2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)

intCV: Automatically Inferring Correlated Variables in Interrrupt-Driven Program

Chao Li[1,2], Zhixuan Wang[1,3], Rui Chen[1,2,*], and Mengfei Yang[4]
[1]Beijing Sunwise Information Technology Ltd, China
[2]Beijing Institute of Control Engineering, China
[3]Xidian University, China
[4]China Academy of Space Technology, China
lichao@sunwiseinfo.com, wangzhixuan@sunwiseinfo.com, chenrui@sunwiseinfo.com, yangmf@bice.org.cn
*corresponding author

Abstract—Interrupt-driven programs are extensively employed in safety-critical areas such as aerospace, autonomous driving, and medical equipment. Nevertheless, the uncertainty of interrupt preemption may result in concurrent bugs. Among these concurrent bugs, atomicity violations are critical and challenging to detect. Existing methods mostly concentrate on predicting or detecting single-variable atomicity violations but fail to address the more intricate multi-variable atomicity violations. In real-world programs, many variables are inherently correlated and must be accessed together with their correlated peers consistently. To significantly improve the ability of techniques in inferring correlated variables, this paper conducts an empirical study on real-world software to understand the manifestation characteristics of variable correlations. Building upon this foundation, an automated method called intCV, based on the XGBoost model, is introduced to effectively infer correlated variables within interrupt-driven programs. Once we accurately identify the correlated variables requiring atomic execution, existing detection techniques can be utilized to identify violations of multi-variable atomicity. Experimental results on real-world aerospace embedded software demonstrate the practicality and effectiveness of our method.

Keywords–interrupt-driven programs; concurrency bugs; correlated variables; atomicity violations

1. Introduction

Interrupt-driven programs are widely utilized in safety-critical areas such as aerospace, autonomous driving, and medical equipment. In these programs, the main task continuously responds to interrupt service routines (ISRs) and executes interrupt functions to perform corresponding operations in real-time [1]. As the triggering of interrupts is non-deterministic and unpredictable, it can be challenging for programmers to comprehend and develop interrupt-driven programs. If the accesses of shared data are not properly synchronized or protected, uncertain interleaving execution of interrupts may cause concurrency bugs, resulting in severe safety issues.

According to [2], multi-variable atomicity is a typical category of interrupt-driven concurrency bugs, accounting for approximately 36% of the cases. The fundamental reason for this type of bugs is that unsynchronized concurrent access violates the semantic relationships between variables, making it the most challenging category of concurrency bugs to detect [3]. In interrupt-driven programs, developers often use multiple global variables to represent a single entity or a variable as a custom synchronization flag to constrain another variable. These variables are inherently correlated and need to be accessed together in a consistent manner. Therefore, identifying variables with semantic correlations is the most significant challenge in detecting multi-variable atomicity violations.

Currently, a variety of techniques and tools exist for detecting concurrency bugs in interrupt-driven programs. These methods mainly focus on bugs caused by a single variable, such as data races [4][5] and single-variable atomicity violations [6][7], but they are unable to handle multi-variables concurrency bug problem. There are a few bug detection methods for multi-variable atomicity violations in threaded programs. However, on the one hand, these methods only rely on a specific variable correlation characteristics [8] or predefined access interleaving patterns [9] to identify correlated variables, resulting in low accuracy and difficulty in applying them to real-world programs. On the other hand, due to differences in concurrency mechanisms and programming practices, these methods are challenging to apply to interrupt-driven programs.

In this paper, we introduce intCV to identify correlated variables in interrupt-driven programs. We begin by summarizing the characteristic of correlated variables involved in interrupt-driven programs through a comprehensive empirical study. Subsequently, we extract the correlation features of variables through static analysis from real-world aerospace embedded software and construct training samples. Furthermore, with a deep understanding and analysis of the software and collaborative communication with developers, we label the genuine correlated variable pairs and employ the SMOTE enhancement algorithm [10] to balance the distribution of positive and negative samples. Ultimately, we train a classifier using the XGBoost model to facilitate the inference of correlated variable pairs.

To evaluate the proposed approach, we construct our dataset based on the real-world aerospace embedded software from the China Academy of Space Technology (CAST). The dataset contains 23791 samples. We then conduct extensive experiments ,and the results on two real-world aerospace embedded software programs demonstrate that intCV outperforms existing methods. Once we can accurately identify the correlated variables that need to maintain atomic execution, existing detection technologies can be employed to detect multi-variable atomicity violations.

In summary, this paper makes the following contributions:

- We study the characteristics of variables correlation within interrupt-driven programs, which can serve as a robust foundation for inferring correlated variables effectively.
- We introduced an automated technique for inferring correlated variables, merging the capabilities of static analysis with the XGBoost model.
- We implement the prototype tool intCV and evaluated it on real-world embedded software to demonstrate the effectiveness of the proposed technique. intCV is publicly available at https://github.com//AceBce/intCV.

The rest of this paper is organized as follows. Background and motivating examples are presented in Section 2. Section 3 describes the proposed approach in detail. The experiment evaluation is given in Section 4. Section 5 reviews the main approaches related to this work. Finally, we conclude the paper and outline our future work prospects.

2. BACKGROUND AND MOTIVATION

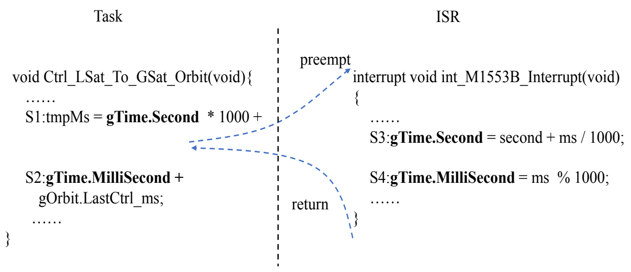
2.1. Interrupt-Driven Program

Interrupt-driven programs represent a typical category of concurrent programs, which mainly rely on the interrupt mechanism of the embedded processor to achieve real-time concurrent response. An interrupt-driven program consists of one main task with an infinite loop structure and several interrupt service routines (ISRs), where the main task cyclically waits for the triggering of ISRs to carry on the corresponding operation in real-time. Distinct from threaded or event-driven programs, interrupt-driven programs feature an asymmetric preemption relationship. Every ISR maintains a specific priority, allowing only those of higher priority to preempt their lower-priority counterparts and not vice versa. Furthermore, bare-metal programming is quite common in the development of interrupt-driven embedded software [11]. Given that embedded software typically has limited memory, developers extensively use global variables as shared data, facilitating communication between tasks and interrupts while avoiding parameter and calling stack usage. Since having few available native concurrency primitives, a large number of flag variables are used as customized synchronization operations.

2.2. Multi-variable Atomicity Violation

In interrupt-driven programs, atomicity violations are the most encountered interrupt concurrency bugs [13]. Previous concurrency bug detection tools mainly focused on concurrency bugs that involve only one variable, that is, only the atomicity of instructions accessing the same variable. However, concurrency bugs caused by unsynchronized access to multiple variables are widespread in interrupt-driven programs. They actually cause a significant percentage (about 36%) of concurrency bugs [2]. We start with two real-world examples to understand why there exist so many multi-variable atomicity violations in interrupt-driven programs.

Figure 1 shows an example from a satellite software, *gTime.Second* and *gTime.MilliSecond* represents the second and millisecond of a certain moment that should be read/written



```
void Ctrl_LSat_To_GSat_Orbit(void){
  ......
  S1:tmpMs = gTime.Second  * 1000 +

  S2:gTime.MilliSecond +
     gOrbit.LastCtrl_ms;
  ......
}
```

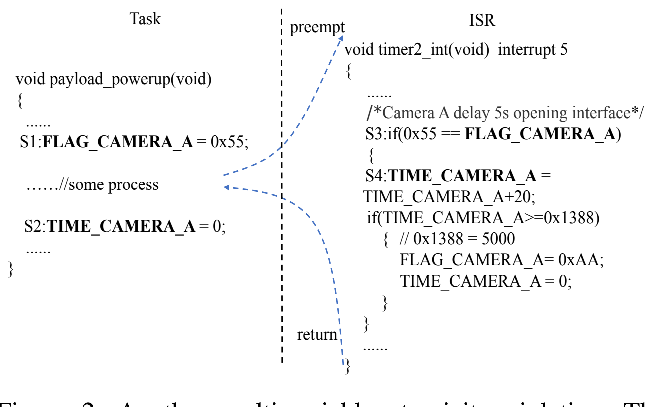
```
interrupt void int_M1553B_Interrupt(void)
{
   ......
   S3:gTime.Second = second + ms / 1000;

   S4:gTime.MilliSecond = ms  % 1000;
   ......
}
```

Figure 1: A multi-variable atomicity violation. *gTime.Second* and *gTime.MilliSecond* represent minutes and seconds of a specific time and should be read together. However, due to interrupt preemption occurring between S1 and S2, *tmpMs* read a wrong time.

together. However, the ISR may preempt between S1 and S2 and update the time. As a result, the value of *gTime.Second* read by *tmpMs* is an old one while the value of *gTime.MilliSecond* is a new one. Such inconsistency can lead to timing errors and subsequent incorrect program behavior.

Figure 2 shows another case from a control software, variable *FLAG_CAMERA_A* and variable *TIME_CAMERA_A* are correlated since they are used in combination to control the camera's switch. In payload_powerup() function, *FLAG_CAMERA_A* is first set to 0X55 to signal the camera's readiness for power-on while simultaneously clearing *TIME_CAMERA_A* to initiate the timer. In the ISR, when *FLAG_CAMERA_A* is found to be 0X55, *TIME_CAMERA_A* is incremented until it reaches a 5-second timer, triggering the camera power-up. However, when interrupt preemption occurs between S1 and S2, the camera startup timing error occurs.



```
void payload_powerup(void)
{
  ......
 S1:FLAG_CAMERA_A = 0x55;

  ......//some process

  S2:TIME_CAMERA_A = 0;
  ......
}
```

```
void timer2_int(void)  interrupt 5
{
   ......
   /*Camera A delay 5s opening interface*/
   S3:if(0x55 == FLAG_CAMERA_A)
   {
   S4:TIME_CAMERA_A =
   TIME_CAMERA_A+20;
    if(TIME_CAMERA_A>=0x1388)
      {  // 0x1388 = 5000
         FLAG_CAMERA_A= 0xAA;
         TIME_CAMERA_A = 0;
      }
   }
   ......
}
```

Figure 2: Another multi-variable atomicity violation. The ISR interleaves Task's update to *FLAG_CAMERA_A* and *TIME_CAMERA_A* and reads inconsistent values. As a result, The satellite camera was turned on at an incorrect time.

As we can see, the above bug is neither a race bug nor a single-variable atomicity violation bug. Even if the accesses to a single variable are well synchronized, the bug still exists. The root cause of this type of bug is that unsynchronized concurrent accesses violate the semantic relationship between variables. In real-world interrupt-driven programs, developers often use multiple variables, consciously or unconsciously, to simulate correlations inherent in the real world. Such semantically correlated variables must be either updated consistently or accessed together to give the program a consistent view instead of a partial one. Nevertheless, the execution may violate the access correlation due to the preemption of ISR.

2.3. XGBoost

XGBoost, proposed by Chen and Guestrn in 2016 [14], is a machine learning model that achieves stronger learning effects by integrating multiple weak learners. Since its introduction, it has shown excellent performance in various classification tasks and has been widely recognized. Because of XGBoost's powerful learning and classification capabilities, it is able to efficiently capture patterns and correlations in data [15]. Therefore, the features of variable correlation in interrupt-driven programs can be effectively learned. Furthermore, XGBoost possesses the unique ability to autonomously assess feature importance, facilitating the identification of critical features in inferring correlation variables.

3. Approach

In this section, we present intCV, an approach built upon an XGBoost model that automatically learns the semantic information from programs and infers whether a shard variable pair is correlated. The overall procedure of intCV is depicted in Figure 3. Firstly, in order to identify the features required for model training, we summarize the features of correlated variables involved in real-world interrupt-driven programs through a comprehensive empirical study. Then, we employed static program analysis to extract the features related to the correlation between pairs of shared variables, thereby constructing training and testing samples. To obtain labeled samples, we conducted an in-depth understanding and analysis of software documentation and source code, and engaged in communication with developers. This process allowed us to identify genuine correlated variables within the software. Finally, We constructed an XGBoost model and trained it using the training set. The trained model serves as a classifier, which can automatically learn the semantic and structural features to distinguish whether shared variable pairs are correlated.

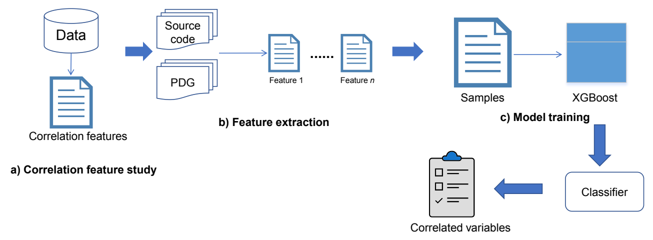


Figure 3: Overview of our approach.

3.1. Correlation Feature Study

The existing methods use relatively simple correlation features and cannot effectively infer correlated variables. To explore how the correlation between variables is reflected in the code, we first conducted an empirical study on real-world cases of multi-variable atomicity violations, revealing significant features that are more detailed than before. The following shows the typical variables correlations in interrupt-driven programs:

- **Data dependence**: Data dependence reflects the logical relationship between variables, since we generated one value from the other. As a result, correlated variables tend to exhibit explicit or implicit data dependencies. For example, Table I(d) shows that *URcheck* is explicit data dependent on *URlen* in the ISR and they need update together in the TASK, Figure 1 shows an implicit data dependencies between *gTime.Second* and *gTime.MilliSecond* since the variable *tmpMs* is data dependent on both variables.
- **Control dependence**: Similar to data dependence, a control dependence also reflects the logical relationship between variables inside a control conditional condition and the branch statement. In interrupt-driven programs, a large number of flag variables are used as customized synchronization operations as lack of native concurrency primitives. These flag variables are correlated with variables that need to be synchronized. As shown in Table I(c), *FlgRequest.RecEndTemp* is control dependent on *FlgRequest.RecEndTemp*.
- **Elements sibling**: Elements of an array are usually used to represent different parts of the same entity. As shown in Table I(b), *Camera_buf[i]* represents a set of buffer data that always needs to be accessed together.
- **Fields sibling**: Similarly, members of the same structure are often used to represent different aspects of the same object. Table I(a) shows that the field *abstime.abshi* and *abstime.abspps* are used together to represent a specific time; therefore, accesses to them are always together. Similar examples can be seen in Figure 1.
- **Short distance**: Because of the way programmers think, they tend to use correlated variables at the same time, so these variables tend to be relatively close to each other in source code distance. As shown in the Table I, all the correlated variables are very close together on the source code.
- **Similar name**: Correlated variables represent the same information in different ways or specify different aspects of the data, often with similar names, such as the same prefix/suffix. As shown in Table I(d), the correlated variable *URlen* and *URcheck* are named with the prefix UR.

3.2. Feature Extraction

Based on the empirical study, we selected multiple variable correlation features from the collected software to build the dataset samples. Since the correlations between multiple variables can be reflected through their pairwise interactions, we adopt a pairwise analysis strategy to reduce the complexity and overhead of the analysis. For this purpose, we construct the samples as follows:

TABLE I: Correlated variable examples

ID	TASK	ISR

a	abstime.abshi = abshi;
	abstime.abspps = abspps
		final_pps = abstime.abspps
		+ (abstime.abshi <<32)

b	for(i = 2;i<16;i++){
	sum = sum+Camera_buf[i];
	}
		for(i = 0;i<17;i++){
		Camera_buf[i] = XBYTE[FIFO];
		}

c	if(FlgRequest.RecEndTemp == 0){
	FlgRequest.EndTemp = VALID;
	}
		FlgRequest.RecEndTemp = VALID;
		FlgRequest.EndTemp = 0;

d	URlen=0;
	URcheck=0;
		URcheck=(URlenˆDIYZS);

$$sample = \langle Pair, L, R, CO, CD, DD, Arr, Fields, Name \rangle$$

where *Pair* represent variable pair *(x,y)*, *L*, *R*, and *CO* represent the occurrences of *x*, *y*, and their co-occurrences in the program, respectively. The co-occurrences refer to the instances where these two variables appear together in the same function with a source code distance of less than 10 lines. *CD*, *DD*, *Arr*, *Fields* and *Name* denote whether there exists a data dependency, control dependency, if they belong to the same array, if they belong to the same structure and if they have similar names, respectively, taking values of 0 or 1.

In order to collect the information needed for the sample, intCV first traverse through the source code, collect the variable names, access locations and access numbers of all global variables. Then, intCV forms pairs of all global variables and computes their frequency of co-occurrences in the program, examines whether they belong to the same array or structure, and evaluates if they have similar names. Subsequently, intCV constructs the program dependency graph. By traversing the program dependency graph, it captures the control and data dependency relationships between variable pairs.

3.3. Model Training

The quantity and diversity of the training data used for training machine learning models play a crucial role in the accuracy of model decisions. However, there are no available datasets to infer correlated variables. what's more, whether variables are correlated is closely related to program semantics, it is challenging to clarify the correlated variables even when the program is given. To address this problem, we used real-world aerospace embedded software from the CAST as our data source. Initially, we filtered software that came with comprehensive documentation and annotated source code. Following this, we delved deep into the software logic and semantics, preliminarily identifying variables that might be correlated. Ultimately, we liaised with the development engineers associated with each piece of software to discuss and verify these correlations. This process yielded four software complete with annotations for correlated variables. On average, the verification and confirmation process for each software took about a month.

As shown in the Table II, this dataset consists of one power supply control software, one propulsion control box software, one attitude control software and one power control software , with sizes ranging from 970 lines to 2275 lines. The term *Samples* represents the constructed pairs of correlated variables, as illustrated in Table III. The Table presents the feature and label information of the samples. For example, sample 1 signifies that variable *gTime.Second* appears 5 times in the program, variable *gTime.MilliSecond* appears 5 times, and they co-occur 5 times. There is a data dependency between *gTime.Second* and *gTime.MilliSecond*, and they belong to the same structure. A label of 1 indicates that this variable pair is correlated. By feeding the constructed sample data into XGBoost, we obtain our final classifier.

TABLE II: Dataset

Software	Description	LOC	Samples
module1	A power supply control software	2275	11935
module2	A propulsion control box software	970	2346
module3	An attitude control software	1153	4950
module4	A power control software	1218	4560

4. Experiment and evaluation

We have implemented intCV, a detector for correlated variables. It builds on some open-source tools including Clang/LLVM [16] for implementing the C front-end, PhASAR [17] for implementing the data flow analysis, dg [18] for building dependence graph. We carried out our experiments on a computer with an M1 Pro CPU, 16 GB of RAM, and the MacOS 13.3 operating system.

4.1. Dataset

Our data set is shown in Table II. We selected module1 and module2 as the training set and module3 and module4 as the test set. Due to the imbalance of positive and negative samples in the dataset, the SMOTE algorithm was used to expand the training sample. It is worth emphasizing that we aimed to include as many cases as possible in our dataset. However, manually analyzing the correlated variables in real-world embedded software is highly time and resource-consuming. This process requires a thorough understanding and analysis of the program. Additionally, all the correlated variables that we manually identify need to be individually confirmed by the developer.

4.2. Evaluation Metrics

To evaluate the effectiveness of intCV, we define the following metrics:

- Precision: The ratio of the number of correlated variables inferred correctly to the total number of correlated variables inferred.

$$Precision = \frac{TP}{TP + FP}$$

- Recall: The ratio of a number of truly inferred correlated variables to the total number of all correlated variables.

$$Recall = \frac{TP}{TP + FN}$$

TABLE III: Samples information

ID	Pair	L	R	CO	CD	DD	Arr	Fields	Name	Label
1	(gTime.Second,gTime.MilliSecond)	5	5	5	0	1	0	1	0	1
2	(open_count_2,receeive_buffer_A)	9	8	0	0	0	0	0	0	0
3	(wk_rData[72],wk_rData[79])	2	2	0	0	0	1	0	0	0
4	(power_state,flag_sf)	5	13	1	0	0	0	0	0	0

TABLE IV: Referring results of intCV

Software	Precision	Recall	Accuracy	F1-score
module3	50.00%	35.71%	99.72%	41.67%
module4	57.14%	36.36%	99.72%	44.44%
average	53.57%	36.04%	99.72%	43.06%

- Accuracy: The ratio of correctly inferred correlated variables and non-correlated variables to the total number of all samples.

$$Accuracy = \frac{TN + TP}{TP + TN + FP + FN}$$

- F1-score: A metric used to evaluate the overall performance of a model, considering both precision and recall trade-offs. Its value ranges from 0 to 1, where 1 indicates the best performance, and 0 indicates the worst.

$$F_1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

4.3. Results

We investigate the following research questions to provide a thorough analysis of the experimental results.

RQ1: **How effective is intCV in inferring correlated variables for interrupt-driven programs?**

In this research question, we aim to measure the effectiveness of our approach based on the metrics defined in section 4.2. Table IV presents the experimental results on our dataset. Firstly, we note that the average precision, recall, and F1-score on the two software are not outstanding, being 53.57%, 36.04%, and 43.06% respectively. However, given the intricacies inherent in concurrent programs, even in the simple task of prediction concurrent condition-related bugs, the state-of-the-art learning methods [19] achieve only a precision of 55% and a recall rate of 39.6%. Considering that we are dealing with the analysis of more intricate program semantics, this performance is acceptable. Furthermore, due to the limited size of our training dataset, consisting of merely two software, the model's effectiveness is hindered. However, with access to a more extensive training dataset, the model's performance could be notably improved. Additionally, our precision is notably high at 99.72%, mainly due to the relatively limited occurrences of correlated variables in the program, resulting in a significant disparity between positive and negative samples.

RQ2: **How does intCV perform compared with other methods for correlated variables inferring?**

We experimentally compared intCV with MUVI for correlated variable inference, and the experimental comparison results are shown in Table V, where #CVs is the number of the correlated variable pairs in software.

TABLE V: The performances of different approaches for inferring correlated variables

Software	CVs	MUVI			intCV		
		TP	FP	FN	TP	FP	FN
module3	· 14	4	6	10	5	5	9
module4	11	3	4	8	4	3	7
overall	25	7	10	18	9	8	16

Since MUVI does not directly identify correlated variables, but instead provides the probability of correlation between variables and presents results in descending order of probability values. To ensure fairness, we extract the top n results from MUVI, where n corresponds to the number of reports from intCV. For example, intCV reports 10 pairs of correlated variables in module1's detection, so we select the top 10 results from MUVI for comparison. As shown in Table V, out of the total 25 pairs of correlated variable pairs in the two software programs, intCV reported 15 pairs, with 9 being true positives and 8 being false positives. In comparison, MUVI reported 7 true positives and 10 false positives. Whether considering individual programs or the overall performance, intCV outperformed MUVI.

5. Related Work

There are several techniques and tools aimed at addressing concurrency bugs in interrupt-driven programming. Wang et al. [12] propose SDRacer, which automatically detects order violations by combining static analysis and symbolic execution techniques. Wu et al. [4] introduce a framework of bounded model checking for analyzing data race. Their key idea is to automatically serialize a concurrent interrupt-driven program as a non-deterministic sequential program. Li et al. [13] present a precise and efficient static detection technique for interrupt atomicity violations, described by access interleaving pattern. However, these methods only focus on single-variable concurrency bugs and cannot handle the multi-variable concurrency bug problem well. intCV is aimed at inferring correlated variables in interrupt-driven programs to help previous concurrency bug detectors to detect multi-variable atomicity violations.

There are also some works focusing on detecting multi-variable atomicity violations in threaded programs. Most of these methods require manual labeling of atomic regions [20][21][22][23] or employ predefined patterns of defect-prone access interleaving for detection [24][25]. However, manual annotation of atomic regions not only requires a deep understanding of the program but also consumes a significant amount of resources and time, making it difficult to apply in practical engineering scenarios. Predefined access interleaving patterns cannot reflect the correlation between variables, leading to numerous false positives and false negatives in the detection results.

Only a few methods explore how to automatically identify correlated variables. Lu et al. [3] proposed MUVI, a hybrid race detector for correlated variables. The algorithm recognizes correlations among variables by combining static program analysis and data mining techniques. It assumes variables that frequently appear in the same method and are relatively close to each other in the source code distance as correlated variables. Inspired by MUVI, Jannesari et al. [8] proposed a method to identify correlations between variables by taking data dependencies into consideration. In addition to identifying the variables that are often accessed near each other, it also recognizes correlations based on the predefined patterns, which indicate strong data and control dependencies between the variables. Sun et al. [26] developed a dynamic analysis method to infer correlation between variables based on their distance in execution trace. However, these methods rely only on specific correlation features to infer correlated variables, resulting in relatively low accuracy of results. Additionally, these methods do not directly provide the correlated variables but offer a probability value for the programmer to confirm. However, the confirmation process is highly complex and time-consuming. The approach presented in this paper improves upon this situation by characterizing multiple correlation features and leveraging machine learning techniques.

6. Conclusion

In this paper, we first conducted an in-depth study on a large number of real-world cases of multi-variable atomicity violations, summarizing the correlation features of variables. Based on this foundation, a correlation variable inference method based on XGBoost is proposed. This method extracts the features of correlated variables from real-world software using static analysis techniques, employs the SMOTE enhancement algorithm to balance the sample distribution, and ultimately constructs and trains an XGBoost model for the detection of correlated variables. The experimental results on two real-world aerospace embedded software programs demonstrate that intCV outperforms existing methods with a minimal number of training samples, validating the effectiveness of the proposed approach in this study.

In future work, we plan to increase the volume of our training data to enhance the model's performance and reduce false positives and false negatives. Additionally, we will explore more features to aid in the identification of correlated variables. We are also committed to integrating our model into practical software development tools, aiding programmers in bug detection and prevention in real-world scenarios.

References

[1] Michael F Siok and Jeff Tian. 2007. Empirical study of embedded software quality and productivity. In 10th IEEE High Assurance Systems Engineering Symposium (HASE'07). IEEE, 313–320.

[2] Li C, Chen R, Wang B, et al. An Empirical Study on Concurrency Bugs in Interrupt-Driven Embedded Software[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023: 1345-1356.

[3] Lu S, Park S, Hu C, et al. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs[C]//Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. 2007: 103-116.

[4] Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, and Ji Wang. 2013. Data race detection for interrupt-driven programs via bounded model checking. In 2013 IEEE Seventh International Conference on Software Security and Reliability Companion. IEEE, 204–210.

[5] Nikita Chopra, Rekha Pai, and Deepak D'Souza. 2019. Data races and static analysis for interrupt-driven kernels. In European Symposium on Programming. Springer, 697–723.

[6] Rui Chen, Mengfei Yang, and Xiangying Guo. 2016. Interrupt data race detection based on shared variable access order pattern. Ruan Jian Xue Bao/Journal of Software 3 (2016), 547–561.

[7] Haining Feng, Liangze Yin, Wenfeng Lin, Xudong Zhao, and Wei Dong. 2020. Rchecker: A CBMC-based Data Race Detector for Interrupt-driven Programs. In 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 465–471.

[8] Jannesari A, Wolf F. Automatic generation of unit tests for correlated variables in parallel programs[J]. International Journal of Parallel Programming, 2016, 44: 644-662.

[9] S. Park, R. Vuduc, and M. J. Harrold, "Unicorn: a unified approach for localizing non-deadlock concurrency bugs," Software Testing, Verification and Reliability, vol. 25, no. 3, pp. 167–190, 2015.

[10] Chawla N V, Bowyer K W, Hall L O, et al. SMOTE: synthetic minority over-sampling technique[J]. Journal of artificial intelligence research, 2002, 16: 321-357.

[11] Wang B, Chen R, Li C, et al. SpecChecker-ISA: a data sharing analyzer for interrupt-driven embedded software[C]//Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 2022: 801-804.

[12] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2017. Automatic detection and validation of race conditions in interrupt-driven embedded software. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 113–124.

[13] Chao Li, Rui Chen, Boxiang Wang, Tingting Yu, Dongdong Gao, and Mengfei Yang. 2022. Precise and efficient

atomicity violation detection for interrupt-driven programs via staged path pruning. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. 506–518.

[14] Chen T, Guestrin C. Xgboost: A scalable tree boosting system[C]//Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016: 785-794.

[15] Zhang W, Wu C, Zhong H, et al. Prediction of undrained shear strength using extreme gradient boosting and random forest based on Bayesian optimization[J]. Geoscience Frontiers, 2021, 12(1): 469-477.

[16] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In The BSD conference, Vol. 5.

[17] Schubert P D, Hermann B, Bodden E. Phasar: An inter-procedural static analysis framework for c/c++[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer International Publishing, 2019: 393-410.

[18] 2021. DG website. https://github.com/mchalupa/dg.

[19] Zhang J, Wang X, Zhang H, et al. Detecting Condition-Related Bugs with Control Flow Graph Neural Network[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023: 1370-1382.

[20] Flanagan C, Qadeer S. A type and effect system for atomicity[J]. ACM SIGPLAN Notices, 2003, 38(5): 338-349.

[21] Vaziri M, Tip F, Dolby J. Associating synchronization constraints with data in an object-oriented language[J]. ACM Sigplan Notices, 2006, 41(1): 334-345.

[22] Wang L, Stoller S D. Runtime analysis of atomicity for multithreaded programs[J]. IEEE Transactions on Software Engineering, 2006, 32(2): 93-110.

[23] Jannesari A, Westphal-Furuya M, Tichy W F. Dynamic data race detection for correlated variables[C]//Algorithms and Architectures for Parallel Processing: 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I 11. Springer Berlin Heidelberg, 2011: 14-26.

[24] Hammer C, Dolby J, Vaziri M, et al. Dynamic detection of atomic-set-serializability violations[C]//Proceedings of the 30th international conference on Software engineering. 2008: 231-240.

[25] Park S, Vuduc R, Harrold M J. A unified approach for localizing non-deadlock concurrency bugs[C]//2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 2012: 51-60.

[26] Sun Z, Zeng R, He X. A method for predicting two-variable atomicity violations[C]//2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2018: 103-110.