35]. All comparative analysis was conducted on a Linux host with a 3.7 GHz Intel i9-10900K CPU and 64 GB memory.

## 4.2 Analysis of experimental results

During the C-program verification by the above checking tools including CEGAS, if a valid counterexample path is detected, the tools will report Unsafe, and if no counterexample path is detected, they will report Safe; if these tools fail to proceed with the anaysis due to their failure to confirm the validity of a current counterexample path, they will report Unknown. Meanwhile, taking account of the small size of the benchmark library used in this paper, we set a failure time of 1 min. In other words, if the verification tools yield no results within 1 min, the analysis will be interrupted, with the verification time named Timeout and the verification result set to Unknown. In addition, some verification tools do not support some C statements (for example, Gazer does not support CallInst), directly throwing exceptions. In this case, we call the verification time Failed and the verification result Unknown. Limited by space, we only listed partial verification results and time in Table 2. For complete results, please see https://github.com/SFChecker/Benchmark.

In Table 2, GT refers to the ground truth, i.e., the real existence of counterexamples in the benchmark library. It can be seen that the verification result and time of the different tools change with respect to different benchmark codes, and due to the existence of many pointer operations in these benchmark codes, the existing tools could provide inaccurate or infeasible results. For example, different tools handle function calls differently. Specifically, Gazer supports function-calling analysis poorly, with the lowest accuracy in analyzing callsite codes. CEGAS manages the calls context by the inline method, and experimental analysis reveals that this method is effective in most cases, but when there is a recursive function, such as callsite 10.c, CEGAS will produce erroneous results. With limited support for recursive functions, SMACK fails to verify properties in callsite 10.c correctly. Path codes generally contain many conditional branch statements, and these branch statements contain many pointer reference and dereference instructions. CEGAS can effectively update points-to relations of pointers through counterexample-guided sparse value-flow strong updates and obtain more accurate checking results. Moreover, the sensitivity of each tool to fields of structures varies, which results in different analysis accuracy. For example, CPA-V has low verification accuracy for array 6.c, array 9.c (including structure array pointers), and struct codes. Using field-sensitive analysis, CEGAS can accurately analyze

**Table 2** Verification results and time (in milliseconds) of benchmark codes

| Benchmark | Tool | | | | | | |
|---|---|---|---|---|---|---|---|
| | GT | CPA-V | CPA-VP | CPA-KVP | SMACK | Gazer | CEGAS |
| array/array0.c | **Unsafe** | **Unsafe/460** | **Unsafe/132** | **Unsafe/589** | **Unsafe/1165.1** | **Unsafe/24.9** | **Unsafe/10.7** |
| array/array4.c | **Safe** | Unknown/351 | **Safe/141** | **Safe/328** | **Safe/1426.8** | **Safe/16.8** | **Safe/17.1** |
| array/array6.c | **Safe** | Unsafe/424 | Unsafe/156 | Unsafe/352 | Unsafe/1158.3 | Unsafe/77.6 | **Safe/19.0** |
| array/array9.c | **Safe** | Unknown/504 | Unsafe/250 | Unsafe/367 | Unsafe/1583.8 | **Safe/19.9** | **Safe/38.5** |
| callsite/callsite0.c | **Safe** | Unknown/205 | **Safe/95** | **Safe/206** | **Safe/1387.3** | Unknown/Failed | **Safe/31.9** |
| callsite/callsite7.c | **Unsafe** | Unknown/196 | **Unsafe/95** | **Unsafe262** | **Unsafe/1582.7** | **Unsafe/60.5** | **Unsafe/24.5** |
| callsite/callsite10.c | **Unsafe** | **Unsafe/209** | **Unsafe/79** | **Unsafe/586** | Safe/938.2 | Unknown/Failed | Safe/71.9 |
| callsite/callsite15.c | **Unsafe** | **Unsafe/244** | **Unsafe/129** | **Unsafe/334** | **Unsafe/1203.9** | **Unsafe/42.7** | **Unsafe/32.2** |
| global/global1.c | **Unsafe** | **Unsafe/208** | **Unsafe/92** | **Unsafe/230** | **Unsafe/1116.4** | Unknown/21.3 | **Unsafe/15.1** |
| global/global3.c | **Unsafe** | Unknown/201 | **Unsafe/95** | **Unsafe/224** | **Unsafe/1126.2** | **Unsafe/35.4** | **Unsafe/17** |
| global/global4.c | **Unsafe** | **Unsafe/194** | **Unsafe/96** | **Unsafe/223** | **Unsafe/1162** | Unknown/49.2 | **Unsafe/20** |
| global/global7.c | **Unsafe** | **Unsafe/200** | **Unsafe/93** | **Unsafe/238** | **Unsafe/1158.8** | **Unsafe/57.8** | **Unsafe/19.3** |
| loop/loop1.c | **Unsafe** | Unknown/233 | Unknown/Failed | **Unsafe/446** | Safe/1476.8 | **Unsafe/179.8** | **Unsafe/68.1** |
| loop/loop3.c | **Safe** | Unknown/490 | Unknown/Failed | **Safe/5832** | **Safe/1518.1** | Unknown/159 | **Safe/335.1** |
| loop/loop11.c | **Unsafe** | Unknown/245 | Safe/370 | Safe/2344 | Safe/1542.9 | **Unsafe/27.3** | **Unsafe/92.2** |
| loop/loop15c | **Safe** | Unknown/266 | Unknown/Failed | Unknown/Timeout | **Safe/1919.5** | Unsafe/404.2 | **Safe/506.5** |
| path/path1.c | **Safe** | Unsafe/518 | Unsafe/271 | Unsafe/519 | Unsafe/1156.7 | Unknown/54.3 | **Safe/30.9** |
| path/path6.c | **Safe** | Unknown/507 | Unsafe/229 | Unknown/Failed | Unsafe/1193.2 | Unsafe/70.6 | **Safe/29.6** |
| path/path13.c | **Safe** | Unknown/599 | Unsafe/304 | Unsafe/858 | Unsafe/1938.3 | **Safe/17.7** | Unsafe/18.6 |
| path/path25.c | **Safe** | Unknown/570 | **Safe/269** | **Safe/524** | **Safe/1770.6** | Unsafe/79.6 | **Safe/29.5** |
| struct/struct1.c | **Unsafe** | **Unsafe/220** | **Unsafe/97** | **Unsafe/229** | **Unsafe/1160.4** | **Unsafe/65.1** | **Unsafe/29.9** |
| struct/struct5.c | **Safe** | Unsafe/230 | Unsafe/110 | Unsafe/230 | Unsafe/1182.5 | Unsafe/78.3 | **Safe/22.1** |
| struct/strcut7.c | **Safe** | Unknown/192 | Unsafe/83 | **Safe/184** | Unsafe/1179.1 | Unsafe/73.3 | **Safe/17.9** |
| struct/struct14.c | **Safe** | Unknown/189 | **Safe/74** | **Safe/189** | **Safe/1375.8** | **Safe/19.1** | Unsafe/15 |

most structure-related benchmark codes, but it may produce misjudgments when structures are combined with static variables or function calls. Among the six groups of benchmark codes, loop codes have high verification-time overhead, and CEGAS limits the number of times of loop unrolling. Due to the small number of loops in loop codes, CEGAS can yield correct verification results. In addition, for the experimental benchmark library, explicit-value analysis of CPAChecker is not available for effective model checking in most cases due to the lack of correct points-to information.

Figures 5 and 6 illustrate the overall effect of each tool on the benchmark library used in the experiment. Specifically, Fig. 5 shows the verification accuracy of each tool with respect to each type of benchmark code, and it can be seen that CEGAS achieves high accuracy in the six types of benchmark codes. Among the three verification strategies of CPAChecker, i.e., CPA-V, CPA-VP, and CPA-KVP, CPA-KVP combining explicit-value analysis, predicate analysis, and k-induction can obtain more accurate verification results; however, CPAChecker only supports Anderson and Steensgaard to obtain pointer relations. As shown in Fig. 5, CPA-V has low

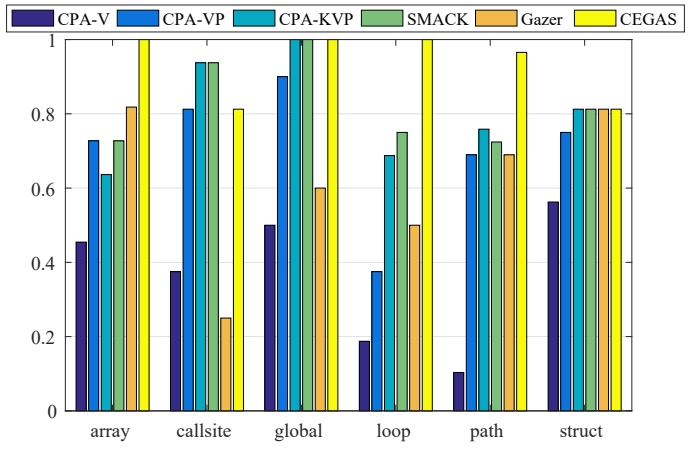


**Figure 5** Verification accuracy of existing tools and CEGAS

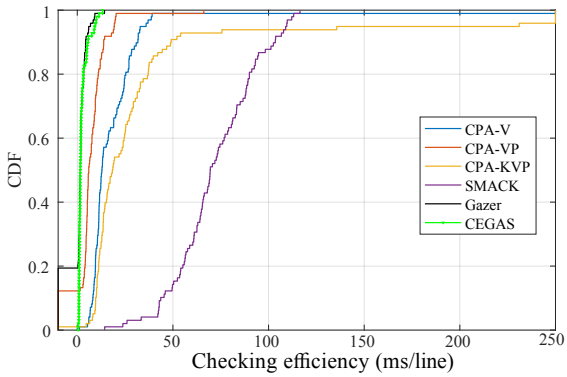


**Figure 6** Verification efficiency of existing tools and CEGAS. CDF: cumulative distribution function

verification accuracy with respect to the benchmark library used in the experiment. By adding predicate path constraints, CPA-VP can improve the accuracy of CPA-V (single explicit-value analysis) to some extent. With the external pointer-analysis tool DSA, SMACK computes the points-to information in advance to provide effective information for model checking, which makes SMACK obtain higher accuracy. Similar to CPA-VP, Gazer improves the accuracy of the analysis by combining explicit-value analysis with predicate path constraints. However, we can see that the accuracy of these tools varies greatly in these six types of benchmark codes. By combining explicit-value analysis with pointer analysis, CEGAS can achieve the highest accuracy in the benchmark codes except for callsite. CEGAS uses the inline method to solve function calling, but when there is a recursive function in callsite codes, this method cannot remove function calls, which might result in invalid results of CEGAS.

Figure 6 illustrates the cumulative distribution of the verification efficiency of the different tools. We evaluated the verification efficiency by dividing the verification time (measured inmilliseconds) by the number of lines of the checked code, and a higher value indicates lower efficiency. For verification results of Timeout, we set the efficiency to 250, while for those of Failed, we set the efficiency to –10. It can be seen that SMACK demonstrates the lowest verification efficiency, followed by the three strategies of CPAChecker, while CEGAS and Gazer have similar efficiency. Different from other tools, SMACK transforms optimized LLVM IR into Boogie and then directly calls the Boogie verifier for verification, which leads to inefficient verification. Other tools all adopt abstraction-based verification strategies, with SMT solvers as constraint solvers, and thus the verification efficiency is promoted. By combining explicit-value analysis with predicate analysis, both CPA-VP and Gazer achieve excellent verification efficiency. By adding the additional checking of k-induction to CPA-VP, CPA-KVP improves its verification accuracy but reduces its verification efficiency. Multiple cases with timeouts or unsupported verification exsit in all CPAChecker's three verification strategies and Gazer, with Gazer having the highest failure rate (20%) and CPA-KVP having the most timeouts (four times). Both SMACK and CEGAS, on the other hand, are able to produce verification results within a certain time. With the combination of explicit value analysis and sparse value-flow analysis, CEGAS can provide accurate state changes at both symbolic-variable and address-space levels during model checking, which reduces the analysis overhead caused by erroneous points-to information and improves the verification efficiency of CEGAS.

Finally, we calculated the overall verification accuracy and average verification efficiency of each tool with respect to the benchmark library after removing structures with the verification time of Timeout and Failed, as shown in Table 3. The table indicates that CEGAS yields overall

**Table 3** Overall accuracy and average efficiency of benchmark codes

| Index | CPA-V | CPA-VP | CPA-KVP | SMACK | Gazer | CEGAS |
|---|---|---|---|---|---|---|
| Accuracy (%) | 31.6 | 69.4 | 79.6 | 80.6 | 61.2 | **92.9** |
| Average efficiency (ms/line) | 16.6 | 9.0 | 25.52 | 72.3 | 2.64 | **2.58** |

verification accuracy of 92.9% and average verification efficiency of 2.58 ms/line on the used benchmark library, superior to other compared tools. In conclusion, compared with the existing C-code model checking algorithms, the counterexample-guided spatial-flow model checking method CEGAS designed through the combination of explicit-value analysis and sparse value-flow analysis can realize more accurate and efficient verification for C code containing various pointer operations.

## 5 Conclusions

The existing C-code model checking algorithms fail to effectively analyze state changes at the address-space level, which causes low verification accuracy. In view of this problem, this paper designs a counterexample-guided spatial flow model checking algorithm. First, a spatial flow model is designed, which can effectively describe state changes of programs at the symbolic-variable and address-space levels by combining control flow and sparse value flow. Moreover, a CEGAS algorithm is proposed. Specifically, a spatial flow model is constructed rapidly by insensitive pointer analysis, and the model is abstracted by variable abstraction. Then, detected invalid counterexamples are used to guide the refinement of model abstraction precision and the strong update of insensitive points-to relations. In this way, an effective tradeoff between verification efficiency and accuracy is achieved. Finally, by designing a C-code benchmark library, this paper compares the proposed method with the existing cutting-edge verification tools. In multiple C-language benchmark codes, the proposed method achieves outstanding results in terms of verification accuracy and efficiency.

Admittedly, the proposed method still has many shortcomings and requires the following improvements in future work: (1) the proposed method solves the context problem by the inline method, but it has some defects. In future work, path-sensitive strong updates should be improved to support context sensitivity. (2) it is found in the experiments that CPAChecker and Gazer with the combination of explicit-value and predicate analysis can obtain more accurate results than methods with pure explicit-value analysis. In the future, CEGAS can be improved to support predicate analysis. (3) CEGAS needs to call the SMT solver in many cases, including state abstraction, path feasibility analysis, and branch-condition judgment, which results in high overhead. Some constraints (such as branch-condition judgment) can be solved by local calculation to improve efficiency, such as the method adopted in Ref. [27].

## References

[1] D'silva V, Kroening D, Weissenbacher G. A survey of automated techniques for formal software verification. IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems, 2008, 27(7): 1165–1178. [doi: 10.1109/TCAD.2008.923410]

[2] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM), 2003, 50(5): 752–794. [doi: 10.1145/876638.876643]

[3] Baldoni R, Coppa E, D'Elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 2018, 51(3): 1–39. [doi: 10.1145/3182657]

[4] Clarke EM, Henzinger TA, Veith H, Bloem R. Handbook of Model Checking. Cham: Springer, 2018.

[5] Kalra S, Goel S, Dhawan M, Sharma S. Zeus: Analyzing safety of smart contracts. Proc. of the Symp. on Network and Distributed Systems Security (NDSS). 2018. 1–12. [doi: 10.14722/ndss.2018.23082]

[6] Yu Y, Li Y, Hou K, Chen Y, Zhou H, Yang J. CellScope: Automatically specifying and verifying cellular network protocols. Proc. of the ACM SIGCOMM 2019 Conf. on Posters and Demos. 2019. 21–23. [doi: 10.1145/3342280.3342294]

[7] Chaki S, Datta A. ASPIER: An automated framework for verifying security protocol implementations. Proc. of the 22nd IEEE Computer Security Foundations Symp. IEEE, 2009. 172–185. [doi: 10.1109/CSF.2009.20]

[8] Zhang XL, Zhu YF, Gu CX, Chen X. C2P: Formal abstraction method and tool for C protocol code based on Pi calculus. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1581–1596 ((in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6238. htm [doi: 10.13328/j.cnki.jos.006238]

[9] Beyer D, Keremoglu ME. CPAchecker: A tool for configurable software verification. Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Berlin, Heidelberg: Springer, 2011. 184–190. [doi: 10.1007/978-3-642-22110-1_16]

[10] Holzmann GJ. Software model checking with SPIN. Advances in Computers, 2005, 65: 77–108. [doi: 10.1016/S0065-2458(05)65002-4]

[11] Ádám Z, Sallai G, Hajdu Á. Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2021. 433–437.

[12] Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 2022. 58–70. [doi: 10.1145/565816.503279]

[13] Henzinger TA, Jhala R, Majumdar R, McMillan KL. Abstractions from proofs. Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 2004. 232–244. [doi: 10.1145/982962.964021]

[14] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. DIKU: University of Copenhagen, 1994.

[15] Beyer D, Dangl M, Wendler P. A unifying view on SMT-based software verification. Journal of Automated Reasoning, 2018, 60(3): 299–335. [doi: 10.1007/s10817-017-9432-6]

[16] Musuvathi M, Park DYW, Chou A, Engler DR, Dill DL. CMC: A pragmatic approach to model checking real code. Proc. of the Operating Systems Design and Implementation (OSDI). 2002. 75–88. [doi: 10.1145/844128.844136]

[17] Havelund VK, Brat G, Park S, Lerda F. Model checking programs. Automated Software Engineering (ASE), 2003, 10(2): 203–232. [doi: 10.1023/A:1022920129859]

[18] Rakamarić Z, Emmi M. SMACK: Decoupling source language details from verifier implementations. Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Cham: Springer, 2014. 106–113. [doi: 10.1007/978-3-319-08867-9_7]

[19] Clarke EM, Kroening D, Lerda F. A tool for checking ANSI-C programs. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2988, Springer, 2004. 168–176. [doi: 10.1007/978-3-540-24730-2_15]

[20] Gadelha MR, Monteiro F, Cordeiro L, *et al*. ESBMC v6.0: Verifying C programs using k-induction and invariant inference. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer, 2019. 209-213.

[21] Ball T, Levin V, Rajamani SK. A decade of software model checking with SLAM. Communication of ACM, 2011, 54(7): 68–76.

[22] Tóth T, Hajdu Á, Vörös A, *et al*. Theta: A framework for abstraction refinement-based model checking. Proc. of the 2017 Formal Methods in Computer Aided Design (FMCAD). IEEE, 2017. 176–179. [doi: 10.23919/FMCAD.2017.8102257]

[23] Steensgaard B. Points-to analysis in almost linear time. Proc. of the Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). 1996. 32–41. [doi: 10.1145/237721.237727]

[24] Oh H, Heo K, Lee W, Lee W, Yi K. Design and implementation of sparse global analyses for C-like languages. Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2012. 229–238. [doi: 10.1145/1273442.1250789]

[25] Sui Y, Xue J. Value-flow-based demand-driven pointer analysis for C and C++. IEEE Trans. on Software Engineering, 2018, 46(8): 812-835. [doi: 10.1109/TSE.2018.2869336]

[26] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. Proc. of the 28th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM, 2007. 480–491.

[27] Shi Q, Xiao X, Wu R, *et al*. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2018. 693–706. [doi: 10.1145/3192366.3192418]

[28] Shi Q, Yao P, Wu R, *et al*. Path-sensitive sparse analysis without path conditions. Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI). 2021. 930–943. [doi: 10.1145/3453483.3454086]

[29] Beyer D, Löwe S. Explicit-state software model checking based on CEGAR and interpolation. Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE). Berlin, Heidelberg: Springer, 2013. 146–162. [doi: 10.1007/978-3-642-37057-1_11]

[30] Sui Y, Xue J. SVF: Interprocedural static value-flow analysis in LLVM. Proc. of the 25th ACM Int'l Conf. on Compiler Construction (CC). 2016. 265–266. [doi: 10.1145/2892208.2892235]

[31] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. Proc. of the Int'l Symp. on Code Generation and Optimization (CGO). 2004. 75–86. [doi: 10.1109/CGO.2004.1281665]

[32] de Moura LM,Bjørner N. Z3: An efficient SMT solver. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2008. 337–340. [doi: 10.1007/978-3-540-78800-3_24]

[33] Int'l competition on software verification (SV-COMP).http://sv-comp.sosy-lab.org

[34] Lattner C, Lenharth A, Adve V. Making context-sensitive points-to analysis with heap cloning practical for the real world. Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). 2007, 42(6): 278–289. [doi: 10.1145/1273442.1250766]

[35] De Line R, Leino KRM. BoogiePL: A typed procedural language for checking object-oriented programs [Technical Report], MSR-TR-2005-70, Microsoft Research, 2005.

**Yinbo Yu**, Ph.D., associate professor. His research interests include software and system security, Internet of Things security, and deep-learning security.

**Dejun Mu**, Ph.D., professor, doctoral supervisor. His research interests include hardware security, machine learning, and data mining.

**Jiajia Liu**, Ph.D., professor, doctoral supervisor. His research interests include network security and intelligent communications.