December 2021

Detecting Dead Code Based on Captured Stack Traces

Edward Palmer

Vytautas Vaitukaitis

John Franco

Suki Zhang

Detecting Dead Code Based on Captured Stack Traces

ABSTRACT

Large codebases, e.g., with millions of lines of code, can have sections that are dead - sections of code that are rarely or never executed in practice. Currently, developers cannot feasibly identify dead code, especially for client applications that are server driven. In the absence of evidence that a code section is dead, developers are obliged to migrate it to higher versions of the codebase. Dead code thus accumulates over time and adds a tax on all future changes. This disclosure describes techniques to detect dead code based on the behavior patterns of client software during runtime. With user permission, stack traces are sampled and captured during runtime across a large number of devices running a given client software. Symbolicated call stacks are listed by frequency of execution to determine sections of code that are rarely or never executed. Rare or never-executed sections of code are reported to developers for further analysis to identify and remove dead code.

KEYWORDS

- Dead code
- Stack trace
- Symbolication
- Static code analysis
- Dynamic code analysis
- Dead code stripping

BACKGROUND

Large codebases, e.g., with millions of lines of code, can have sections that are dead - sections of code that are rarely or never executed in practice. Currently, developers cannot

feasibly identify dead code and, in the absence of evidence that a code section is dead, are obliged to migrate it to higher versions of the codebase. Dead code thus accumulates over time and adds a tax on all future changes.

A popular technique for detecting dead code is static analysis, which entails examining a line of code to determine if another line calls it. Traditional static analysis doesn't work for server-driven applications where behavior of the client application is controlled from the server. The client may have code for responding to a server request, but the server may have deleted the corresponding code in the past. A large codebase can have thousands of possible messages, making a manual search infeasible. The server can still send such messages to older, not recent, clients, or to a different client platform entirely.

```
CommandFactory.m

Line 101:  … else if (message.hasOldMessage) { ← this line is not
           dead, but will be removed when we remove the other
Line 102:  [oldMessageController handleMessage:message];
Line 103:  } ... ← Not dead, but will be removed


OldMessageController.m

Line 1:    @interface OldMessageController
… 498 lines …

Line 500:  @end
```

Fig. 1: An example of dead code that remains undetected via static analysis

Fig. 1 illustrates an example of dead code that remains undetected via static analysis. Syntactically, the module CommandFactory.m has a line 102 that calls the module

OldMessageController.m, such that static analysis will not consider line 102 dead, even if OldMessageController.m is never actually called during runtime. Effectively, the server never sends the client an 'old message,' and OldMessageController (underlined code) is effectively dead; yet static analysis fails to catch it. A static analyzer works if the entire codebase is available. This is typically not the case in client-server systems, in situations where third-party code calls into a codebase, in situations where clients run on disparate platforms, etc.

DESCRIPTION

This disclosure describes techniques to detect dead code based on the runtime behavior patterns of client software.

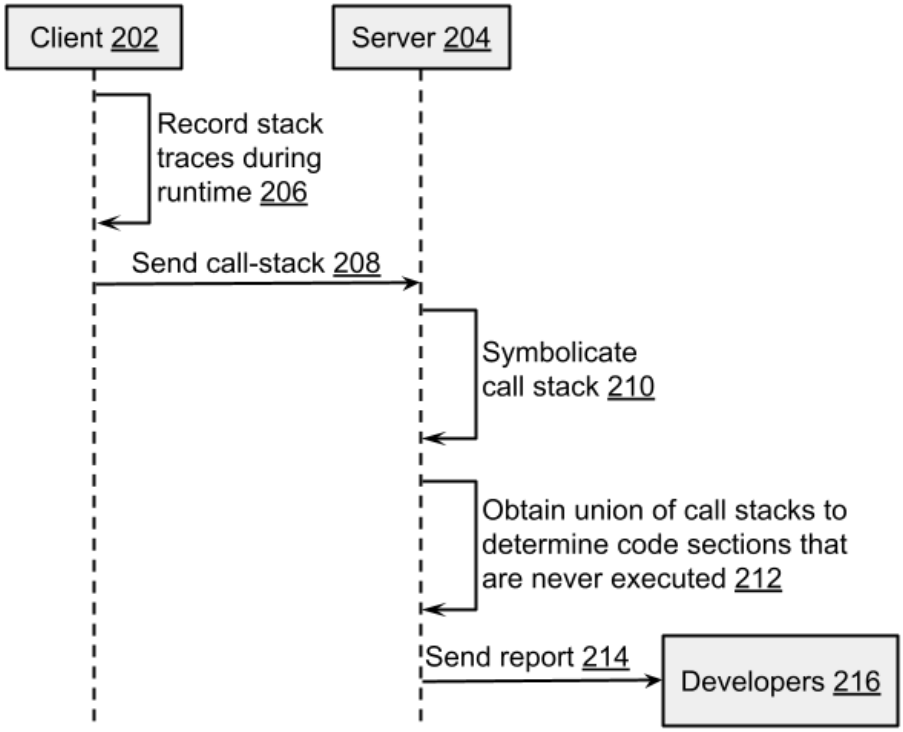

Fig. 2: Detecting dead code based on captured stack traces

Fig. 2 illustrates detecting dead code based on captured stack traces. A client (202) records stack traces during runtime (206). A stack trace includes a sequence of memory

addresses (the call stack) that indicates what was running at that instance. To distribute and optimize instrumentation load, stack traces are recorded only occasionally, e.g., at periodic intervals across a multiplicity (ensemble) of users, with user permission. Thus, rather than sample the run of a client millions of times from a single user device, with user permission, the runs of clients on a large number of user devices are each sampled once.

The call stack is sent (208) to a server (204). The call stack is symbolicated (210), e.g., client memory addresses mapped to lines of code within filenames. Symbolication is similar to procedures used by debuggers to pinpoint locations of runtime errors. While Fig. 1 shows symbolication being executed at the server, it is also possible to perform symbolication at the client. If symbolication is executed at the client, the client sends to the server not the call stack but the lines of code within filenames that were sampled during runtime.

Name of function / filename	The number of times called (executed) in one million runs
@function-X / file-A	767,345
@function-Y / file-B	334,567
…	…
@function-Z / file-C	1
@OldMessageController / OldMessageController.m	0
Line 102 / CommandFactory.m	0

Table 1: Report of the number of times various sections of code were executed

A union of received and symbolicated stack traces is performed to determine the number of times sections of code were executed (212). Alternatively, symbolication can be performed after performing a union of stack traces received in the form of memory addresses. For example,

as illustrated in Table 1, it can be the case that function-X (in file A) was observed being called many times (more than seven hundred thousand times out of a million runs), but other functions, e.g., function-Z (in file C), were observed being called very rarely, and one function, e.g., OldMessageController (including all of its five hundred lines), was never called.

Additionally, one line (line 102 in CommandFactory.m) was never observed being executed. Data such as Table 1 strongly indicates that line 102 of CommandFactory.m and OldMessageController.m are both likely to be dead code. Additionally, it may be the case that function-Z (in file C) is called so rarely that it isn't a popular feature, and it can be excised in the interest of reducing code bloat.

A report (214) detailing the number of times various code sections were exercised (similar to Table 1) is provided to developers (216). Developers can use the report for further code analysis and/or to remove dead code sections. Removal of dead code may reveal additional code sections that are dead, which can be removed by static analysis or other techniques.

CONCLUSION

This disclosure describes techniques to detect dead code based on the behavior patterns of client software during runtime. With user permission, stack traces are sampled and captured during runtime across a large number of devices running a given client software. Symbolicated call stacks are listed by frequency of execution to determine sections of code that are rarely or never executed. Rare or never-executed sections of code are reported to developers for further analysis to identify and remove dead code.