Technical Note

UE4 Redundant Asset Detection Method Based on Pointer Analysis

Tingzhen Liu *

Posted Date: 18 September 2024

doi: 10.20944/preprints202409.1236.v1

Keywords: program analysis; static analysis; pointer analysis; data flow analysis; flow sensitivity; game develop

Technical Note

UE4 Redundant Asset Detection Method Based on Pointer Analysis

Tingzhen Liu

Tencent IEG, CROS; firstsg@outlook.com

Abstract: In the process of game development, identifying and deleting currently abandoned assets as content iterates can effectively reduce the size of game packages. But the built-in reference checking tool in UE4 can only check static references of assets and cannot identify dynamically referenced resources in the program. We have developed a static analysis tool for analyzing unused assets in the UE4 project to address this issue. This tool checks the function parameters of all loading points, analyzes the value range of the parameter string based on the data dependency relationship of the actual parameter variable, and considers that the assets that match within the range are referenced. Due to the fact that in some cases, the exact data stream of the actual parameter variable is not computable (resulting in false negatives), the reverse analysis tool supports manually marking the parameter range at the loading point. This tool can generate asset collections that are dynamically referenced. The union of its results with the built-in reference checking tool in UE4 is the set of all referenced assets. The difference set between all asset sets and the referenced asset set is the abandoned asset set. This achieves a more complete cleaning of redundant resources and reduces the size of game packages.

Keywords: program analysis; static analysis; pointer analysis; data flow analysis; flow sensitivity; game develop

Method

Our static analyzer checks the function parameters of all WBP loading points. Analyze the value range of the variable based on the data dependency relationship of the actual parameter variable [1]. It approximates as closely as possible all the possible literal values that the variable at the loading point may receive. To achieve this goal, we constructed a two-stage approach: the first stage collects all the variable sets that the variable depends on (if the variable is assigned values by other variables), and the second stage collects the literal assignments of all variables in the variable set.

Step 1: The symbols tracked by algorithms are divided into three categories: ordinary variables, array types, and record types. The algorithm starts from the analysis of using symbol s and constructs a data flow graph for all three types of symbols that symbol s depends on.

Suppose there is a symbol a in the data flow graph G. The rule for adding elements to G for array types is:

$$\frac{\Gamma \vdash a \quad \Gamma \vdash b: Array \quad \Gamma \vdash (a := b[M])}{add \ \ a \to b \ \ to \ G} \ ArrayEdgeAddA$$

$$\frac{\Gamma \vdash a: Array \quad \Gamma \vdash b \ \ \Gamma \vdash (a.insert(b))}{add \ \ a \to b \ \ to \ G} \ ArrayEdgeAddB$$

The rule for adding elements to G for record types is:

$$\frac{\Gamma \vdash a \quad \Gamma \vdash b: Record \quad \Gamma \vdash (a := b.c)}{add \ \ a \to b.c \ \ to \ G} \ RecordEdgeAdd$$

Both b and $b.c$ are considered elements in G.

For any symbol, there are rules:

$$\frac{\Gamma \vdash a \ \ \Gamma \vdash b \ \ \Gamma \vdash (a := b)}{add \ \ a \to b \ \ to \ G} \ BasicEdgeAdd$$

 doi:10.20944/preprints202409.1236.v

The algorithm first adds the symbol s to G, and then analyzes the code upwards, applying three types of rules until G can no longer be expanded.

For function calls, there are rules:

$$\frac{\Gamma \vdash a \quad \Gamma \vdash F(a, \dots\dots) \quad F(A) \vdash b \quad F(a, \dots\dots) \vdash (a := b)}{Do\ EdgeAdd\ (F(A)\ as\ \Gamma,\ G'\ as\ G)\quad add\ a \xrightarrow[F(A_{Argument})\ by\ CallA]{} A_{Argument}/a\ to\ G}\ CallEdgeAddA$$

Note: $F(a, \dots\dots)$ *is equivalent to* $F(A_{Argument})$ *that represents the actual call point. Among them,* $a \in A_{Argument}$

This rule states that if symbol a is modified in function F, and the modification depends on symbol b in the function. Then construct a separate data flow graph G' for F. Then add it to the original data flow graph G that the edge from symbol a to the other arguments passed to F. This edge contains call point information, that is, the corresponding relationship between these arguments and formal parameters of F.

There are rules for modifying symbol a by return value:

$$\frac{\Gamma \vdash a \quad \Gamma \vdash (a := F(A_{Argument}))}{Do\ EdgeAdd\ (F(A)\ as\ \Gamma,\ G'\ as\ G)\quad add\ a \xrightarrow[F(A_{Argument})\ by\ CallB]{} A\ to\ G}\ CallEdgeAddB$$

This rule adds it to the original data flow graph G that the edge from symbol a to all arguments passed to F (assuming $a \notin A_{Argument}$).

Afterwards, the algorithm replaces the $CallA$ and $CallB$ edges in G with the data flow graph G' of F:

$$\frac{\forall e: Edge\ by\ CallA\ in\ G}{Do\ Replace\ e\ to\ G'\ (A_{Argument}\ as\ A)}\ CallAEdgeReplace$$

This rule replaces A (several formal parameter nodes) in G' with several arguments nodes (i.e. $A_{Argument}$ in $a \xrightarrow[F(A_{Argument})\ by\ CallA]{} A_{Argument}/a$) recorded in edge e. This allows it to connect with the existing nodes in G.

$$\frac{\forall e: Edge\ by\ CallB\ in\ G \quad F(A) \vdash (return\ r)}{Do\ Replace\ e\ to\ G'\ (A_{Argument}\ as\ A,\ a\ as\ r)}\ CallBEdgeReplace$$

In this rule, since the symbol a is modified by the return value of F, it is also necessary to replace the symbol node returned in G' with the symbol a. This allows it to connect with the existing nodes in G.

Step 2: The algorithm collects all the statements in G that have been assigned literal values:

$$\frac{G \vdash root \quad G \vdash a \quad G \vdash (root \rightharpoonup a) \quad \Gamma \vdash (a := M)}{add\ M\ to\ R}$$

Among them, $root$ is the starting point for analysis, $root \rightharpoonup a$ represents that $root$ is connected to a. R is the set of possible literal values that the analyzed symbol a may receive.

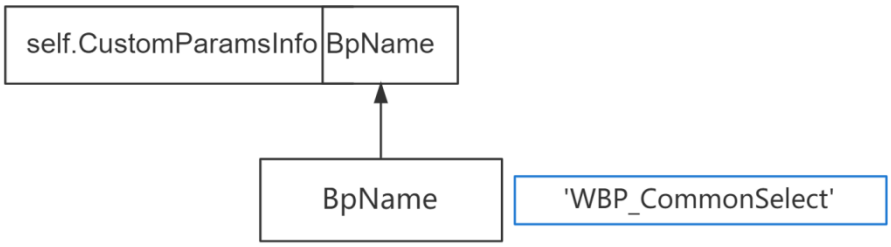
Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 September 2024 doi:10.20944/preprints202409.1236.v

Example

As a code chip:

```
local BpName = self.CustomParamsInfo.BpName
if self.bIsOpt then
      BpName = 'WBP_CommonSelect'
end
self:LoadComponent(BpName, self.NSlotOpt, ParamsInfo)
```

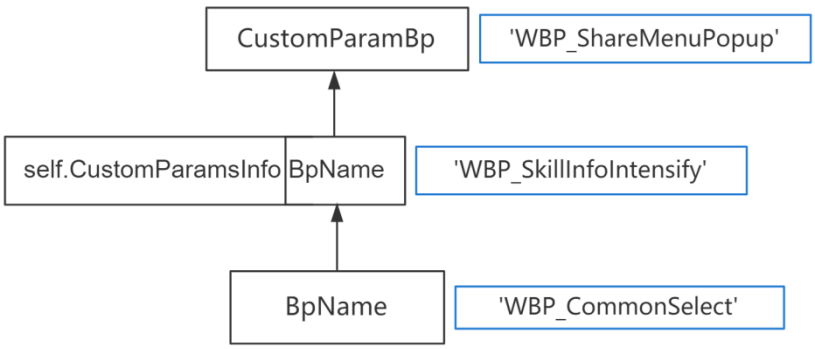
The argument *BpName* of the loading point *LoadComponent* is the symbol to be analyzed. Looking back, it can be found that *BpName* has two modification points, both of which are assignment statements. Therefore, the data flow graph G becomes:



After scanning all the modified points of *BpName*, scan the newly added symbols in the data flow graph. If there is code:

```
self.CustomParamsInfo.BpName = 'WBP_SkillInfoIntensify'
if bInitial then
      CustomParamBp = 'WBP_ShareMenuPopup'
      self.CustomParamsInfo.BpName = CustomParamBp
end
```

Then the data flow graph G becomes:



If all symbols have been scanned and no new symbols have been added, then the range of literal values for all symbols will be merged. The possible values for *BpName* will be *WBP_CommonSelect*, *WBP_ShareMenuPopup*, *WBP_SkillInfoIntensify*.

From this example, it can be seen that due to the trade-off between the speed and accuracy of the analysis algorithm, we must choose appropriate control flow and data flow abstractions to avoid difficult to handle calculations. To achieve this goal, our algorithm over-approximates the exact set of values that its parameters may have. Therefore, only flow sensitive analysis methods [2] are used.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 September 2024 doi:10.20944/preprints202409.1236.v

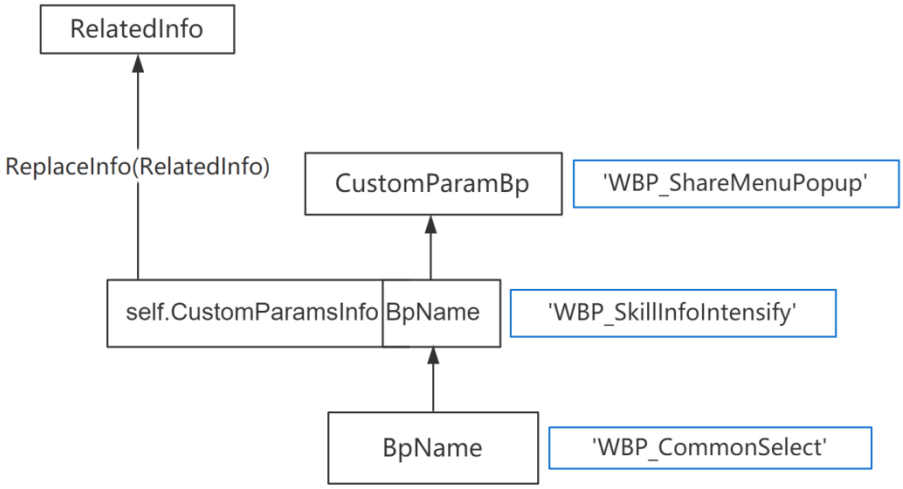
In this example, we do not calculate whether *self.bIsOpt* cannot be *true* (if it is always *false*, we can exclude one value), but instead adopt all possible paths and contexts that can expand the set of values.

For the analysis of function calls, such as code chip:

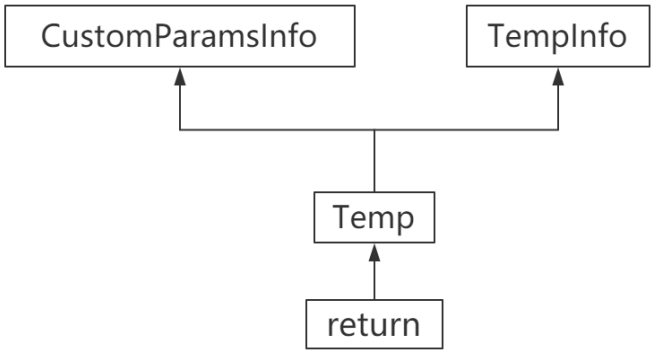
```
function ReplaceInfo(CustomParamsInfo, TempInfo)
	local Temp
	if CustomParamsInfo.bIsOpt then
		Temp = TempInfo
	else
		Temp = CustomParamsInfo
	end
	return Temp
end

self.CustomParamsInfo = ReplaceInfo(self.CustomParamsInfo,RelatedInfo)
```

The function *replaceInfo* indirectly modifies *self.CustomParamsInfo.BpName* by modifying *self.CustomParamsInfo*. So according to the rule CallEdgeAddB, update the data flow graph to:

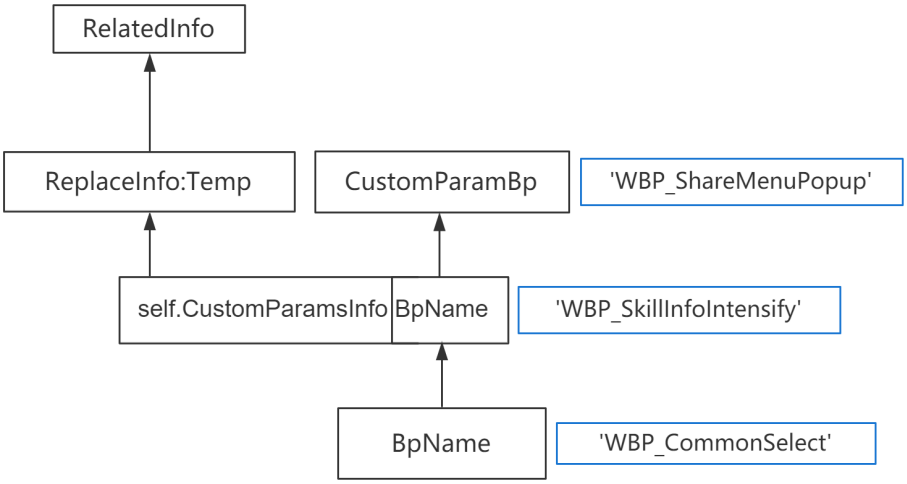


Then build a separate data flow graph G' corresponding to the *replaceInfo*:



Apply the rule CallBEdgeReplace to replace the newly added nodes in G with G':

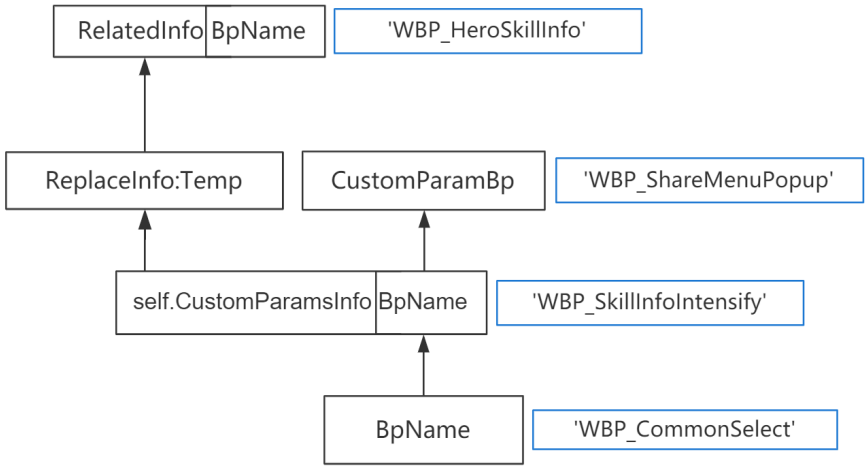
Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 18 September 2024 doi:10.20944/preprints202409.1236.v



Then the algorithm will collect the modification points of the newly added symbol *ReplaceInfo* (Step 1) and literal assignment to *ReplaceInfo.BpName* (Step 2). If there is code:

```
RelatedInfo.BpName = 'WBP_HeroSkillInfo'
```

Then the data flow graph G becomes:



For code involving string calculations such as substitution and concatenation. At present, some string static analysis algorithms can analyze the patterns that generate string symbols (such as automata [3,4,7,8] or regular expressions [5,6]). If any asset can match the pattern of the symbol, then it is considered to be in use.

Conclusion

Our method uses asset loading points as the root to construct a data flow graph in reverse. This method can determine which symbols to abstract. Due to the fact that only a small portion of the code is related to loading points, a large number of paths that do not interact with valid symbols will not be detected, effectively reducing the cost of analysis. In addition, in some cases, due to the complexity of real-world programs, the precise data flow of actual parameter variables is incalculable (resulting in false negatives), so our reverse analysis tool supports manually marking parameter ranges at the loading point.

 doi:10.20944/preprints202409.1236.v

Reference

1. Shun N A . Analysis Technology Reseach of Data Flow Oriented Java Program Pointer[J]. Computer Programming Skills & Maintenance, 2014.
2. Thiessen R . University of Alberta Expression Data Flow Graph: Precise Flow-Sensitive Pointer Analysis for C Programs. University of Alberta, 2011.
3. Gordon M I , Kim D , Perkins J , et al. Information-Flow Analysis of Android Applications in DroidSafe[C]// Network & Distributed System Security Symposium. 2015.
4. Vincenzo Arceri; Isabella Mastroeni. Static Program Analysis for String Manipulation Languages. Electronic Proceedings in Theoretical Computer Science 2019, 299, 19 -33.
5. Negrini L , Arceri V , Ferrara P , et al. Twinning Automata and Regular Expressions for String Static Analysis[C]// International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, Cham, 2021.
6. Trinh M T , Chu D H , Jaffar J . S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. ACM, 2014.
7. Aske Simon Christensen; Anders Møller; Michael I. Schwartzbach. Precise Analysis of String Expressions. BRICS Report Series 2003, 10, 1 .
8. D. Shannon; I. Ghosh; S. Rajan; S. Khurshid. Efficient symbolic execution of strings for validating web applications. Proceedings of the 2nd International Workshop on Defects in Large Software Systems Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) - DEFECTS '09 2009, 22 -26.