DeHydra Source Analysis Tool

Taras Glek *

April 10, 2007

Abstract

Static analysis tools are a mature technology. They can be applied to large codebases and can find certain classes of errors in large code bases. However even if the tools have an internal AST-level view of the code analyzed, they do not allow custom queries of the AST. Instead, they are largely used to check certain "canned" properties and either do not support user-extensible checks or provide limited domain-specific languages specialized for specifying automatons.

This paper describes DeHydra, a code analysis tool that can specify automatons in addition to allowing other code queries and checks. It uses JavaScript which is a general purpose programming language as opposed to a specialized DSL. Additionally, the tool allows some inter-procedural analyses instead of just intra-procedural ones.

1 Introduction

Mozilla[2] is a large project which in the default configuration produces over 2400 object files. While working on a source-to-source transformation tool for the Mozilla project it became clear that an AST alone is not sufficient and one needs to use static analysis in order to check that certain functions can be rewritten correctly. Additionally, when fixing a bug, optimizing code or re-factoring it is useful to find certain patterns in the source code. Some examples are:

- Finding functions that always return the same error code. These can be rewritten to not return an error code.
- Verifying that fields in structs are ordered efficiently.
- Finding code that interacts with garbage collection in an unsafe manner. This is similar to finding incorrect malloc/free uses.
- Building a callgraph.
- Finding dead functions, classes or even modules.
- When fixing a bug it is useful to be able to find all other similar cases in the codebase.
- Proving that certain functions are never called from a particular function. This is useful for security and performance audits.

*The author is employed by the Mozilla Corporation, which supported this work.

Since these are typical problems for any large-scale C++ project, existing tools were evaluated. These tasks require a tool that can do user-defined control-flow sensitive inter-procedural checks on C and C++ source code. Additionally the tool needs to be scale to large codebases.

Most static analysis tools do not address C++ due to the complexities of parsing the language. Coverity Prevent[4] and CQUAL[5] are exceptions. However, Coverity is proprietary so it is unclear whether it does control flow analyses. Additionally, it does not support user-defined analyses. CQUAL checks data flow instead of control flow so one can not express queries depending on code flow.

There are two open source tools that fit the user-extensibility criteria and support control flow checking.

1.1 UNO

UNO[1] is a lightweight static analysis tool with a DSL for checking user-defined properties. The C-like imperative DSL allows one to pattern match paths in control flow graph in order to find invalid ones. The DSL is limited to inter-function analysis. Unfortunately, UNO is based on a primitive C parser which does not support all C code present in Mozilla or parse C++. Due to limitation in the DSL, UNO also flattens the AST before pattern matching, losing information in the process.

Technical limitations aside, UNO is easy to learn and use, the DSL has room for future extensibility and serves as a good model for other tools to follow.

1.2 mygcc

Mygcc[3] is a static analysis backend that integrates into GCC and thus would be easy to integrate into most build systems in order to perform analyses. Additional benefit of using Mygcc is that it can parse anything that GCC can compile.

Mygcc is similar to UNO in that it is designed for intra-function analysis. It also does pattern matching over a control flow graph, but unlike UNO uses an even more restrictive declarative DSL. Like UNO, it also does not pattern match on types. It is unclear if it supports C++ or if the DSL could be extended to allow user analyses other than path-based error checking.

2 DeHydra

Due to limitations in existing tools, it was decided to implement DeHydra. It is a tool similar in spirit to UNO except that it supports user-defined analyses written in JavaScript, works with C and C++ and is capable of inter-procedural analyses.

The design of DeHydra is based on the following guidelines:

- Avoid building a direct competitor to Coverity, UNO or other existing tools. Instead DeHydra is to focus on niches not covered by other analysis tools. So no bounds checking, points-to or any other defect-finding analyses are implemented in the C++ or are part of the supplied JavaScript library.

- The tool should be equally good as a bug-finder and a query tool for source code

- Keep the full AST on the C++ side and only pass a simplified version to the JavaScript automaton. This way the CFG is easier to match on, and keeps the scripts from being overly C++ specific. This will allow the tool to be extended to also analyze JavaScript.

- Do not require the JavaScript scripts to implement all possible callbacks. This way the scripts end up shorter and easier to understand.

- Support persistence for incremental analyses.

2.1 JavaScript for Analysis

In order to shorten implementation time and provide more features JavaScript was chosen instead of a specialized DSL. This saved a lot of debugging time and provided DeHydra with a language that has a large developer base. Since JavaScript is a general purpose language, it allows more more sophisticated analyses than possible with a limited DSL.

By leveraging the SpiderMonkey[10] JavaScript runtime DeHydra was able to rely on a widely deployed and efficient language runtime instead of introducing and optimizing yet another DSL. JavaScript is sufficiently similar to the C-like DSL used in UNO that it was trivial to implement an UNO compatibility layer written in pure JavaScript such that UNO analyses could be ported with minimal changes.

JavaScript is also appealing in terms of features. It provides objects, closures and higher order functions which allow for concise and powerful pattern matching via functional programming.

Another benefit of using a scripting language is that it enables fast turn-around time. There is no need to recompile analyses for every change, resulting in faster incremental development. JavaScript's ability to print out any data structure to console is also a great debugging aid. Additionally, JavaScript is familiar to a lot of developers so the learning curve should be gentler than that for other static analysis tools' DSLs.

2.2 Control Flow Graph

DeHydra contains a control flow graph builder module that translates the Elsa AST into a control flow graph. It supports goto and return statements, and branch and iteration structures.

The graph vertices are basic code blocks connected with transition edges. Edges are labeled with the condition statements that must evaluate to `true` in order for the edge to be traversed. Figure 6 illustrates how a CFG is represented in DeHydra.

2.3 Control Flow Graph Traversal

To find errors such as the simple mismatched malloc/free check, DeHydra scripts are executed iteratively in depth-first traversal over every path between the entry and the exit blocks in the graph to prove that every malloc is always followed by a free. Scripts are not made aware of conditionals. From the script's perspective, functions are sequential basic blocks.

This is modeled by having two notions of state. There is global JS state, which corresponds to the global variables and there is a local state. For every basic block entered, local state is passed as the second parameter to the `process(vars, state)` function and the return value becomes an input state for subsequent basic blocks.

DeHydra attempts to avoid the exponential complexity of traversing every path in the graph. An edge is only visited if the same edge has not been previously visited with an equal input state. This requires scripts to make local state transitions independent of side-effects through the global JavaScript state.

```
int main(int argc, char **argv) {
char *c;
  if(argc)
    c = malloc(1);
  if(!argc)
    return;
  free(c);
}
```

Figure 1: Code with an unfeasible path

2.4 Symbolic Evaluator

Following every path in the CFG is computationally expensive, but also causes false positives because not all paths are feasible. Figure 1 shows the source code with an example of this problem, Figure 6 has the corresponding CFG. In the graph, there exists a path from malloc.cc:7 to malloc.cc:9, but the path will never be taken at runtime.

Initially it was planned to use value numbering to detect “equal” condition statements but that proved unsuitable since one still needs to evaluate expressions to check that their values are identical. Simple abstract interpretation is turned out to be good solution to the problem.

DeHydra infers variable values based on known values and checks guard conditions in order to determine if a conditional expression may evaluate to `true`. If an expression may evaluate to `true`, values of variables used in the expression are inferred to be one of `ZERO`, `NONZERO` or `UNKNOWN`.

Paths are deemed to be unfeasible if, in order for a conditional expression on an edge to evaluate to `true`, a variable along the CFG path is required to be both `ZERO` and `NONZERO`.

3 Running DeHydra

3.1 API

JavaScript sees a small and simplified subset of the abstract syntax tree so it will be possible to add another engine in addition to the current C/C++ parser to add support for analyzing other languages such as JavaScript, Java, etc.

When traversing a function’s CFG, JavaScript is provided with information on variable name, type and id. Additionally JavaScript is aware of structs, classes, virtual functions, class hierarchy information and bitfields. New types and fields are easy to add and are added as the need arises.

Table 2 lists callbacks that can be defined in user scripts and Table 3 lists the types passed as arguments. Note that both class structure and function bodies are exposed to the scripting API, allowing for conservative global analyses. For a complete list of built in functions, see Table 1.

4 Processing Source Code

DeHydra processes a source code unit-at-a-time. For analyzing Mozilla there is frontend script that produces a CPP-expanded .i file for every object .o file produced during a Mozilla build. Then DeHydra is applied to every one of those files.

Table 1: Built-in Functions

Function	*Description*
`error(string)`	Terminates DeHydra with an error message.
`print(string)`	Prints out a message, execution continues.
`read_file(string)`	Returns the contents of a file specified by the string.
`write_file(string, string)`	Writes the second parameter to a file specified by the first.
`is_global(id)`	Returns true when passed an id of a var object corresponding to a C++ global variable
`is_zero(id)`	Returns `true` if the variable is definitely zero.
`is_nonzero(id)`	Returns `true` if the variable is definitely not zero.
`graph(string)`	Writes out the CFG graph in .dot format to a the specified file.
`set_block_color(string)`	Colors the current basic block with the specified color

Table 2: JavaScript Callbacks

Function	*Description*
`process(vars, state)`	Called sequentially for every statement using variables in a CFG path. All path-related state is passed in via the state variable and out as a return value.
`path_end(state)`	Called once a path has been traversed
`graph_end()`	Called once the CFG for a function has been traversed
`process_class(class)`	Called for every struct and class declaration
`input_end()`	Called once all source files have been processed. Usually used for serialization

Scripts can save, reload and extend a persistent analysis state, to incrementally analyze a whole program. However, care must be taken when analyzing a large codebase like that of Mozilla to avoid running out of address space on a 32-bit system. DeHydra leaves such space optimizations to the script writer. It is simply a matter of being prudent and only retaining information required for the analysis and possibly spreading it across multiple files. Alternatively, one can switch to a 64-bit system.

5 Analyses Implemented

5.1 Bitfield Packing Checker

DeHydra can be used to avoid compiler bugs. For example, the struct in Figure 2 fits into 1 word on most compilers, but the Microsoft compiler will make that 2 words if the types of bitfields in the struct are not identical. Changing the `char` to an `int` fixes this problem. Figure 3 shows a DeHydra script that found all the remaining packing problems in Mozilla.

```
struct {
char ch:1;
int i:1;
};
```

Figure 2: This packs into 2 words in Microsoft C++ compilers.

```
function process_class(c) {
  var bitf
  for each (var t in c.members) {
    if(!t.bitfieldBits)
      continue
    else if(!bitf)
      bitf = t.type
    else if(bitf != t.type)
      error("Bitfield types in "+c.name+" disagree: "+bitf + " != "+t.type)
  }
}
```

Figure 3: A DeHydra script to find packing problems like in Figure 2

5.2 Ported malloc/free Checker

DeHydra contains an UNO-compatiblilty layer implemented in JavaScript. Figure 7 contains the JavaScript version. UNO version can be found in the tarball on the UNO website[1].

6 Persistence and Inter-Procedural Analyses

Without having any user-extensible tools for inter-procedural analysis to draw inspiration from it was hard to come up with a scalable design. Generally, tools have two modes of operation for inter-procedural analysis: inlining and a conservative mode. Inlining builds a CFG by inlining every function called from `main()`, whereas the conservative approach annotates functions and then uses callgraphs to propagate and verify inter-procedural properties.

Table 3: JavaScript Types

Type	*Description*
vars	An array of var objects.
state	An object that keeps path-specific state. It starts out as a JavaScript `undefined` value until it is defined by the script.
var	An object describing a variable-like expression. See Table 4.
class	An object describing a class. See Table 5.
member	An object describing a class member. See Table 6.

```
const GRAPH_FILE = "/tmp/inhertitance.js"

var graph = eval(read_file(GRAPH_FILE)) || new Object()

function input_end() {
  write_file(GRAPH_FILE, uneval(graph))
}

function process_class(c) {
  if(graph[c.name]) return
  graph[c.name] = c.bases.map(function(x) {return graph[x]})
}
```

Figure 4: A DeHydra script to generate an inheritance graph

Inlining the CFG graphs is not realistic for a project the size of Mozilla, so the conservative approach was implemented. Only `write_file` and `read_file` functions had to be written in C++ so scripts could save/restore their state with JavaScript `uneval()` and `eval()` when analyzing multiple files. The rest of the conservative analysis can then be done in pure JavaScript. However, at the time of writing there aren't any completed inter-procedural analyses.

6.1 Completing JavaScript Analyses in Web Applications

JavaScript was chosen for DeHydra for its easy embedding, efficient runtime and functional programming features. However, serialized DeHydra state produced with `uneval()` can also be loaded by a JavaScript script in a web page. Then the second part of the analysis can then be done in a web browser.

Modern Web browsers such as Mozilla Firefox support interactive vector graphics and can be used to build sophisticated visualization tools. DeHydra can be used as a backend to generate graphs for these web applications to visualize. One example is a class browser[7] implemented from a graph generated by code in Figure 4. Most IDEs struggle with the seemingly simple task of displaying a class hierarchy for a large project and either make the browsing too slow to be usable or run out of memory. On the other hand, a carefully written incremental class browser front-end could be easily implemented by relying on DeHydra as a backend.

Existing text tools that rely on regular expression such as LXR[8] could be complemented by AST-aware graphs produced by DeHydra.

7 Future Work

7.1 Nightly Analyses

When working on a large codebase like Mozilla it is easy to get lost in the source. Questions such as "what types of object hold references to objects of type A and its derived classes?" or "what functions call function B?" can take a long time to obtain an answer to manually. DeHydra can speed that up by generating graphs describing certain aspects of the source code. However, depending on the analysis these graphs can take a few hours to a day to produce for Mozilla.

Thus DeHydra could run a few generic analyses nightly, then the graphs would be made available for downloading.

With prepackaged nightly analyses, the developer would navigate to a page with an interactive JavaScript console, express these questions in JavaScript code and get the answer immediately.

7.2 Web Visualization

Displaying textual results of a query can be meaningless if there are thousands of matches returned, in fact it wont be much different than the result of a refined query with only hundreds of matches returned. An interactive application-specific code browser could be built with web technologies such as SVG and hyperlinking. That could be an efficient and aesthetically pleasing way to query a large codebase.

7.3 Dead Code Detection

Over time, a large project like Mozilla accumulates dead code. While dead code elimination is a well-known compiler technique, it is not wide-spread at code level. There are no open source dead code detection tools for C++. PC Lint[9] is a commercial package that claims to do dead code detection, but it is originally a C tool, so it is unclear how it deals with C++ features such as virtual methods or polymorphic methods. It is unclear if it can detect dead exported functions. Additionally, lint-type tools generally have a high signal to noise ratio, so it may be hard to find dead code warnings.

On the other hand, one can write a specialized script in DeHydra that would resolve all virtual methods, analyze all components in the repository to even be able to suggest removal of exported methods. An alternative analysis could use the class hierarchy and compare it against all invocations of constructors to detect unused classes.

7.4 Source-to-Source Transformation Assistance

Mozilla code typically uses out-parameters to return values and `return` statement to indicate if an error has occurred. Additionally, at the call-site the return value is checked for errors: if an error occurs, the caller typically returns with that error. Thus, there are at least two values "returned" from every getter and there is a large amount of code within Mozilla that just checks for error return values only to propagate them further.

In the long term the solution is to use the `return` statement to return values and C++ exceptions to propagate errors. However, switching to exceptions is a big task. It requires doing a Mozilla-wide exception safety analysis, converting unsafe code to be exception-safe and may take a long time to write a tool that can do both the analysis and source transformation correctly.

A simpler rewrite task would be to convert the really simple getters to return values on success and `NULL` on error. Callers would have to be rewritten accordingly, but it would be a much smaller task that switching to exceptions. Figure 5 contains an example getter. DeHydra can detect these, by checking parameter types and checking the CFG to make sure that there are only two possible return values: `NS_OK` and some error. Then the serialized callgraph described in Section 7.1 would be referenced to provide all of the call sites. The names of the getter and callsites would then be written to a config file to be used as input to the source transformation tool.

```
nsresult nsBidiPresUtils::GetBidiEngine(nsBidi** aBidiEngine)
{
  nsresult rv = NS_ERROR_FAILURE;
  if (mBidiEngine) {
    *aBidiEngine = mBidiEngine;
    rv = NS_OK;
  }
  return rv;
}
```

Figure 5: An example of typical function that would be rewritten

8 Conclusion

Static analysis is a mature technology for finding defects in software, and can also be used for querying code. This requires the static analysis tools to allow the user to specify queries. However most tools do not support user-customizable analyses, shy away from C++, or are exclusively focused on defect finding. There are two existing tools that come close to offering code querying abilities. Both Mygcc and UNO allow user-scripting, but apart from other limitations, their scripting languages still do not allow for general queries against a code-base.

DeHydra is a new general static analysis tool for C++. While most tools provide “canned” and hard to customize hard-coded analyses, DeHydra only allows analyses in JavaScript – drawing a clear line between parsing code, CFG building, abstract interpretation and source analysis scripts. By reusing the Elsa C++ parser and the SpiderMonkey JavaScript runtime, DeHydra development mostly focused on the static analysis aspect. Most of the development effort went into the control flow graph builder, abstract interpreter and API design.

DeHydra has already found bugs in the source code and shown potential for more complicated future analyses.

List of Figures

1 Code with an unfeasible path 4
2 This packs into 2 words in Microsoft C++ compilers. 6
3 A DeHydra script to find packing problems like in Figure 2 6
4 A DeHydra script to generate an inheritance graph 7
5 An example of typical function that would be rewritten 9
6 CFG for code in Table 1 10
7 Malloc checker implemented in JavaScript 12

List of Tables

1 Built-in Functions 5
2 JavaScript Callbacks 5
3 JavaScript Types 6
4 Fields in the var object 11

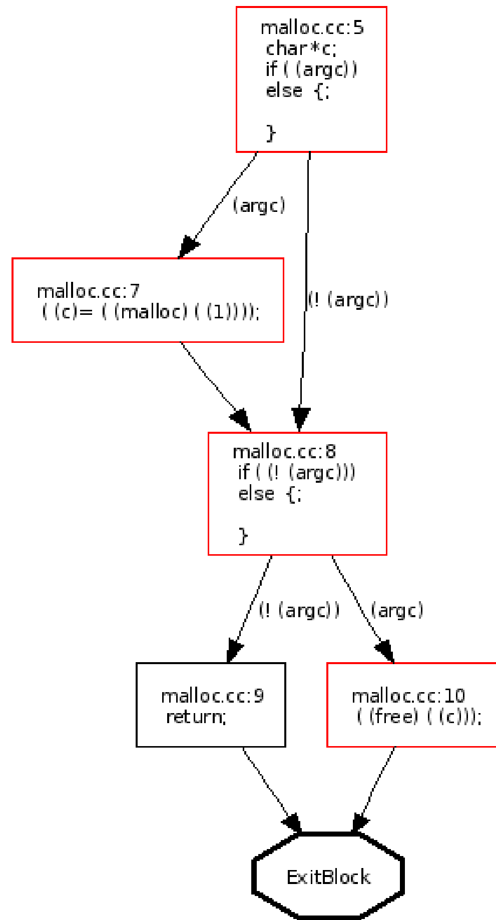

Figure 6: CFG for code in Table 1

Table 4: Fields in the var object

Field	*Description*
`name`	Symbol name
`type`	C/C++ type.
`id`	Unique id (for a single dehydra run) used for tracking variable use throughout the source because names in C++ refer to different things depending on the scope.
`assign`	An array of var objects used in assignment: symbol = (exp). Note, in the future assign will point directly at another var object instead of using an intermediate array.
`decl`	String describing location of symbol declaration: "example.cc:3/symbol"
`fieldOf`	Used for field access: (expr).symbol. Points to the var object that defines (expr).
`isFcall`	Used to call a function: symbol(x).
`isUse`	Evaluated to derive value: x = symbol.
`isAlias`	Address taken: &symbol.
`isDecl`	Symbol appears in a declaration: Foo symbol.
`isDeref`	Dereferenced *symbol.
`isParam`	parameter: function(Foo symbol).
`isReturn`	Used in a return value: return symbol.

Table 5: Fields in the class object

Field	*Description*
`name`	Class name
`bases`	Array of strings of names of base classes.
`members`	Array of member objects.

Table 6: Fields in the member object

Field	*Description*
`name`	Member name
`type`	C/C++ type.
`decl`	String describing location of member declaration, same format as the `decl` field in the var object.
`bitfieldBits`	Array of member objects.

```
function uno_check(vars) {
  if (select("malloc", [FCALL]))    // unmarked symbols of type function call
  {
    if (select("", [DEF]))    // unmarked symbols DEFined in those stmnts
    {
      if (match(1, [DEF]))  // are there matching symbols with mark 1?
        error("malloc follows malloc");
      else {
        mark(1);    // mark 1
      }
    } else
      error("result of malloc unused");
  }
  else if (select("free", [FCALL]))
  {
    if (select("", [USE]))
    {
      if (match(1)) {
        unmark();   // remove mark
      }
      else {
        error("free without malloc");
      }
    } else
      error("no argument to free");
  }
}

function path_end(state) {
  if (marked(1))
  {
    if (!known_zero())
      error("malloc without free");
  }
}
```

Figure 7: Malloc checker implemented in JavaScript

5 Fields in the class object 11
6 Fields in the member object 11

References

[1] G.J. Holzmann, UNO: a simple tool for source code analysis. `http://www.spinroot.com/uno/`

[2] Mozilla Project. `http://www.mozilla.org`

[3] N. Volanschi, and S. Pop, Mygcc. `http://mygcc.free.fr/`

[4] Coverity Prevent. `http://www.coverity.com/`

[5] J. Foster, M. Fhndrich, A. Aiken. A Theory of Type Qualifiers. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Atlanta, Georgia. May 1999

[6] Elsa/Oink parser. `http://www.cubewano.org/oink`

[7] Class browser prototype `http://people.mozilla.org/~tglek/graph/examples/classes.html`

[8] LXR code browsing tool. `http://lxr.mozilla.org/seamonkey/`

[9] PC-lint. `http://www.gimpel.com/html/pcl.htm`

[10] SpiderMonkey JavaScript engine. `http://www.mozilla.org/js/spidermonkey/`