Memory Safety Bug Hunting with BAP

Tiffany Bao, Dominic Chen, Rijnard van Tonder

April 29, 2015

1 Introduction

The Problem

Although numerous debugging tools have been developed for performing static and dynamic software analysis, the majority of these require access to source code or utilization of special compiletime flags to embed debug information within the output binary. In contrast, little effort has been spent on developing static binary analysis tools which can directly analyze offtheshelf executables without executing them. For this project, we plan to focus on a subset of common programming bugs that tend to introduce security vulnerabilities, known as memorysafety errors. Examples of these include buffer overflows, double frees, and uninitialized variables. In particular, we define *Security Outcomes* with respect to certain memory-safety errors, and subsequently implement and test static analyses that yield these outcomes. All analyses were written against our group's Binary Analysis Platform (BAP) [1, 3].

A factor of this research area that appeals to many is the ability to conclude whether vulnerabilities do or do not exist by performing an analysis purely on the binary level. Due to the inherent difficulties of operating at this level, our approach to the problem starts out coarse, and continues to be incrementally refined as we become more familiar with the numerous barriers that inhibit vulnerability detection in binaries. In this respect, we have much to say towards lessons learned in §5.

1.1 Approach Overview

At the outset of this project, we planned to implement and test the static analyses with respect to our original proposal:

1. Intraprocedural data dependency resolution of instructions
2. Intraprocedural detection of unchecked memory allocations
3. Basic intraprocedural detection of buffer overflows in the stack
4. Basic intraprocedural detection of buffer overflows in the heap

Equipped with the knowledge of data flow dependency between instructions (1), we determined to reason about various properties of typical vulnerabilities that we could check (2-4). For example, we might like to follow how arguments to a 'strcpy' call are influenced by previous instructions as a function of the stack register.

1.2 Related Work

Engler et. al. found numerous bugs with system-specific static analyses, implemented by way of a meta-compilation language for the `gcc` compiler [5, 6]. Analyses are composed in the form of correctness rules and automatic inference of rules form source code. Memory errors that are checked include null pointer bugs, not checking allocation results, and uses of freed pointers. This relates to our goal in that we would ideally like to perform similar checks, albeit at the level of complexity of binary code. Whereas Engler's work matured into a generic platform for specifying analyses, we must start first with an ad-hoc approach to gain insight and familiarity in the problem domain presented by binaries.

The Jakstab framework [7] provides a platform for analyzying binaries. However, it is not oriented toward detecting security vulnerabilities, but rather recovering abstractions from binaries (e.g., CFG reconstruction). Frama-c is an example of a framework which allows one to check various properties, including security properties, but works on the source level [4].

1.3 Contributions

Our key technical contributions are as follows:

1. A dataflow analysis framework for BAP.
2. General plugins for BAP that deliver
   (a) reaching definitions and
   (b) data flow dependency resolution (use-def and def-use chains).
   (c) registers corresponding to function call arguments and returns, according to the ARM ABI.
3. Security-specific plugins for BAP that produce security outcomes by checking
   (a) whether memory allocation checks are performed on 'memcpy',
   (b) whether calls such as strcpy are safe from buffer overflows under specific conditions, and
   (c) whether malloc calls...
4. Results of testing the aforementioned plugins on the `GNU Coreutils` software suite.

1.4 Structure

Due to the exploratory nature of this project on the binary level, we concentrate much of our efforts on looking at viable approaches of analysis, rather than optimizing properties of the analysis or testing against measurable benchmarks. For this reason, it doesn't come as a surprise that our evaluation section may be less expressive than our approach and "lessons learned".

2 Approach

2.1 Overview

Here we discuss our approach toward ensuring security properties as it evolved over the course of the project. Initially, we determined to reason about each of the cases in §1.1 that relate to vulnerabilities (2-4). To do so, we needed some basic infrastructure (1) which we describe first.

Thereafter, we describe our specific approaches to performing security analyses with respect to specific *Security Outcomes*.

2.2 The Case for Reaching Definitions

For example, consider the following code, where we desire to verify that the return value of `malloc` is checked:

Listing 1: malloc.c

```
#include <stdlib.h>
#include <stdio.h>

int main() {
  // Uninitialized variable
  char *array = 0;


  array = malloc(32 * sizeof(*array));

  if (!array)
    return -1;

  free(array);

  return 0;
}
```

The disassembly of this binary might appear as follows (with two basic blocks, before and after the call):

Listing 2: Malloc disassembly

```
begin(main_ENTRY)
    000082c8: 04 e0 2d e5    str lr, [sp, #-4]!
    000082cc: 20 00 a0 e3    mov r0, #0x20
    000082d0: 49 00 00 eb    bl #0x124          ; call malloc
end(main_ENTRY)

begin(main_0xc)
    000082d4: 00 00 50 e3    cmp r0, #0x0       ; check malloc return value
    000082d8: 02 00 00 0a    beq #0x8
end(main_0xc)
```

We would like to confirm that a check takes place after the call to malloc; i.e., we would check for the presence of the `cmp` instruction and the operand `r0` in `main_0xc`.

However, it may easily be the case that the contents of `r0` are first moved to another register, `r3`, which is subsequently checked by a `cmp` instruction. In order to verify the buggy condition that the malloc return value is never checked, we must verify that any flow of this data from `r0` to other registers are also not checked. To obtain the statements where `r0` data flows to, we make use of def-use chains. For the definition of `r0` in the BIL IR, we check all uses of `r0` and transitively look up the def-use chains for further registers, such as `r3`, which the value in `r0` flows to. On the final set of instructions, we verify that the data is subject to a `cmp` instruction.

This example demonstrates the need for def-use chains; our stack and heap based analyses in turn also require use-def chains. Due to retreating edges in the CFG, it is difficult to produce def-use chains given a single instruction. For this reason we first implemented a dataflow framework and a reaching-definitions pass. An exhaustive description of the implementation details for this framework, as implemented in OCaml for BAP, may be viewed in Appendix A. From the reaching definitions, we generate def-use and use-def chains as necessary—e.g., we filter on those reaching definitions which concern a particular use or def of a register.

2.3 Data Flow Dependency

An extension to the def-use and use-def chains was to produce the data (flow) dependencies of a given statement in a forward or backward direction, respectively. Essentially, given a register in a definition (such as a Move instruction in the BIL IR), we can produce a set of all instructions that are determined to transitively affect its value. We implement this by way of a data dependency plugin for BAP. Consider the arguments to the following `strcpy` call:

Listing 3: strcpy.c

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv) {
  char dst[10] = {0};
  strcpy(dst, argv[1]);
}
```

For the `dst` variable on the stack, encoded as an `<address,statement>` pair, we have:

```
<(0x8490:32, 0)> R0 := R2
```

The plugin will determine all preceding instructions it is dependent on:

```
<(0x8444:32, 3)> SP := SP - 0x8:32
<(0x8448:32, 2)> R11 := SP + 0x4:32
<(0x848C:32, 2)> R2 := R11 - 0x10:32
```

Now, we can observe the role that the stack pointer `SP` plays when considering `dst`. A full listing of the BIL disassembly, the data dependencies for `argv[1]`, and a graphical output may be viewed in the Appendix B.

2.4 Malloc Checking Security Outcomes

For Malloc checking, we suppose that a security program should always check the return value of `malloc`. This is a strong yet reasonable requirement for ensuring secure code, because if the pointer fails to be assigned a value due to a lack of space for allocation, an attacker may be able to exploit a NULL pointer dereference at a later point in the program.

To demonstrate how our plugin check works, observe the example shown in 2.2. In this example, the function `malloc` is called at address `0x82d0`. Followed by the instruction, we have a comparison instruction which compare the return value `r0` to constant value `0`. This piece of code is thus a candidate for `malloc` checking.

Note that the comparison instruction is not necessarily available immediately after the malloc call instruction. Also, a more complicated case could be that the return value is passed to another register which in turn is checked instead of `r0`.

Another example that we came across is from binary coreutils_O1_[, which is shown as following:

Listing 4: Malloc disassembly

```
d4d0:       e0800009      add     r0, r0, r9
d4d4:       e280a002      add     sl, r0, 2
d4d8:       e2800003      add     r0, r0, 3
d4dc:       ebffee80      bl      8ee4 <malloc@plt>                *
d4e0:       e1a08000      mov     r8, r0                           *
d4e4:       ea000006      b       d504 <locale_charset+0x208>
d4e8:       e0891000      add     r1, r9, r0
d4ec:       e081100a      add     r1, r1, sl
```

```
d4f0:         e281a002         add      sl, r1, 2
d4f4:         e1a00004         mov      r0, r4
d4f8:         e2811003         add      r1, r1, 3
d4fc:         ebffee4b         bl       8e30 <realloc@plt>
d500:         e1a08000         mov      r8, r0
d504:         e3580000         cmp      r8, 0                          *
d508:         1a000005         bne      d524 <locale_charset+0x228>    *
d50c:         e1a00004         mov      r0, r4
```

In this example, after calling malloc at address d4dc, the program copies the return value to register `r8`, and then to address `d504` where it compares the value of `r8` to constant 0. The reason of compiling the program in such way is mainly because of optimization. Specifically, this helps decreasing the redundant code, by reusing `d504: cmp r8, 0` and the subsequent instructions.

To address this problem, we use the forward-data-deps plugin to help finding the instructions that depends on the return value. Here dependence means that one or more variables used in the instruction depends on register `r0`. For example, given instruction `d4e0: mov r8, r0`, we find that instruction `d504: cmp r8, 0` depends on `r0`, because the value of `r8` is from `r0`. Also, instruction d508 depends on r0, because the flag registers are determined by the `d504`, and in `d504` `r8` depends on `r0`.

Looking back to the example, and using forward-data-deps plugin, we get all dependence instructions once we find a move instruction with `r0` as source. For In the next step, we check if there is `cmp` instruction among all instructions we found. If there is one, we consider call is safe. If not, we consider the call unsafe.

2.5 Stack-based Security Outcomes

A number of unfruitful approaches documented in §5 lead us to opting for an approach where we identify two properties which are important from a security perspective, without inferring sizes of stack-based buffers.

`src` is a constant If we can determine that the `src` buffer is a constant, (and therefore not dependent on any user input), the `strcpy` call is safe. While a good property to check, it was found that a constant value corresponding to the `src` argument didn't occur in practice. Depending on compiler optimizations `strcpy` (as well as `sprintf` calls) would be compiled to a `LDMIA.W` ARM instruction which directly copies memory. Lastly, we observed that `strcpy` and `sprintf` may be replaced by the `__strcpy_chk` and `__sprintf_chk` counterparts, which are less interesting to analyse from a security standpoint.

Nevertheless, we implemented a partial check of this case in the plugin, and assumed the standpoint that any `src` address originating from the `ro.data` section implies a constant, fixed value. However, for this check we only consider the statement which loads a value immediately before the call, and not originating previous statements.

Stack accesses within the frame For this outcome, we confirm that all instructions which access elements on the stack, plus a fixed offset, are within the stack range of a function. We infer the size of the frame by considering the statements that manipulate SP in the function prologue. While we didn't observe any cases where the stack range was exceeded, this type of check forms the basis for further, more sophisticated improvements.

2.6 Malloc-based Security Outcomes

3 Experimental Setup

Our experimental setup consisted of running our plugins with BAP `0.9.6` [1] on a set of binaries, including GNU `Coreutils` and our own examples.

3.1 Unchecked Malloc

3.2 Stack Check Plugin

The stack check plugin was tested against the GNU `Coreutils` suite. We primarily chose the `Coreutils` suite since a) it can be compiled for ARM, and b) it contains numerous calls to `strcpy`, `memcpy`, `malloc`, and so forth. The stack check plugin, while configurable, concretely checked for function calls to:

- `strcat`
- `strcpy`
- `strncpy`
- `memcpy`

The plugin collected a number of statistics on the binaries, including the number of argument sites (that is, statements that set up registers before a call), the number of functions which contained calls to 'dangerous' functions, and the maximum number of statements on which an argument site statement was dependent. These are illustreated in Figure 1.

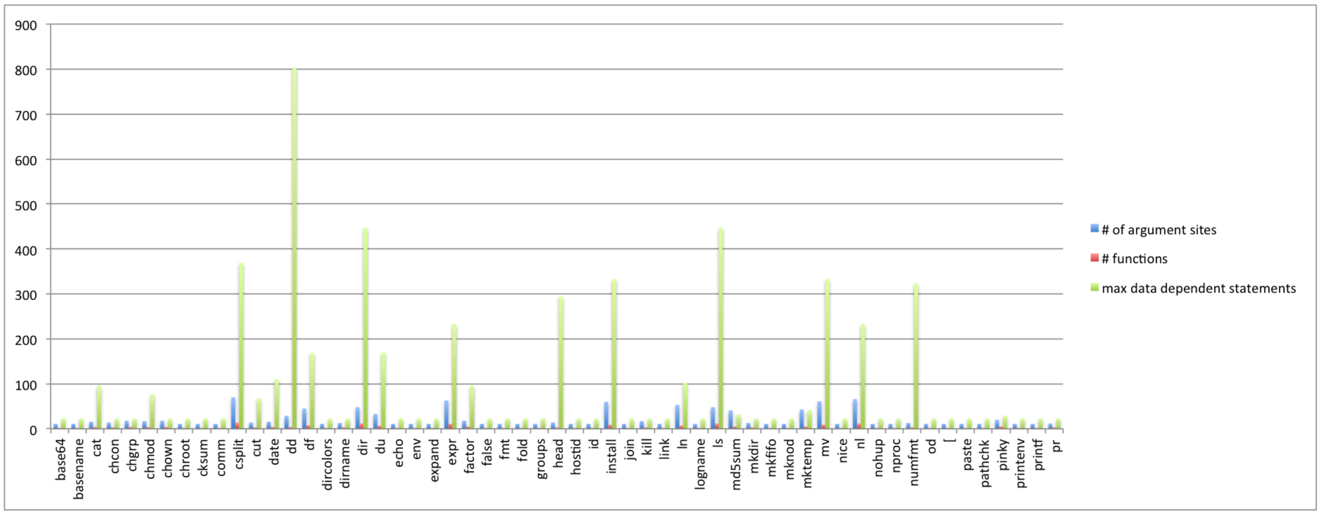
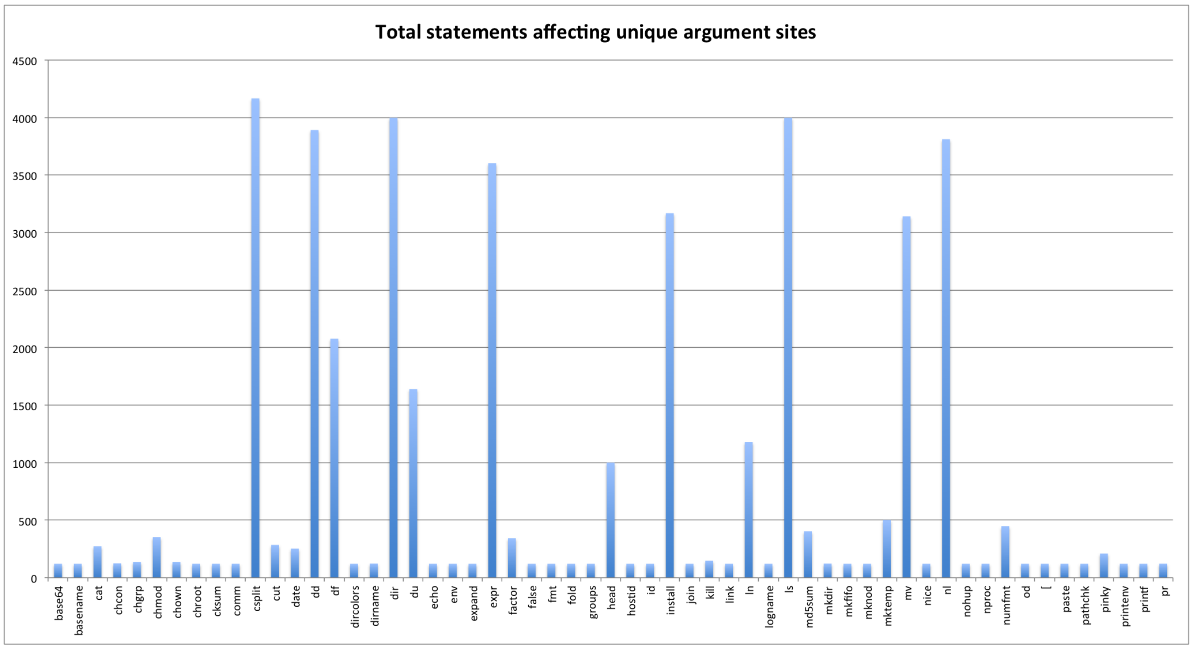


Figure 1: Coreutils statistics generated by plugin

Figure 3.2 presents the total number of statements in a binary which affect unique argument sites.



The figures illustrate some interesting properties with respect to argument sites and data dependencies. Consider that each potentially dangerous function takes 2 or 3 parameters–of these, our data dependency plugin determined the maximum number of statements a single argument site statement is dependent on. We observe from Figure 1 that this could number in the hundreds, whereas the total number could range in the thousands (Figure 3.2). The implication here is that any additional reasoning and refinement to static analysis methods across cases composed of sizeable dependencies need to be carefully considered. For instance, if we add forms of pointer analysis, we may speculate about the scale of complexity of memory operations for argument sites.

We did not observe any violations with respect to the security outcomes in §2.5. This is perhaps unsurprising, given the ubiquitous use of the `Coreutils` suite. At the same time, it is difficult to choose a test suite which purposefully exhibits the stack properties that we check; see §5 for further discussion.

3.3 Malloc

4 Experimental Evaluation

In general, we were pleased that our plugins ran successfully on our own example binaries as well as `Coreutils`. A key aspect of evaluating the success of our project is that we were able to start with a basic API in BAP, and after implementing a non-trivial amount of 'infrastructure' code, arrive at three individual plugins which are able to produce security outcomes.

Our evaluation of these plugins are largely qualitative rather than quantitative at this stage. However, they serve as very useful data points toward developing more sophisticated analyses.

4.1 Malloc Check Plugin

4.2 Stack Check Plugin

As mentioned, our stack check plugin did not trigger any violations when run over the `Coreutils` test suite. This result was largely anticipated: the `Coreutils` library was chosen so that we could verify that our plugins run and to rule out the possibility that binaries in `Coreutils` exhibited rudimentary flaws.

In critique of this plugin, there are a number of stack-based errors that could occur despite our checks. For example, consider a stack frame containing multiple local buffers at runtime. It is feasible that a buggy `strcpy` call could copy bytes beyond the boundary of a single buffer into an adjacent buffer, while remaining within the stack bounds. Our plugin would not detect this case. Consider a further case where memory accesses to the stack are dependent on general purpose registers—here we cannot conclude whether the access is within range of the stack frame.

A positive side-effect of the plugin is the potential of reusability, and by extension, refinement toward more sophisticated analyses. Recall that this plugin collects and makes use of statements that an argument site depends on. Thus, we have the ability to store the output of our analysis without having to rerun the plugin. For example, we may be able to couple this data with pointer-analysis, and infer stack boundaries.

4.3 Malloc Plugin

5 Surprises and Lessons Learned

There were a number of surprises and lessons learned throughout this project.

5.1 Reasoning about buffer overflows

An initial approach towards ensuring stack buffer overflow detection was to attempt to reason about sizes of buffers which are passed to "Potentially Dangerous Functions" [8] such as `strcpy`. This turned out to be an unfruitful approach for the scope of the project, since one would have to invariably have to reason about memory accesses.

A concrete example that illustrates the difficulty of performing checks on `strcpy` was found when we considered `libxkbfile.so.1.0.2`. Initially we thought that this is a candidate we could perform our check on, due to containing a `strcpy(buf,"none")` [2] call. However, this was found to compile to the `LDMIA.W` instructions mentioned before. This example principally highlights the difficulty in choosing candidates for testing.

One surprising result was the great number of statements (over 200) that some argument sites were found to depend on in `Coreutils` binaries. It was also the case that this result was exhibited inside large functions. When testing the plugin on small examples initially, the number of statements rarely numbered more than 20. As mentioned previously, we speculate that this will affect

5.2 General Things Learned

In general, this project made clear the difficulty of tackling binary problems. Specifically, we cannot advance the problem of analyzing binaries without basic “building blocks” which provide information about the security outcome we wish to address. For each such potential outcome in the future, we will have to consider critically what types of “building blocks” we need, whether pointer analysis, path-sensitivity, context-sensitivity, etc.

We learned that it is more effective in our research to develop our approaches with respect to security outcomes. By doing so, we incrementally identify the requirements for producing desired security outcomes (e.g., reaching definitions, use-def). It also seems to be a pattern that more sophisticated analyses are built on top of simpler analyses (like those produced in this project), rather than sophisticated analysis being stand-alone. This follows because we require more fundamental information before reasoning about further complexities.

6 Conclusions and Future Work

We will focus specifically on refining the sophistication of our analyses to handle broader cases as outlined in our discussions. A major benefit of identifying the limitations in our approch and evaluation is that it immediately highlights areas of refinement. The additional goals beyond 100% in our original proposal are all candidates for formulating future analyses. In summary, these are:

- Implement intraprocedural detection of null pointer dereferences
- Implement detection of dynamic memory errors e.g. dangling pointers, double frees, and invalid frees
- Implement intraprocedural detection of variables with uninitialized variables e.g. usage of variables with unassigned values
- Evaluate soundness, completeness, and performance of our tool against similar bug checking tools, e.g. Address Sanitizer, Valgrind MemCheck, Clang Static Analyzer, Coverity, CodeSonar, etc.

It has been a promising exercise to prototype and test our analyses within the context of this project. We consider the experience and insights gained throughout to be particularly valuable for performing continued research in our group.

7 Distribution of Total Credit

A Appendix

A.1 Technical Description of the BAP Dataflow framework

Due to the preliminary status of the current BAP framework, a number of common compiler analyses have not yet been implemented, including a basic dataflow framework. In particular, BAP currently only includes a disassembly engine using the LLVM framework, a lifter from ARM/x86/x86_64 to an intermediate language known as BIL (BAP Intermediate Language), and a basic semantic analysis for identifying basic blocks and performing a graph-based traversal through predecessors and successors.

As a result, we spent the first portion of our project implementing a basic dataflow analysis framework capable of performing both forwards and backwards dataflow analysis. This was accomplished by first implementing a sparse Bitvector module capable of representing both the infimum and supremem lattice elements (e.g. the empty and universal sets). Existing Bitvector representations in the standard library and Jane Streets Core library are not capable of this, because they utilize a fixed-size bit representation for the Bitvector. As a result, this would require a preliminary pass through each basic block and each instruction to identify the number of lattice elements in the Bitvector, reducing performance and flexibility of analysis.

In particular, our custom implementation defines a Bitvector record (datatype) composed of three subvalues: (1) infinite: a boolean value that determines whether the bitvector is of infinite size, (2) value: a boolean value that determines whether the bits of the bitvector are 0 or 1, and (3) indexes: a set of integers that identifies indexes of the bitvector that are set to value. As a result, this implementation allows for the empty and universal sets to be respectively identified by the following simple Bitvector values: {infinite: true, value: false (0), indexes: {}} and {infinite: true, value: true (1), indexes: {}}. Standard Bitvector operations, such as union, intersection, and difference, are defined using these primitives. As an example, consider an arbitrary Bitvector a, and the following operations: empty - a = empty, a - empty = a, universal - a = not a, a - universal = empty. These operations are straightforward, even for the relative complement of the universal set with the arbitrary Bitvector; this can be represented by simply toggling the value attribute from true to false or vice-versa.

Specifically, this highlights the idea that each finite Bitvector has two distinct representations: one with value set to true and indexes containing a set of bit indexes that are set, and another with value set to false and indexes containing a set of bit indexes that are unset. Although this does increase the logical complexity of the implementation, it allows for greater lattice representation flexibility with decreased memory consumption.

Otherwise, the dataflow framework implementation is relatively standardized, and is built on top of this lattice representation. A pair of maps are defined for translating unique instruction addresses of BIL or assembly instructions to lattice values, and vice-versa. This is achieved through the use of a mutable counter that identifies the unique Bitvector index associated with each instruction. Then, analysis results from each sequence of BIL instructions are propagated to each assembly instruction, and subsequently to each basic block, allowing for analysis granularity at all levels. Users of the framework specify initial values for each interior and boundary point, the entry block of analysis, and the direction of the dataflow analysis. Since the BAP framework is written in a functional language, the user can also pass a user-defined meet function and transfer function for performing the body of analysis. An internal worklist is used to keep track of basic blocks that needs to be scheduled for reanalysis; this is achieved by making the main dataflow analysis function recursive, and iterating through every basic block on the worklist before constructing a new worklist from those basic blocks with modified dataflow lattice values.

B Appendix

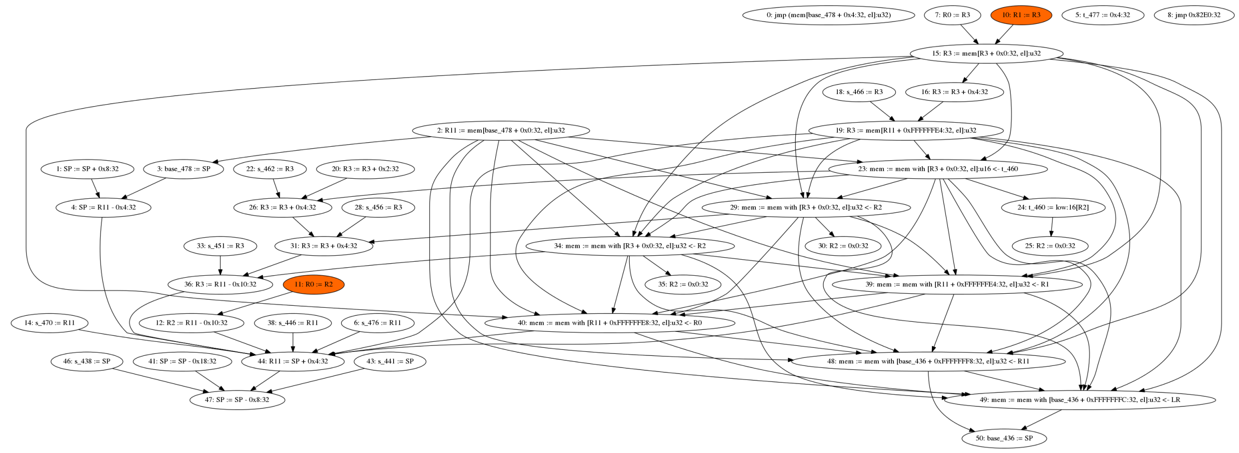
BIL Dissasembly of Listing 3

```
SP := SP - 0x8:32
mem := mem         with [base_436 + 0xFFFFFFF8:32, el]:u32 <- R11
mem := mem         with [base_436 + 0xFFFFFFFC:32, el]:u32 <- LR
base_436 := SP
R11 := SP + 0x4:32
t_439 := 0x4:32
s_438 := SP
SP := SP - 0x18:32
t_442 := 0x18:32
s_441 := SP
mem := mem         with [R11 + 0xFFFFFFE8:32, el]:u32 <- R0
mem := mem         with [R11 + 0xFFFFFFE4:32, el]:u32 <- R1
R3 := R11 - 0x10:32
t_447 := 0x10:32
s_446 := R11
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
R3 := R3 + 0x4:32
t_452 := 0x4:32
s_451 := R3
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u32 <- R2
R3 := R3 + 0x4:32
t_457 := 0x4:32
s_456 := R3
R2 := 0x0:32
mem := mem         with [R3 + 0x0:32, el]:u16 <- t_460
t_460 := low:16[R2]
R3 := R3 + 0x2:32
t_463 := 0x2:32
s_462 := R3
R3 := mem[R11 + 0xFFFFFFE4:32, el]:u32
R3 := R3 + 0x4:32
t_467 := 0x4:32
s_466 := R3
R3 := mem[R3 + 0x0:32, el]:u32
R2 := R11 - 0x10:32
t_471 := 0x10:32
s_470 := R11
```

```
R0 := R2
R1 := R3
jmp 0x82E0:32
LR := 0x849C:32
```

BIL data dependency instructions for `argv[1]`

```
<(0x8444:32, 0)> base_436 := SP
<(0x8444:32, 1)> mem := mem        with [base_436 + 0xFFFFFFFC:32, el]:u32 <- LR
<(0x8444:32, 2)> mem := mem        with [base_436 + 0xFFFFFFF8:32, el]:u32 <- R11
<(0x8444:32, 3)> SP := SP - 0x8:32
<(0x8448:32, 2)> R11 := SP + 0x4:32
<(0x8450:32, 0)> mem := mem        with [R11 + 0xFFFFFFE8:32, el]:u32 <- R0
<(0x8454:32, 0)> mem := mem        with [R11 + 0xFFFFFFE4:32, el]:u32 <- R1
<(0x8458:32, 2)> R3 := R11 - 0x10:32
<(0x845C:32, 0)> R2 := 0x0:32
<(0x8460:32, 0)> mem := mem        with [R3 + 0x0:32, el]:u32 <- R2
<(0x8464:32, 2)> R3 := R3 + 0x4:32
<(0x8468:32, 0)> R2 := 0x0:32
<(0x846C:32, 0)> mem := mem        with [R3 + 0x0:32, el]:u32 <- R2
<(0x8470:32, 2)> R3 := R3 + 0x4:32
<(0x8474:32, 0)> R2 := 0x0:32
<(0x8478:32, 0)> t_460 := low:16[R2]
<(0x8478:32, 1)> mem := mem        with [R3 + 0x0:32, el]:u16 <- t_460
<(0x8480:32, 0)> R3 := mem[R11 + 0xFFFFFFE4:32, el]:u32
<(0x8484:32, 2)> R3 := R3 + 0x4:32
<(0x8488:32, 0)> R3 := mem[R3 + 0x0:32, el]:u32
```



Graphical output of BIL data dependencies, with highlighted arguments

References

[1] Binary analysis platform. `https://github.com/BinaryAnalysisPlatform/bap`. Accessed 03/16/2015.

[2] xorg-server. `https://github.com/wereHamster/xorg-server/blob/server-side-xcb/xkb/xkbtext.c`.

[3] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: a binary analysis platform. In *Computer aided verification* (2011), Springer, pp. 463–469.

[4] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c. In *Software Engineering and Formal Methods*. Springer, 2012, pp. 233–247.

[5] Engler, D. Finding bugs with system-specific static analysis. `http://web.stanford.edu/~engler/paste02-talk.pdf`. Accessed 03/16/2015.

[6] Hallem, S., Chelf, B., Xie, Y., and Engler, D. *A system and language for building system-specific, static analyses*, vol. 37. ACM, 2002.

[7] Kinder, J., and Veith, H. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification* (2008), Springer, pp. 423–427.

[8] Seacord, R. C. *The CERT C secure coding standard*. Pearson Education, 2008.