GBIL: Generic Binary Instrumentation Language

Andrew Calvano

September 24th, 2015

Introduction:

Analyzing machine-targeted, compiled code is a much more difficult task then analyzing software in which source code is available. Often times, commercial software does not come with source code resulting in binary code, in the form of libraries and executables, being all that is available to the developer using the code. Binary code has a lot less apparent information then source code and the implementation of a binary only distribution is often mysterious. As a result, when an error is encountered it can take days, weeks, or even months to debug and reverse engineer.

Having used many different types of closed-source software in my day to day tasking professionally, I often want to analyze and collect data on the behavior of the unknown code I am using. It is useful to know how long a specific API call in a third party library is executing for, or how well a test suite is performing during unit testing. Instrumenting closed-source software allows a user to have introspection on what is going on in software that otherwise would be treated as a blackbox.

From a security standpoint, binary instrumentation allows complex types of analysis to be performed on closed source software. One can insert instrumentation into target code that allows for the collection of code coverage information which is a useful metric when performing automated random tests such as fuzzing[1]. It should be noted that these types of analysis rely on the runtime behavior of the target process meaning that they can not be run on the code at rest as in a static analysis. Thus, these types of analyses are typically categorized as dynamic analyses.

There exist a number of binary instrumentation frameworks, however, each has pros and cons depending on the type of code it is instrumenting. Some frameworks specialize in static binary instrumentation which is the process of inserting code directly into the target binary before it is executed. When the binary is executed the inserted code will also be executed allowing the user to obtain run time information. Other frameworks specialize in dynamic binary instrumentation which is a type of instrumentation that monitors the code being executed at run time and inserts instrumentation when it observes a user specified event. Additionally, each framework has its own esoteric API usually written in C or C++ that has a learning curve.

Language Proposal:

I am proposing the Generic Binary Instrumentation Language (GBIL) as a language to unify the existing binary instrumentation framework tools so that one domain specific language for binary instrumentation can be used to target each framework. The GBIL language will allow a user to generically specify the type of instrumentation he or she wants to insert into the target binary. The GBIL language will be a compact language that allows a developer to quickly prototype their instrumentation scheme and compile it to one of several back end binary instrumentation framework APIs. This means that GBIL source code will be compiled into C or C++ code utilizing the APIs from one of several binary instrumentation frameworks such as Intel PIN[2], DynamoRIO[3], or Paradyn's Dyninst[4].

There is existing research that create languages for binary instrumentation but none generically define a language that can be used to target multiple frameworks. For example, the MIL language[5] was developed to express analysis in the MAQAO framework but can not target any other framework. There is also the MDL language[6] created by Paradyn, the creators of the Dyninst instrumentation framework. While MDL is a platform independent instrumentation language that is capable of targeting multiple machine architectures it is limited to the Paradyn tool set.

To create GBIL, I will need to coalesce the common features shared among many of these binary instrumentation frameworks to create an accessible way for a GBIL developer to express their intent to use a common feature. For example, each instrumentation framework supports the ability to install instrumentation functions that insert code at the instruction level, basic block level, and function level. Thus, GBIL must be designed with the ability to express the same type of instrumentation functions.

Language Overview:

The GBIL language is meant to unify various features shared between many binary instrumentation frameworks such that a developer using the GBIL language will be able to generically specify how they want their target binary to be instrumented. The GBIL source code should also be very easy to write and understand, and be an easy platform for developing a prototype instrumentation setup. No knowledge of any target instrumentation API should be needed by the developer in order to write a GBIL program.

Types

The GBIL language will be statically typed. There will be a small number of basic variable types in the GBIL language. These are the uint (unsigned int), sint (signed int), float, list, map, string, and file types. There are also multiple types of functions. Depending on the type of function, there are “special” variables that can be accessed from within the scope of that function. For example, there is a basic block instrument function that is executed every time a basic block event is triggered in the instrumentation code. From within the basic block function, a developer can access a special variable called “$begin_address”, which corresponds to the address of the first instruction in the basic block and a variable “$end_address” which corresponds to the address of the last instruction in the block.

The uint type is synonymous to the uint64_t type in C or C++ presented in the inttypes.h header file. The sint type is exactly the signed opposite and corresponds to the int64_t type. These two types are included to allow the developer to specify collection variables to log things like the number of times an instruction has been executed, or the number of times a specific function has been called.

The float type is included to allow the developer to specify statistic variables to collect statistics on the running process. This type could be used to store things such as the distribution of load instructions relative to other instructions executed.

The string type will exist primarily for output purposes. The developer will be able to create strings that can be outputted to a file. The purpose of this is to examine the output of the instrumentation code after the monitored process has finished executing.

The file type is very similar to the FILE * type in the C language. It exists to represent a file on disk that can have data written to it. An instance of the file type can also be read from. A developer may want to read from a file when they want to serialize or extract results from a previous instrumentation run.

The list type is present to allow the user to work with lists of objects of a specified type. All elements in the list must be of the same type. For example, one will be able to create a list of sints, a list of lists, a list of floats, etc.

The map type exists to associate objects with each other. It will function much like a dictionary type where an object can be looked up in the list as a key and its corresponding value can be extracted. As an example, one can use this type to associate a list of memory addresses that have been resolved from a variable memory access with an instruction address. Additionally, one can use this to keep track of a hit count for an instruction address where the map keeps track of how many times a given instruction has executed.

There are eight types of functions that are available to the developer. Depending on the type of function the developer has access to special variables that are relevant to the type of function that is being executed. The eight types of functions are: instrument initialization, instruction instrument, basic block begin instrument, basic block end instrument, function begin instrument, function end instrument, termination, and utility. Each type of instrument function has a yet to be determined set of special variable associated with them. These special variables exist to allow the developer to operate on constructs that they normally would want to operate on in the scope of a similar API instrumentation function in the instrumentation framework. I gave one example of this earlier, but another could be the variable “$instruction_address” from within the scope of an instrument instruction function type. The “$instruction_address” variable will return a uint representing the address in the process' virtual address space containing the binary code for the instruction.

Operations

Each type has a number of operations that can operate on an instance of the specified type.

There are a number of polymorphic operators that can operate on multiple built in types. These operators are standard in many languages such as C, C++, and Java. These are addition (+), subtraction (-), multiplication (*), division (/), equality (==), inequality (!=), greater than (>), greater than or equal to (>=), less than (<), less than or equal to (<=), and assignment (=).

All of these operators can operate on uint and sint types. The boolean operators: equality,inequality, greater than, greater than or equal, less than, and less than or equal, can operate on every type except the float type. For list and map types, only the equality and inquality operators are supported which will recurse through the data structure and compare at each level of recursion.

There is also the bracket operator ([]) used for accessing the list and map data structures. For lists, the bracket operator is used to index into the list. For maps, it is used to access a particular value for a supplied key.

There will be a builtin function len() that will operate on list objects. The len variable will return the length of the list in an instance of a uint type.

Maps and lists can be searched with a quickhand notation as follows:

```
if element in map
{
  map[element] +=1;
}
else
{
  map[element] = 1;
}
```

List can be substituted in the above code for map with the exception that it can be dereferenced with element as an index.

Control Flow

The function level control flow for the GBIL language is event based. Events occur as the instrumented process is executing. There are 7 types of events: initialization, function begin, function end, basic block begin, basic block end, instruction, and termination.

Control flow in the GBIL language from within a function happens, in general, with the order of the desired behavior to execute happening in the order specified in the program. All functions have access to the looping constructs of the for loop, the while loop, and the do-while loop. These constructs operate in the same way they do in languages like C, C++, and Java. There is also the conditional construct if-then-else which operates as expected.

Event functions can not be recursive but there can be more then one event function. If there is more then one event function all are assumed to execute every time an event trigger happens. Each event function can also call utility functions. Utility functions can be recursive. Scope is not carried over from an event function to a utility function. Utility functions behave in much the same way as a function in the C language. There is a return type associated with the function and a list of typed, expected arguments. If a utility function wishes to access a special variable the special variable must be passed as an argument to the utility function from the event function.

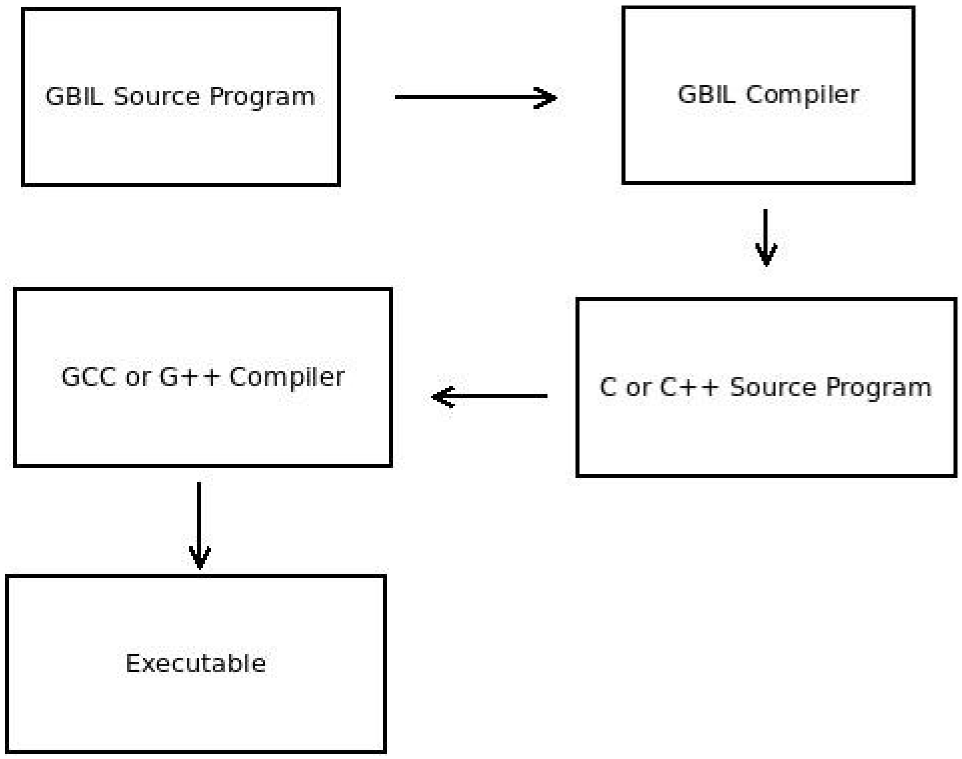
Variable Scope

All variables must be declared before any statement regardless of scope. To create a globally accessible variable, whose symbol is available throughout each instrument function type, the variable must be declared in the initialization function. Other then globally accessible variables, locally accessible variables can be defined in the beginning of a function definition. Variables can not be created inside of a looping construct or an if then else construct.

If a locally declared variable shares a name with a globally declared variable, the locally declared variable will override the resolution procedure for the shared symbol.

Compiler Architecture:

The GBIL compiler will accept input in the form of a GBIL source file, it will then translate this source file into one of the binary instrumentation framework's expected format. Thus, it will produce a C or C++ source file using the specified API for the framework. The C or C++ file will be passed to the gcc or g++ compiler which will produce the final executable.



Example Program:

```
@function_begin
function handle_function_begin {

 call_context.push($func_begin);

 if $func_begin in function_cover_map
 {
  func_cover_map[$func_begin] += 1;
 }
 else
 {
  func_cover_map[$func_begin] = 0;
 }
}

@function_terminate
function handle_function_end {

 call_context.pop();
}

@basic_block
function handle_basic_block {

 if $bb_begin in basic_block_cover_map
 {
  bb_cover_map[$bb_begin] += 1;
 }
 else
 {
  bb_cover_map[$bb_begin] = 0;
 }


}

@instruction
function handle_instruction {

 //Demonstrating special variable
 if $instr_address in cover_map
 {
  instr_cover_map[$instr_address] += 1;
 }
```

```
  else
  {
   instr_cover_map[$instr_address] = 0;
  }
}

@init
function initialize {

  //Global data structures must be declared in the initialization routine
  map<uint,uint> instr_cover_map;
  map<uint,uint> basic_block_cover_map;
  map<uint,uint> function_cover_map;

  list<uint> call_context;
}

@terminate
function terminate {
  //Empty functions are allowed.
}
```

This example program demonstrates the desired syntax of the GBIL language. This proposal does not go into detail to discuss the exact syntax for the GBIL language but the important parts of the syntax are demonstrated in the above program.

“{}” are used to create blocks and bodies for functions. Each statement other than function definitions must be followed by a semicolon. Whitespace is unimportant. Function definitions must be attributed with the @type and preceded by the “function” keyword before the body is declared. For polymorphic types, such as list and map the type of the object are specified in templated syntax using the “<.,.>” syntax which is similar to C++ templates. Nested blocks are not allowed.

Complete syntax and more fine grained details regarding operators and types will be discussed in the language reference manual.

References:

[1] American Fuzzy Lop; http://lcamtuf.coredump.cx/afl/

[2] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." *ACM Sigplan Notices*. Vol. 40. No. 6. ACM, 2005.

[3] DynamoRIO; http://www.dynamorio.org/home.html

[4]Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (November 2000), 317-329. DOI=10.1177/109434200001400404 http://dx.doi.org/10.1177/109434200001400404

[5] Charif-Rubial, A.S.; Barthou, D.; Valensi, C.; Shende, S.; Malony, A.; Jalby, W., "MIL: A language to build program analysis tools through static binary instrumentation," in *High Performance Computing (HiPC), 2013 20th International Conference on* , vol., no., pp.206-215, 18-21 Dec. 2013

[6] Hollingsworth, J.K.; Niam, O.; Miller, B.P.; Zhichen Xu; Goncalves, M.J.R.; Ling Zheng, "MDL: a language and compiler for dynamic program instrumentation," in *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on* , vol., no., pp.201-212, 10-14 Nov 1997