Vetting API Usages in C Programs with IMChecker

Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Ming Gu, Jiaguang Sun
School of Software, Tsinghua University, Beijing China
guzx14@mails.tsinghua.edu.cn

***Abstract*—Libraries offer reusable functionality through Application Programming Interfaces (APIs) with usage constraints, such as call conditions or orders. Constraint violations, i.e., API misuses, commonly lead to bugs, even security issues. The existing approaches to API-misuse detection either suffer from the sparse usage problem (i.e., bugs that rarely occur) or report false alarms due to inaccurate semantics. In this paper, we introduce IMChecker, a constraint-directed static analysis toolkit to vet API usages in C programs powered by a domain-specific language (DSL) for specifying API usages. First, we propose a DSL which covers a majority of API usage constraint types and enables straightforward but precise specification through studying real-world API-misuse bugs patches. Then, we design and implement IMChecker to automatically parse specifications into checking targets and employ a static analysis engine to identify potential API misuses and prune false positives with rich semantics. We have instantiated IMChecker for C programs with user-friendly graphic interfaces and conduct evaluation on widely-used benchmarks and real-world projects such as Linux kernel and OpenSSL. The results show that IMChecker outperforms 4.78%-36.25% in precision and 40.25%-55.21% w.r.t. state-of-the-arts. We also found 56 previously unknown bugs, 36 of which have been confirmed by the corresponding development communities.**
Video: https://youtu.be/YGDxeyOEVIM
Repository: https://github.com/tomgu1991/IMChecker

***Index Terms*—API Misuse Detection, Domain-Specific Language, Static Analysis**

I. Introduction

Libraries provide Application Programming Interfaces (APIs) to increase productivity, and these APIs often have usage constraints, such as restrictions on call orders or call conditions. Violations of these constraints, called API misuse [1], is a prevalent cause of software bugs, crashes, and vulnerabilities [2]. To understand the nature of API-misuse bugs that occur in widely used C programs, we conduct an empirical study of API-misuse bugs and fixes (The details of our empirical study can be found in our repository). The results indicate that API misuses usually occur due to the following three reasons: Incorrect/Missing Parameter Using (IPU), Incorrect/Missing Error Handling (IEH) and Incorrect/Missing Causal Calling (ICC). Figure 1 shows an example of these misuse bugs, where missing parameter validation of *fopen* at Line-5 will result in a null pointer dereference bug (IPU), returning SUCCESS when *fopen* fails at Line-10 and incorrect checking the error code status of *fgets* at Line-15 will break the error handling mechanism (IEH), and missing to close the opened file handler 'pFile' at Line-19 will cause a memory leak bug (ICC).

Many different tools, techniques and methodologies have been proposed to address the above problems, especially approaches with code mining and static analysis techniques have proven to be effective. For example, PR-Miner [3] encodes

```
1   int foo(char *fileName){
2      char buffer[100] = "";
3      FILE *pFile;
4      // 1. missing parameter validation, resulting NPD
5   +  if(fileName == NULL) return ERROR;
6      pFile = fopen(fileName, "r");
7      if (pFile == NULL){
8         Log("Error open file");
9      // 2.1 incorrect error propagation
10  -     return SUCCESS
11  +     return ERROR;
12     }
13     // 2.2 incorrect error code status checking
14  -  if (fgets(buffer, 100, pFile) < 0){
15  +  if (fgets(buffer, 100, pFile) == NULL){
16        Log("Error read file");
17     // 3. incorrect causal calling, resulting memory_leak
18  +     fclose(pFile);
19        return ERROR;
20     }
21     ...
22     fclose(pFile);
23     return SUCCESS;
24  }
```

Fig. 1: Example of four most common API-misuse bugs.

usages as the set of all function names and employs frequent-itemset mining to find violations with a minimum support of 15 usages of a single API. Ray et al. [4] propose ErrDoc to explore error handling bugs by under-constrained symbolic execution using specification, and Yun et al. [5] present APISan for causal relation and semantic relation on arguments by semantic cross-checking on intraprocedural paths.

Despite the vast amount of work on API-misuse detection, these bugs still exist widespread in practice [6], [7]. Basing on the performance of existing detectors, we observe that there are two main obstacles to efficient API-misuse detection:

- Sparse usage problem. Mining techniques leverage the key idea that correct usages appear frequently in large corpora and deviations are bugs with a minimum support predefined to filter out false positives, such as PR-Miner only analyzing usages over 15. However, a challenge for such a belief is the "sparse usage problem" [8], where these tools will miss API-misuse bugs under the usage threshold, which is severe for program-specific APIs.
- Insufficient semantic analysis. Most methods are built on abstract syntax trees, namely, matching the usage based on syntactic information with less semantic information, such as Errdoc only supporting error handling with constant integers. Moreover, they usually apply an intraprocedural analysis strategy, as APISan employing, to address the path-exploration problem in large-scale source code. These approaches will generate false positives and false negatives when the usage contains point-to relationship or cross functions (such as *malloc* and *free* in two separate functions).

In this paper, we present IMChecker, a constraint-directed static analysis toolkit to augment the current API-misuse detection abilities for large-scale C programs. We propose a domain-specific language IMSpec, to specify common API usage constraints leveraging the empirical study result of real-world API-misuse bug patterns. In this way, we provide an explicit API usage pattern to direct bug detection in order to overcome sparse usage problems. Then, we design and implement IMChecker, a static analysis engine which automatically parses IMSpec into checking targets and detects API-misuse bugs. IMChecker employs under-constrained symbolic execution [9] to address large-scale programs and performs a soundy [10] approach which means that our technique is mostly sound in order to achieve a higher precision. For potential API misuses, IMChecker prunes false positives by rich semantics and multiple usage instances. We package IMChecker static analysis engine with two user-friendly GUI for IMSpec creation and bug detection results audit as a toolkit, which can be used in a single command line.

For evaluation, we conduct our experiments on a widely-used benchmark Juliet Test Suite [11]. The results demonstrate that IMChecker has a better performance in precison and recall, improving 4.78%-36.25% and 40.25%-55.21% w.r.t. the state-of-the-arts. We also apply IMChecker to real-world projects, such as Linux kernel, Openssl and 5 packages in Ubuntu which use OpenSSL library. IMChecker detects 56 previously unknown bugs, 36 of which have been confirmed or fixed by the corresponding development commutities.

II. IMChecker Design

A. Framework

As presented in Figure 2, IMChecker consists of three components. First, **Preprocessor** parses the source code into an extended Control Flow Graph (CFG), and verifies the API usage specification defined in IMSpec language. Then, **Static Analysis Engine** employs the CFG and specifications to select target analysis entries, collect path traces with rich semantic and detect API-misuse bugs along the traces. Finally, **Result Generator** filters the bug detection results according to semantic information and usage statistics and produces the final bug reports.

Preprocessor. Input of our tool contains two parts, source code and target APIs specification. First, Preprocess will parse the source code into Control Flow Automata (CFA), an extension of CFG, where we classify the edges into two types: (1) *ControlEdge* to carry the concrete statements to conduct semantic computation for static analysis; (2) *SummaryEdge* to maintain program summary information pre-computed to skip loops and function calls for large-scale programs. Then, Preprocessor will parse the IMSpec specification according to the syntax of IMSpec and verify the semantic conflicts among specification itself. In order to help users build specifications, we have implemented a GUI client name IMSpec Writer (see Section II-C). More details of IMSpec, including the syntax, the semantic and examples can be found in our repository.

Static Analysis Engine. Static analysis engine is build to conduct API-misuse bug detection using the CFA and the specificiation. Similar to the traditional static analysis, the key challenge of large and complex programs is to overcome the path-explosion problem. We make two design decisions to achieve scalability without sacrificing substantial accuracy: limiting inter-procedural analysis and unrolling loops according to preliminary experiments. In details, our engine consists of three steps. (1) An entry selection algorithm is proposed to select analysis targets. (2) Then, our engine performs under-constrained symbolic execution to generate program path traces that capture rich semantic information for each target API defined in the specifications. (3) Finally, engine employs the constraints defined in IMSpec and the program path traces to detect API-misuse bugs.

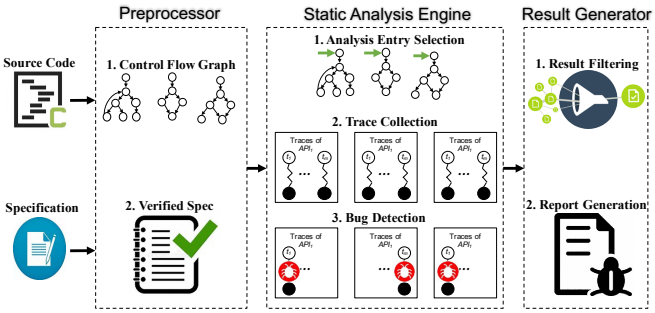


Fig. 2: Framework of IMChecker

Result Generator. To achieve the scalability for real-world programs, we employ a limiting inter-procedural strategy to address the path-explosion problem. Taken the bug traces generated from static analysis engine, result generator will filter out the false positives according to inter-procedural semantics and usage statistics first. In the end, we produce the final bug report in a well-defined format. We implement a Report Displayer to help user audit the bug report.

B. Implementation

IMChecker is built in Java language. We preprocess the source code into LLVM-IR 3.9[1], which provides a typed, static single assignment (SSA) and well-suited low-level language. We parse the LLVM-IR by javacpp[2] and build CFAs. To this end, we can conduct a path-, field- and context-sensitive analysis. We implement a point-to and range analysis based on abstracted AccessPath proposed by Bodden et al. [12] to collect rich semantic information. Our specification language IMSpec and bug report is formatted in the human-readable data serialization language YAML[3]. We build our GUI using Python-Tkinter package[4].

C. Usage

We package IMChecker with two user-friendly GUI (IMSpec Writer and Report Displayer) into a toolkit, which can be used in simple command lines. Our toolkit can be used by the following three steps (employing the code in Figure 1 as an example):

[1]http://releases.llvm.org/3.9.0/docs/ReleaseNotes.html
[2]https://github.com/bytedeco/javacpp
[3]http://yaml.org/
[4]https://docs.python.org/2/library/tkinter.html

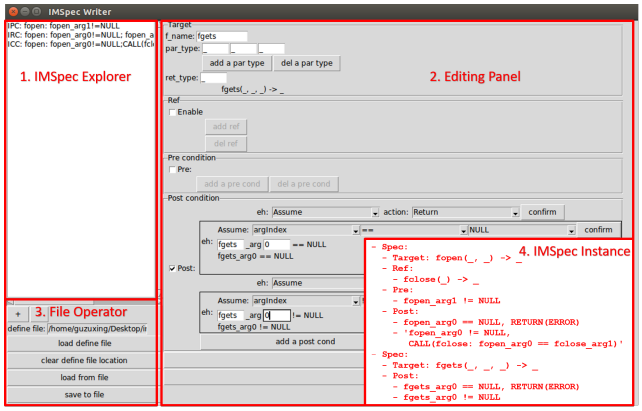
```
1  ubuntu@~: python3 imspec_writer.py
2  ubuntu@~: python3 engine.py --spec=spec.yaml
       [--specDefine=define.h] --input=example.c
3  ubuntu@~: python3 report_displayer.py
```

Step1: ***imspec_writer.py*** is used to call the writer client as shown in Figure 3a. We produce the specification into YAML format as illustrated in the *IMSpec Instance* box. Therefore, users can direct write specification according to IMSpec syntax.

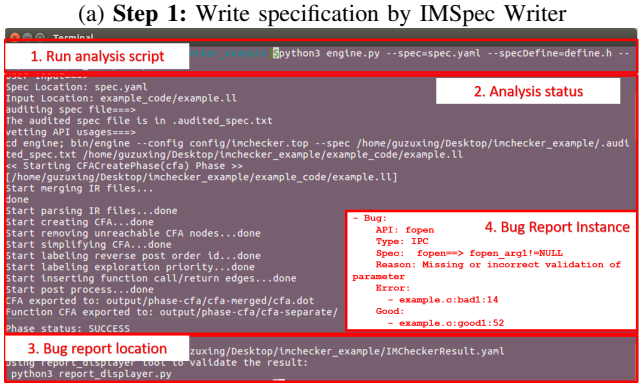
Step2: ***engine.py*** is used to call the static analysis engine. Engine requires the specification 'spec.yaml' and a compilable c file or a LLVM-IR file which is compiled by clang with '-S -emit-llvm -g' options. Parameter 'specDefine' is used to import macros defined in 'spec.yaml', such as including error codes. For real-world projects that can be built by clang, we provide a build-capture tool 'bcmake' to generate the input files (see our repository for details). We output the analysis status into Terminal as shown in Figure 3b. Bug detection results are ourputed in the format of Yaml as presented in the *Bug Report Instance* box.

Step3: ***report_displayer.py*** is used to call the bug report visualizer to audit the results as shown in Figure 3c. When users select a target API, Report Displayer will list all the potential bugs on the left Bug List panel. For each bug instance, we provide the bug description as well as reference usages on the right Ref List panel.

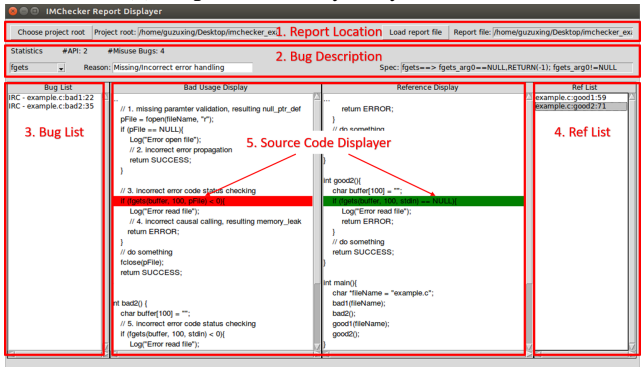
For more details of IMSpec and our tools, users can refer to the user manual in our repository at https://github.com/tomgu1991/IMChecker.



(a) **Step 1:** Write specification by IMSpec Writer



(b) **Step 2:** Run analysis by IMChecker



(c) **Step 3:** Audit detection results by Report Displayer

Fig. 3: Usage of IMChecker Toolkit

III. Evaluation

We evaluate IMChecker on a controlled dataset from Juliet Test Suite benchmark. IMChecker is compared with two state-of-the-art tools, APISan[5] and Clang Static Analyzer[6] which are both designed for multiple types of API-misuse bugs and frequently mentioned in other works. (Errdoc performs well on real-world projects, but only supports IEH bugs.) We also apply IMChecker to latest versions of real-world projects, including Linux kernel-4.18-rc4, OpensSSL-1.1.1-pre8 and 5 packages using OpenSSL library (dma, exim, hexchat, httping and open-vm-tools) in Ubuntu 16.04. All the tools ran on Ubuntu 16.04 LTS (64-bit) with a Core i5-4590@3.30GHz Intel processor and 16GB memory.

TABLE I: Evaluation Results on API-Misuse Benchmark

Case Info		APISan		Clang-SA		IMChecker	
Type	Total	Report	TP	Report	TP	Report	TP
IPU	510	310	0	107	105	490	423
IEH	612	446	173	0	0	580	506
ICC	1050	447	435	710	565	1012	878
Total	2172	1203	608	817	670	2082	1807
Precision%		50.54		82.01		86.79	
Recall(A-R^{α})%		27.99-36.58		30.84-42.95		83.20-83.20	

A-R^{α} is recall results with **A**ll cases considered and a **R**efined result where we remove the type a tool fails to detect all of them, such as IPU for APISan.

Table I shows the controlled evaluation results. In total, we select 2172 single cases, each of which contains a bug and several correct usages to evaluate precision and recall. It covers three types of API-misuse bugs and 13 different Common Weakness Enumeration (CWE) types[7]. From the true positive (TP) columns of each tool, we can observe that APISan fail to detect the IPU bugs and Clang-SA fails in IEH on this dataset. We investigate the cases and algorithms behind the tools. The result shows that APISan only supports the bugs with an explicit validation checking indicating it will fail in bugs requiring a semantic inferring, such as CWE-590 free-memory-not-on-heap[8]. Even though Clang-SA provides a large number of checkers targeted at finding API usage bugs[9], it fails to detect error handling bugs. Similar

[5]https://github.com/sslab-gatech/apisan

[6]http://clang-analyzer.llvm.org/

[7]https://cwe.mitre.org/index.html, (IPU-CWE121/122/131/476/590; IEH-CWE252/253/390; ICC-CWE401/404/415/690/775)

[8]https://github.com/sslab-gatech/apisan/issues/7

[9]http://clang-analyzer.llvm.org/available_checks.html

TABLE II: List of Confirmed New Bugs

	Project	Issue	Misused API	S.
1	Linux	200505	pktcdvd.c: **alloc_disk**	✓✓
2	Linux	200533	ncsi-netlink.c: **nla_nest_start**	✓✓
3	Linux	200535	conntrack.c: **nla_nest_start**	✓✓
4	Linux	200551	dpc.c: **pci_find_ext_capability**	✓✓
5	Linux	200561	namespace.c: **ida_pre_get**	✓✓
6	Linux	200489	team.c: **__send_and_alloc_skb**	✓
7	Linux	200511	clk-pxa.c: **kzalloc**	✓
8	Linux	200541	tlb_uv.c: **kzalloc**	✓
9	OpenSSL	6567	speed.c: **RAND_bytes**	✓✓
10	OpenSSL	6568	tasn_utl.c: **ASN1_INTEGER_get**	✓✓
11	OpenSSL	6569	p12_init.c: **ASN1_INTEGER_get**	✓✓
12	OpenSSL	6572	t1_lib.c: **BN_set_word()**	✓✓
13	OpenSSL	6574	statem_srvr.c: **EVP_PKEY_*_DH**	✓✓
14	OpenSSL	6781	ec_ameth.c: **EC_*_curve_name**	✓✓
15	OpenSSL	6789	p12_init.c: **ASN1_INTEGER_set**	✓✓
16	OpenSSL	6820	ts_lib.c: **ASN1_INTEGER_to_BN**	✓✓
17	OpenSSL	6822	rsa_ossl.c: **BN_sub**	✓✓
18	OpenSSL	6973	ocsp_srv.c: **EVP_MD_CTX_new**	✓✓
19	OpenSSL	6977	pk7_lib.c: **ASN1_INTEGER_set**	✓✓
20	OpenSSL	6982	asn_moid.c: **OBJ_nid2obj**	✓✓
21	OpenSSL	6983	bn_x931p.c: **BN_sub**	✓✓
22	dma	59	crypto.c: **SSL_connect**	✓✓
23	exim	2317	tls-openssl.c: **SSL_CTX_*_list**	✓✓
24	hexchat	2244	dh1080.c: **BN_set_word**	✓
25	httping	41	mssl.c: **SSL_CTX_new**	✓
26	open-vm-tools	291	sslDirect.c: **SSL_CTX_set*list**	✓✓
27	open-vm-tools	292	certverify.c: **X509_STORE_*cert**	✓✓

We are creating issues and patches for all of 56 previously unknown bugs (listed in our repository). We list part of confirmed ones above where ✓is confirmed and ✓✓is already applied to mainline branch.

to the most of universal static anlaysis tools, it hard-codes detecting algorithm and takes little program-specific semantic into consideration. Moreover, Clang-SA prefers a conservative strategy that they only report bugs with high confidence to improve the precision. Leveraging a DSL to capture program-specific properties and a static analysis engine to compute rich semantic, IMChecker finds 1137-1199 more bugs with a more accurate result, improving 4.78%-36.25% in precision and 40.25%-55.21% in recall.

The main motivation of IMChecker is to detect API-misuse bugs in real-world programs. In total, IMChecker detects 56 previouly unknown bugs. We are currently attempting to create issues and patches for all the bugs and send them to the developers of each program. Up to now, 36 of the new bugs have been confirmed by the developers. We list part of them in Table II. For example, in Figure 4 we present a memory leak bug found in OpenSSL (Issue #6781), which was fixed in two stable versions and the master branch within eight hours after we submitted the issue with a bug description, explanation of the bug traces and potential fix strategy.

IV. Conclusion

Modern APIs are rapidly evolving and error-prone. Incorrect usages of APIs will cause severe bugs. In this paper, we propose IMChecker to vet API usages in C programs. We evaluated our approach on a widely-used benchmark as well as real-world projects. Our results show that our methods perform better than the current state-of-the-art techniques. We also find 56 previously unknown bugs, 36 of which have already been confirmed by the developers. In the future, we will conduct experiments on more programs and investigate heuristics to automatically rank bug reports based on potential severity.

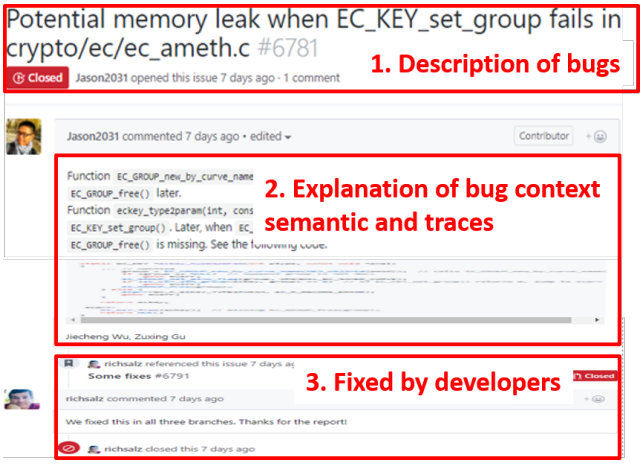


Fig. 4: Screenshot of a memory leak bug found by IMChecker, which is fixed at eight hours on the two stable versions and mainline branch.

References

[1] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 464–467.

[2] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 38–49.

[3] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 306–315.

[4] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 752–762.

[5] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing API usages through semantic cross-checking," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 363–378.

[6] O. Legunsen, W. U. Hassan, X. Xu, G. Rosu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java API specifications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 602–613.

[7] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, pp. 1–1 (Early Access), 2018.

[8] S. K. Samantha, H. A. Nguyen, T. N. Nguyen, and H. Rajan, "Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining," *PACMPL*, vol. 1, no. OOPSLA, pp. 83:1–83:29, 2017.

[9] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 49–64.

[10] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[11] "Juliet test suite," https://samate.nist.gov/SRD/testsuite.php, 2018.

[12] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 619–629.