A Seminar Report

on

Improving BAP to Help Analyze COTS Software for Security Vulnerabilities, Accurately and Efficiently

Submitted in requirement for the course
TRAINING AND SEMINAR (CSN-499)
of Bachelor in Computer Science and Engineering

by

Jay Hitesh Bosamiya

(Enr No: 13114024)

Project under guidance of

Prof. David Brumley

Director, CyLab; Professor, Electrical and Computer Engineering

Carnegie Mellon University (CMU)

Pittsburgh, Pennsylvania, USA

2^{nd} May 2016 - 15^{th} July 2016

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE
ROORKEE- 247667 (INDIA)

Contents

1 Introduction

In today's highly interconnected world, it is not surprising to find such a vast number of platforms for software, running on a myriad of different architectures; and it is quite plausible that these numbers are only bound to increase. With all the instances of "hacking" going on, it becomes essential now, for software security researchers to come up with better techniques to find and patch vulnerabilities in software.

However, only a small fraction of software can be handled by traditional techniques, which require that the source of the software be available. Also, due to the fact that many software now exist of new, or even proprietary architectures, it becomes even more difficult to be able to analyze all software. Hence, many techniques and platforms have been proposed to do this. CMU's own offering comes in the form of a software known as Binary Analysis Platform (BAP).

The platform can perform many different and complex analyses automatically on software in a platform and architecture agnostic way, through a process of semantic reasoning about the code. However, even with all of this, most of the vulnerabilities found today, are found by human ingenuity.

This projects aims to bridge the gap between human intuition and machine accuracy to create a human-in-the-loop system which can quickly, and accurately, analyze common-off-the-shelf (COTS) software.

Additionally, this project aims to reduce the effort required to write an analysis, or extend BAP, thereby promoting more experimentation and research into finding better analyses.

As an ambitious long term goal (longer than the scope of the internship), the project also aims to come up with new viable ideas to get rid of certain classes of bugs/vulnerabilities completely from the world's software.

The rest of this report is organized as follows: Section 2 details the motivation behind the choice of this project. Objectives for the project are sketched in Section 3. A plan of work is then described in Section 4, followed by the description of the work in Section 5. The implementation details are explained in Section 6, and Section 7 has results and discussion. Finally, conclusions are drawn in Section 8, followed by references and acknowledgements in Sections 9 and 10 respectively.

2 Motivation for the Project

Analyzing binary (i.e. executable) code is an extremely important goal in security research and software analysis. A lot of the world's software is either closed source or is hidden away in various proprietary devices with differing architectures (such as x86, MIPS, ARM, x86-64, etc.). Advances in IoT devices further stress the importance of being able to understand and analyze binary code. Common Off The Shelf (COTS) software is know to be riddled with bugs and vulnerabilities that are frequently used for malicious purposes, making it extremely crucial that vendors are able to quickly find and patch their software.

However, much of the analysis that must be done is done by vulnerability researchers not having access to the code, and requires many hours of manually going through low-level machine code. Automating the task of analysing the code would be a large step in the direction of being able to rid the world of bugs and vulnerabilities from software, but this is a hard task. Since machine level code does not have the high level types or constructs that languages like C, Java, C++, etc. have, even trivial analyses that could be done on high level code (such as ensuring that `malloc`s are checked before being used) cannot be analysed easily from only the binary code.

Many techniques have been proposed to achieve this, such as BitBlaze [1], Jakstab [2], BAP [3] etc. One commonality that can be seen in them is that instructions are first lifted to an Intermediate Representation (IR) which is then analysed for certain properties in an automatic fashion to see whether certain properties are violated or not. However, they fail to take into account the ingenuity of the human mind, which tends to notice flaws/vulnerabilities much faster than a machine can.

Hence, it becomes imperative to come up with ways to interactively do analyses, in a way that leverages human intuition and machine precision to find security vulnerabilities in COTS software in a more efficient and accurate fashion. Also, it is essential that when a researcher wants to write a new analysis, he/she should be able to do so with the least effort required. These two reasons motivate the work done in the project.

3 Objectives of Project

In order to reach the goal of improving analysis capabilities in BAP to find security vulnerabilities in COTS software, it is essential that a user is able to

1. Write analyses in an efficient and re-usable manner
2. Easily visualize results of pre-existing analyses
3. Work with well established industry standard software, along with cutting edge research analyses
4. Interactively edit and guide analyses in directions that might be more suitable/feasible, due to intuitive domain knowledge
5. Effortlessly port analyses in one architecture/OS/system/etc. into another
6. Reproduce results found during analysis

In order to be able to handle all the above mentioned points, the objectives of the project were to

1. Integrate the industry-standard IDA Pro [4] with the cutting-edge research analyses done in Binary Analysis Platform (BAP) [3]
2. Provide a visual aid to certain commonly used analyses
3. Use the Hex-Rays decompiler plugin [5] to simplify ease of understanding of information flowing from BAP into IDA Pro
4. Provide for a "BAP View" in IDA Pro which allows arbitrary execution traces of BAP to be interacted with from IDA
5. Create an easy to use interface in BAP to write configurations of plugins
6. Propagate rooting information to aid in easier manual analysis by using Byteweight [6] through BAP

4 Work Plan to Meet the Objectives

Since the project revolved around the core idea of improving human-in-the-loop analysis of binary software, which can leverage both the capabilities of the human mind, as well as the accurate repetitive checking capabilities of machines, this significantly reduces complexity and time taken to find vulnerabilties, and ensures that researchers can concentrate on the actual bug-finding. In order to do this, initially a set of plugins were to be built that would be able to integrate the Binary Analysis Platform (BAP) [3], with the industry-standard IDA Pro [4] in order to be able to propagate taint information in the code, visually. By integrating this with the Hex-Rays decompiler plugin in IDA Pro, it would be possible to perform program slicing of the binary, at a visual level. Thereafter, I introduced BAP Intermediate Represantation (BIR) tags were to be introduced to IDA Pro, which would allow for all tagging based analysis in BAP to propagate to the interactivity of IDA. Further on, a system had to be designed to further integrate BAP with IDA, allowing user-edited analysis results from IDA into BAP. This would allow user to easily leverage the work they normally do in IDA, to aid in speeding up and making BAP analyses more accurate. By also introducing a "BAP View" into IDA, it would further accelerate the vulnerability finding process. One of the common issues faced while writing analysis plugins in BAP, is that configuration of such plugins was extremely complicated and convoluted to write, and so, the next focus would be on improving the configuration ability, thereby allowing analyses to be written effortlessly, with all the hard work being hidden behind the right abstractions. Alongside all of this, there was also focus on looking into ideas for analyses, and for efficient fuzzing (dynamic testing of code) in an architecture-independent fashion. These are questions that were aimed at a much longer time-frame, to extend beyond the period of the internship.

5 Work Description

BAP [3] is a platform used for Binary Analysis. It is written in OCaml, which is an industrial strength programming language supporting functional, imperative and object-oriented styles [7]. Despite BAP being used to analyze binaries, which are inherently imperative, its own written in a strongly functional way, with a very strong type system. It has a layered architecture consisting of four layers. Although the layers are not really observable from outside of the library, a large part of my work involved understanding the inner workings of the library, along with the workings of the different plugins in BAP.

1. The Foundation library defines BAP Instruction language data types, as well as other useful data structures, like Value, Trie, Vector, Graph, etc.
2. The Memory model layer is responsible for loading and parsing binary objects and representing them in computer memory. It also defines a few useful data structures that are used extensively by later layers, like Table and Memmap.
3. The Disassembly layer performs disassembly and lifting to BAP Intermediate Language (BIL).
4. The semantic analysis layer transforms a binary into an IR representation, that is suitable for writing analysis.

While these 4 layers form the basis of the library, BAP itself consists of a plugin architecture. Currently, there exist 3 extension points to the library:

1. Loaders - to add new binary object loaders
2. Disassemblers - to add new disassemblers
3. Program Analysis - to write analysis

The last category of these plugins is the most widely used, and is where a large portion of focus is directed by most researchers. A select few of BAP's plugins include

1. arm - provide ARM lifter
2. byteweight – find function starts using Byteweight algorithm
3. map-terms – map terms using BML DSL
4. warn-unused – warn about unused results of certain functions

5. ida – use IDA to provide rooter, symbolizer and reconstructor
6. propagate-taint – propagate taints through a program
7. etc.

Alongside this, there exists the industry-standard IDA Pro [4], along with its own Hex-Rays decompiler plugin [5]. IDA Pro is used by almost all security researchers across the world to perform reverse engineering on binaries, aided by its ability to do Interactive Disassembly (which leads to its name, Interactive DisAssembler – IDA). It has a plugin system of its own, which allows to write plugins in a language called IDC, which is defined by IDA. With some restrictions, it also allows to write plugins in a dialect of Python, known as IDAPython. Its GUI is written in the Qt framework, and can be hooked into using its SDK and API, to perform complex tasks to aid binary analysis.

By leveraging the features that are inbuilt in BAP and IDA Pro, and enhancing their interfaces further with custom written code, it becomes possible to implement the modules presented in subsections 5.1 and 5.2.

5.1 Modules of the Project

The whole project can be split into a hierarchy of different modules and sub-modules (for easier understanding of work done). Note that this hierarchy does not strictly define the order of work, since a lot of the work needed to be done in parallel, or across different branches of the hierarchy in sequence. Also, this hierarchy is quite deep, since OCaml encourages the use of small, easy-to-understand, nested modules.

1. IDA Integration
    (a) Plugins
        i. Taint propagation
            A. To Text/Graph View
            B. To Pseudocode/Decompiler View
        ii. BAP Intermediate Representation (BIR) Attribute Tagging
            A. To Text/Graph View
            B. To Pseudocode/Decompiler View
            C. Allow arbitrary BAP Execution
        iii. Symbol Propagation from IDA to BAP
        iv. Type information Propagation from IDA to BAP
        v. BAP View in IDA Pro

vi. Rooter Propagation from BAP to IDA

vii. Brancher Propagation from IDA to BAP

(b) Utilities

i. Abstract IDA Plugins

ii. S-Expression Library

iii. IDA Comment Manipulation Library

iv. ini-config Manipulation Library

v. IDA Information Dumping Library

A. Dump loader information

B. Dump symbol information

C. Dump type information as a C Header

D. Dump brancher information

vi. IDA Integration Library

A. Creation of hotkeys mapping to actions

B. Conversion of functions to equivalent decompiled functions

C. All EA (effective address) enumerator

vii. BAP Integration Library

A. Analyzing and finding BAP installation location at runtime

B. Passing along all requisite information from IDA into BAP directly at runtime

2. BAP Enhancement

(a) Emit IDA script code to export information from BAP into IDA

(b) IDA interaction, via `Ida` module interface

(c) Improve ARM Lifter

(d) Improve Continuous Integration (CI) Testing Framework

(e) Allow disabling certain plugins from compiling (rather than the pre-existing "compile-everything" strategy used).

(f) Allow runtime configuration of plugins

i. Introduction of new `Config` interface

ii. Refactor of all plugins to the new interface

(g) Enable caching of pre-built dependencies

(h) Usage of `future`s to ensure correct ordering in certain evaluation schemes

(i) Allow runtime configuration of frontends

i. Introduction of new `Frontend.Config` interface

ii. Refactor of all frontends to the new interface

3. Analyses that are currently under work (i.e. partially worked on during the internship, and research on these also extends beyond the duration of the internship)

   (a) Using Saluki [8] to model complex real-world vulnerabilities, and using it to find bugs in COTS Software

   (b) Modeling buffer overflow [9] conditions and analyzing them on COTS Software

   (c) Large scale, cloud-based distributed fuzzing, under minimilastic knowledge assumptions on the binaries

   (d) Modeling Use After Free (UAF) vulnerabilities in COTS Software, and writing analyses to quickly identify and correct for them.

5.2 Detailed Description of the Individual Modules

5.2.1 IDA Integration

BAP, as a standalone application, cannot achieve any sort of interactivity, but is extremely good at performing semantic understanding of binaries, and application of analyses on this semantic layer. On the other hand, IDA Pro is an application that is interactive, but can only perform actions and understanding at a syntactic layer. The motive of IDA Integration was to effectively marry these two disparate systems into one coherent whole, using elegant abstractions, so that each of them could be swapped out at any point, if a better implementation comes along.

The different plugins and utilities implemented in this part revolve around either (i) passing information from IDA to BAP or (ii) passing information from BAP to IDA.

Additionally, by using the Hex-Rays decompiler plugin [5], and hooking directly into the Qt GUI implementation of IDA, it becomes possible to propagate all information from the Disassembly View (which is called either Text View, or Graph View in IDA), into the Decompiler View (which is called the Pseudocode View).

Figure 1 displays an executable on which taint analysis has been done, where IDA Pro hooks into BAP to propagate taint information from a user specified source (marked in yellow), and displays parts of code affected tainted by this controlled register (marked in red). An equivalent taint analysis is shown in the decompiled output, in the Pseudocode View in Figure 2.

Allowing arbitrary BAP execution directly from IDA Pro allows for a whole new range of possibilities, such as conducting complex analyses like Saluki [8] and have the results displayed directly in IDA, from where further manual analysis can be performed. Figure 3

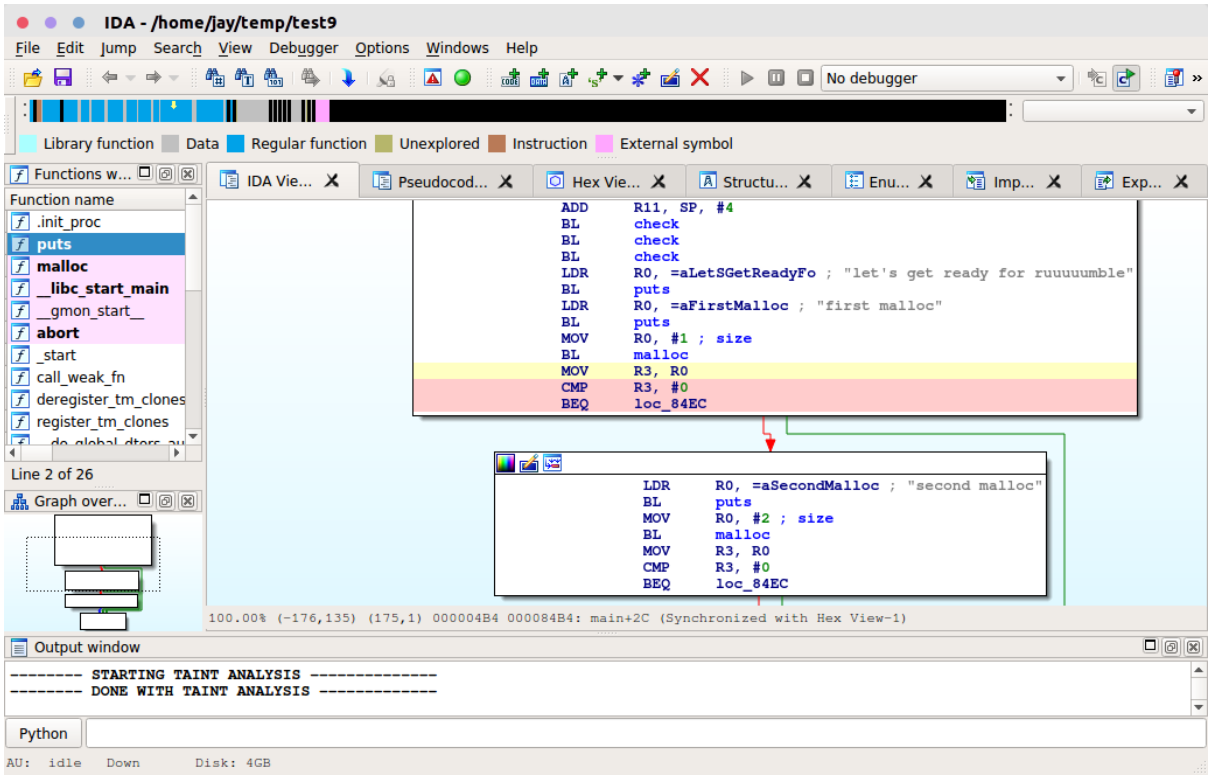

Figure 1: Taint Analysis in Graph View

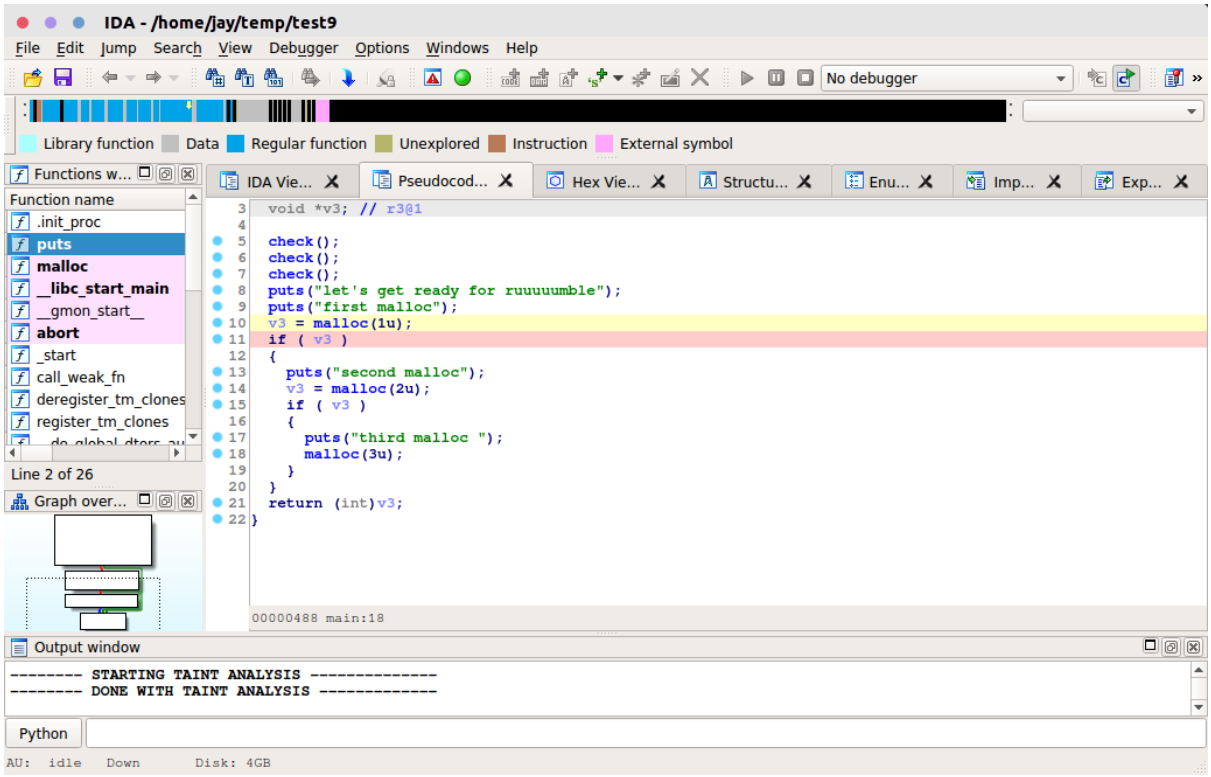

Figure 2: Taint Analysis in Pseudocode View

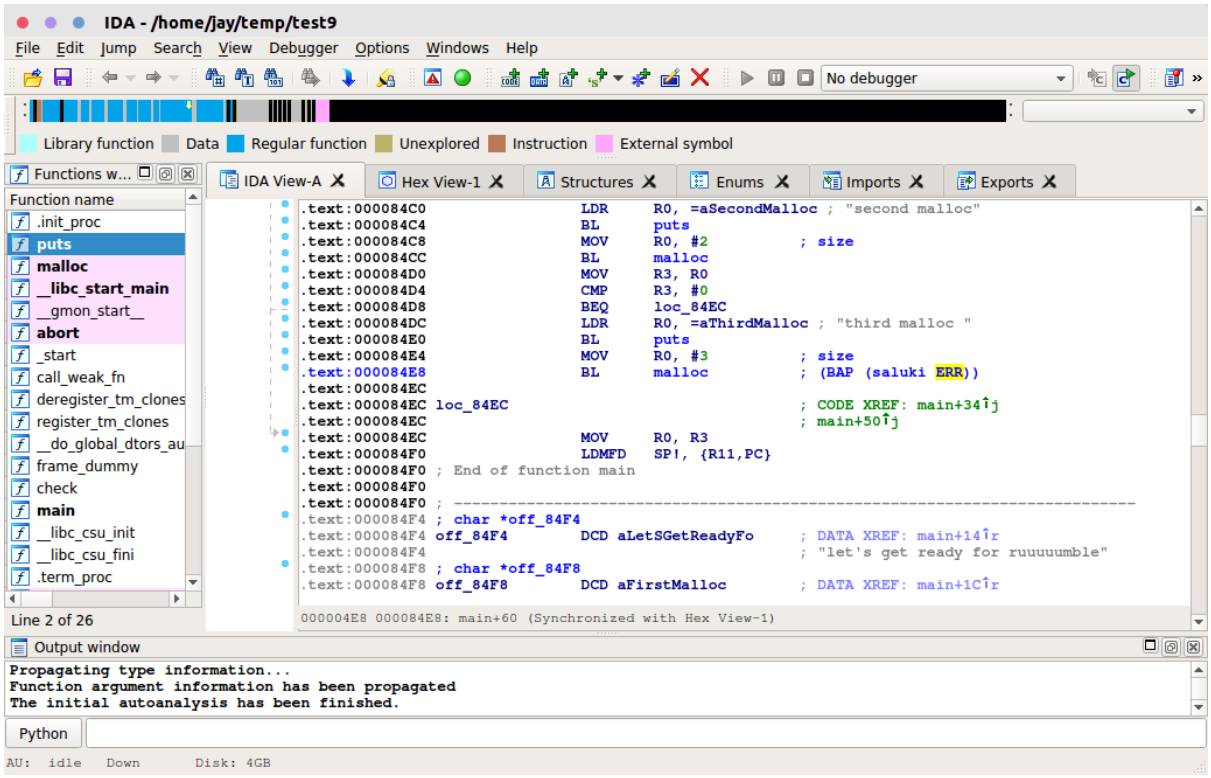

Figure 3: Transfer of BIR Attributes from Saluki into Text View

displays a `malloc` which is recognized to have errored since its output is unchecked. Figure 4 shows the same in the decompiler output. Note that all tagged attributes are shown in the form of S-Expressions, which are syntactically equivalent to LISP expressions, and thus have extremely large expressive power.

Taking into account that arbitrary BAP executions could be executed from IDA, it becomes necessary that output from BAP can be analyzed in an easy way. Also, this allows for examination of BAP Intermediate Language (BIL) and BAP Intermediate Representation (BIR) directly from IDA Pro, in addition to the already existing disassembly features in IDA. Figure 5 shows the BIR output of a sample program through BAP View, through a sample execution.

While most of the above mentioned features fall into the category of BAP to IDA information flow, they strongly use IDA to BAP flow internally, where all information that the user interactively edits in IDA is automatically propagated to BAP, in order to inform its analyses to be more accurate. For example, if a user (after manual analysis) is able to determine that a function is equivalent to a `calloc`, then the user only needs to rename the function in IDA, and the information automatically propagates to BAP.

Additionally, BAP has an extremely good "rooter", which refers to an analyzer which can find function starts. This rooter, called Byteweight [6], significantly outperforms the state-of-the-art (including IDA Pro), and by leveraging this information, it is possible now, to have function starts automatically be marked more accurately in IDA Pro, aiding

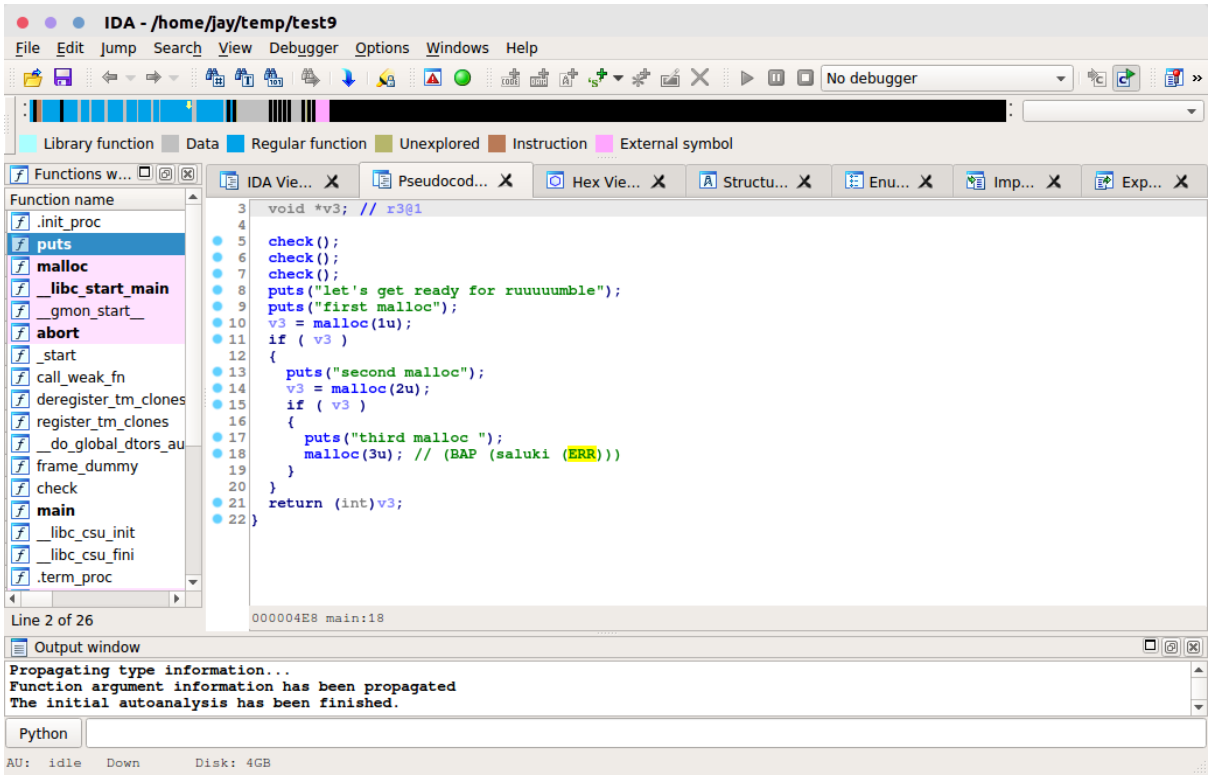

Figure 4: Transfer of BIR Attributes from Saluki into Pseudocode View

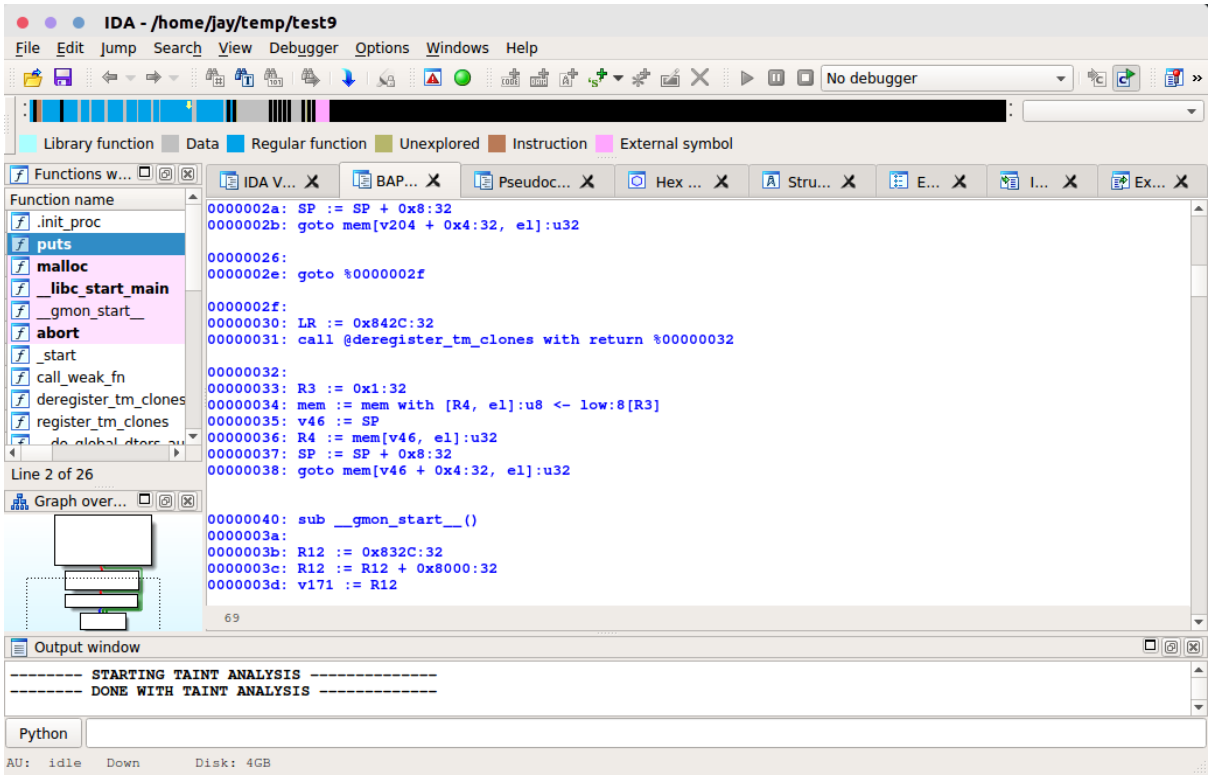

Figure 5: BAP View showing BIR output of a program

faster analysis.

While BAP is extremely good at semantic analysis, IDA has years of experience in heuristics which allow it to very accurately predict destinations of indirect jumps (such as `jmp eax`). This functionality, known as the "brancher", is now exported for use from IDA into BAP.

5.2.2 BAP Enhancement

BAP consists of 3 main portions: (i) `Bap.Std` library, (ii) frontends, and (iii) plugins. The project work involved changes in all 3 of these very important portions of BAP.

The first of these was to write an `emit-ida-script` plugin, which would allow for generating a script file after BAP analyses, to be able to import information into IDA Pro, either on the same system, or on a completely different system; aiding in collaborative vulnerability research and analysis.

The previously existing `Ida` interface, which provided ways to call and execute IDA scripts from BAP, was initially written as a hack, and violated properties of idempotency of library functions. Hence, after a thorough redesign of this, it was decided to have the library function that would be provided for via a plugin. This was implemented using OCaml's type system to create an elegant dependency inversion.

BAP is able to understand binaries at a semantic level because it is able to lift code from an architecture dependent assembly code, to an architecture independent Intermediate Representation (BIR). This process is achieved by plugins called lifters. One of these plugins, responsible for lifting ARM instructions, was unable to handle certain types of instructions, which were then rectified, and taken care of for proper lifting.

Since BAP is built as an open source software, with collaboration done via PRs on GitHub (`https://github.com/BinaryAnalysisPlatform/bap`), it becomes essential to have a system for Continuous Integration (CI) Testing, to ensure that the build never has any regressions. For this, we employ the widely used Travis CI infrastructure. However, there are some caveats when using Travis CI with OCaml, and OPAM (the package manager for OCaml), which might sometimes (rarely) lead to erroneous tests or long testing times. By introducing a constraint solver, `aspcud`, the OPAM caveats were taken care of. Additionally, by writing a custom caching system, build and test times were reduced by over 25% of previous values for the same.

BAP, like many other open source software, follows the commonly used idiom of installation via `./configure; make; make install`. Since BAP consists of many pieces that can work together, it should be possible to compile only a sub-section of BAP, and disable the rest. Earlier, this was unable to be done, and instead everything would compile. However, as part of the project, this feature was added, in order to be able to enable and disable features at configuration time.

Most plugins in BAP used to use `Cmdliner` in order to take command line arguments and this was implemented mainly as a "hack". After much redesigning of an abstraction on top of this, it became possible to implement a new `Config` interface. By applying the concepts of Monads [10], Variadics [11], Applicatives [12] etc, it was possible to design an extremely elegant abstraction around the implementation of the feature. With this new interface, it was possible to refactor all of the plugins to be much more simpler to read, write, understand, and manage.

Another extremely elegant part of BAP, is its `Future` library. This allows for the creation of the concept of a "variable" that will know its "value" at some arbitrary (or non-existent) time in the future. This, combined with the extremely strong algebraic typing system that is present in OCaml, allows for specification of complex orderings, and seemingly concurrent callbacks to be specified as just reactions to the events. This was used to simplify the code in many different parts of the implementation of BAP.

After there was a `Config` interface for plugins, there came the obvious question of such an interface for frontends, of which BAP has a few, namely `bap`, `bap-mc`, `bap-byteweight`, `bap-fsi-benchmark`, etc. However, implementation of `Config` for frontends is significantly more complex than for plugins, due to the additional capabilities that they need to be given, such as allowing choosing which plugins to load, at runtime. Hence, a new interface, namely `Frontend.Config` was introduced, that overrides and increases the capabilities of the `Config` interface, but may only be used by frontends. Thereafter, all frontends were refactored to use this interface.

5.2.3 Analyses Ideas

Many different ideas for analysis were constantly being discussed and refined throughout the internship, along with small prototypes being made to prove/disprove certain hypotheses.

Some of these ideas, such as Saluki [8] were already being worked upon, well before my internship, and have already submitted to top-tier conferences. Other ideas, such as modelling buffer overflows, use-after-free (UAF) vulnerabilities, etc are still being researched.

One other, seemingly unrelated, but strongly correlated field of research is in the direction of large scale, cloud-based distributed fuzzing. In this scenario, we try to minimize the number of knowledge assumptions we make about the binaries. This would lead to an unprecedentedly large potential in checking the world's software (such as in proprietary/outdated routers, custom IoT devices, etc.). The state-of-the art in this area is still far from capable of doing such an ambitious task, and thus further research is required in this area.

6 Implementation Details

BAP, as mentioned in an earlier section, is implemented in OCaml [7], which is a strongly typed functional programming language, that also allows for imperative and object oriented programming. What this means, is that the fully algebraic type system can be used to catch many programming bugs at a very early stage, and one can guarantee that certain types of errors are impossible to occur in the implementation. This makes it perfect for making a rock-solid library upon which analyses can be built.

As for IDA Pro, most of its code (though it is a mature, industry-standard software), tends to change version to version, and it requires careful perusal of the API and SDK documentation to notice minor differences. However, by implementing scripts in IDAPython, it is possible to ensure stability at least over versions spanning across a time frame of a few years. Hence, the language of choice for implementing IDA Plugins was chosen to be IDAPython.

6.1 Experimental Setup

The development environment for the project was chosen to be a laptop (Intel i7, 8GB RAM) running a customized Ubuntu 14.04 Trusty Tahr, with OCaml v4.02.3. Code was written in the Emacs editor, which has extremely good support for OCaml via its Tuareg mode, along with Merlin.

The IDA Pro versions that all of the changes were tested on were: 6.6, 6.7, 6.9, and IDA Demo.

All of the code that was implemented during the duration of the project is accessible on GitHub at `https://github.com/BinaryAnalysisPlatform/bap`, and at the `https://github.com/BinaryAnalysisPlatform/bap-ida-python` repository.

All of the code is rigorously checked via Travis CI, as well as manual code-review; alongside regular meetings with all members working on BAP.

6.2 Datasets Used

In order to test the different analyses on executables, all tests were run on the `coreutils` package, compiled onto the following architectures: (i) ARM, (ii) x86, and (iii) x86-64. In addition to this, many custom written executables were written to stress certain edge cases and add them to the test-suite. Also, challenges from Capture-The-Flag (CTF) contests were also used to provide insight into further areas of research/improvement, since these demonstrate specific vulnerabilities and can be used to analyze things in a "concentrated" fashion.

7 Results and Discussions

As a result of this project, many large changes were made to BAP, leading to a speedier advance in the goal of making the world's software more secure. With these changes, researchers can now focus on writing better analyses, and also be able to work with analyses interactively, in a completely platform and architecture agnostic way. Additionally, Common Off The Shelf (COTS) Software can now be checked with greater ease and precision.

A large part of the changes made in this project have already been used to demonstrate the superior capabilities in vulnerability research, in project working with the U.S. Department of Defense, at Defense Advanced Research Projects Agency (DARPA).

The interactivity introduced by these changes also led to the detection of certain edge cases in already existing analyses, which can now detect vulnerabilities with much greater accuracy.

Also, the change introduced by the new interfaces added to the `Bap.Std` library, namely `Config` and `Frontend.Config` surfaced pre-existing issues that were previously uncovered by testing.

Additionally, by the changes introduced by the project, two of the leading analysis systems, IDA Pro (which looks from a syntactic perspective, using heuristics), and BAP (which looks from a semantic perspective, using precise/provable formulations), can now work in synergy, in order to produce novel results.

Furthermore, with the new ideas that have been generated for analyses, it becomes possible to work towards an overarching system, that can (in the possibly near) future, hope to get rid of certain types of bugs completely from software, both open- and closed-source.

8 Conclusions

It is found that the analysis of binary code is an extremely hard task, which cannot be done individually either by machine, or by humans, both efficiently and accurately. Hence, it was realized that a combination of the two was essential to good analysis. Also, it needs to be easy to write new analyses, if and when need be. In order to check the world's software for security issues, it becomes extremely necessary for this challenge to be solved, and this project aimed, and achieved progress towards this goal. Further research in this field can definitely still be done, and is under way (as a continuation of the ideas in this project).

9 References

[1] Dawn Song, David Brumley, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *In Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[2] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *in CAV, ser. LNCS*, pages 423–427. Springer, 2008.

[3] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the Conference on Computer Aided Verification*, 2011.

[4] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/.

[5] Hex-Rays. Hex-Rays Decompiler Plugin for IDA Pro. https://www.hex-rays.com/products/decompiler/.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, San Diego, CA, August 2014. USENIX Association.

[7] OCaml. An industrial strength programming language supporting functional, imperative and object-oriented styles. https://ocaml.org/.

[8] BAP Developers. Saluki - A Vulnerability Patterning and Searching Framework. https://github.com/BinaryAnalysisPlatform/bap-plugins/tree/master/saluki.

[9] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[10] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991.

[11] Douglas Gregor, Jaakko Jrvi, and Gary Powell. Variadic templates, 2004.

[12] Artem Alimarine, Sjaak Smetsers, Arjen Weelden, Marko Eekelen, and Rinus Plasmeijer. Applicative (functional) programming.

10 Acknowledgements

I would sincerely like to thank, and express my deep sense of gratitude to my supervisor, Prof. David Brumley (Director, CyLab; Professor, Electrical and Computer Engineering) for his guidance and support throughout the internship. Both him and his team have been extremely helpful, approachable, and friendly; and have my utmost respect and admiration.

I would also like to thank Ivan Gotovchits (Research Programmer, CyLab; Lead Developer, Binary Analysis Platform), without whose guidance, I might have not been able to get accustomed to the exotically beautiful and elegant language of OCaml. His insights into the perfect abstractions for any given scenario are unparalleled by anyone else I have had the opportunity to work with.

Additionally, I would also like to thank Rijnard van Tonder, Matthew Maurer, and Tiffany Bao, (Ph.D. students, supervised by Prof. Brumley) who helped out the most in the first few days, when I was getting accustomed to both the code-base, as well as the culture at CMU. They were a constant source of knowledge and inspiration throughout the internship.

Also, I would like to thank Jacob Parker (another intern under Prof. Brumley), for his changes to the code base which led to improvements in compatibility with newer versions of LLVM.

I also would like to thank Maverick Woo (Systems Scientist, CyLab) who was always there whenever I needed guidance, access to some software or hardware resource, or books to get specific details on things.

Last but definitely not the least, I would like to thank Ms. Tina Yankovich, Ms. Toni Fox, and Ms. Brittany Frost, for helping me throughout all the confusingly complicated formalities, and money matters, that get involved when doing an internship in a foreign country.

I hope to continue working as a collaborator with the members of CyLab, and use the knowledge and experience that I have gained, to make more valuable contributions to the field in the future.