ICE: Binary Analysis That You Can See

Dean Pucsek, Jennifer Baldwin, Laura MacLeod, Celina Berg, and Yvonne Coady Department of Computer Science University of Victoria, Victoria, Canada Martin Salois

Defence Research and Development Canada (DRDC) Valcartier

Quebec City, Canada

Abstract—Tools for high-level languages often assist developers in successfully comprehending complex systems without worrying about low-level details. However, new architectures and paradigms now pose new challenges in program comprehension that often require high-level reasoning about low-level issues—sometimes even at the level of processor instructions. This is particularly true for the new generation of developers learning to harness the power of SIMD operations, multi-core, multi-processor systems. Though industrial-strength tools for malware analysts are available, these typically come at considerable cost and require extensive expertise.

Our proposed solution is to extend high-level comprehension tools, commonly available in IDEs, to low-level representations. This paper presents the design and prototype implementation of an Integrated Comprehension Environment (ICE), which provides an Eclipse-based tool suite extended to analyse code in intermediate and assembly languages. Preliminary evaluation based on visualisations for wayfinding, call graphs, sequence diagrams and control flow show, (1) correspondence to requirements for comprehension tools in this domain, (2) flexibility in the spectrum of data sources it can accept, and (3) scalability with respect to the explosion of instructions in the code base, while still providing a means to build new visualisations for analysis.

I. Introduction

With the growing number of new computing paradigms, new processor features, and the now ubiquitous nature of computing, developers not only need to be able to reason about the details of how their code is represented in high-level languages, but also how it will interact with the hardware when it runs. The problem is that it can be hard to recognise the implications of the final translation from the original source code to an executable binary. Additionally, comprehension tools have typically focused on the question of how high-level concerns are represented, leaving only specialised tools for malware analysis and reverse engineering to focus on its final, low-level state.

A previous study designed to elicit requirements from groups of developers and analysts that work with assembly code on a regular basis revealed many common challenges in this domain [1]. As a proof-of-concept, this work focuses on the following subset of challenges identified by the security analysts in that study:

- 1) *Multiple Executables:* Their disassembler tool cannot disassemble more than one executable file at a time (e.g. DLL libraries) and link between them.
- 2) *Map of Analysis:* It is easy to get lost when going deeper into the code—hard to track where the exploration started and how a deeper point was arrived at.

3) Cross Reference Mechanism: Lack of a cross reference mechanism between a given function in an executable file to a DLL.

This paper is organised as follows. Background and an overview of related tools and techniques are presented in Section II. Section III introduces the design of our prototype solution followed by a description of its implementation in Section IV. Section V provides an end-to-end example of working with our prototype, along with an evaluation of the costs and benefits in Section VI. Finally, Section VII presents ideas for future work and summarizes our contribution.

II. BACKGROUND AND RELATED WORK

Given the importance of binary analysis, it is no surprise that highly popular commercial products offer a wide variety of specialized visualization tools for experts. In addition to those, there are several academic and open-source initiatives.

IDA Pro [2] is the industry-standard interactive disassembler. It has been designed to integrate a suite of built-in analysis tools with those provided by third-parties in order to provide an extensible general-purpose binary analysis framework. IDA Pro provides this functionality through a traditional plugin architecture and a well-defined API that allows third-party developers to produce task-specific analysis algorithms, including our own sequence diagram tool [3]. The primary strength of IDA Pro is its disassembler which supports a multitude of instruction set architectures (ISAs).

BitBlaze [4] is an open-source project consisting of three integrated binary analysis frameworks (one static, one dynamic, and one hybrid), for 32-bit x86 binaries. All tools produce textual output only, and are dependent on an intermediate language generated using several third-party tools including an emulator called QEMU [5], [6]. A spinoff of BitBlaze, BAP (Binary Analysis Platform) [7], includes a feature to allow comments with readable assembly code to be attached to a sequence of instructions.

LLVM [8] is an open-source project that was initially designed as a framework for compiler construction but has now evolved to provide a collection of tools and libraries including debuggers, disassemblers, and high-level language parsers. It is built on an Intermediate Representation (IR), which provides not only a fully-specified ISA, but also many other functions and datatypes that might be useful to developing a binary analysis framework.

There are also tools that have been developed for the analysis of low-level assembly through visualisation and navigation support. In terms of navigation, MapUI [9] leverages human spatial memory and represents functions as 'countries' on a map. The size of the 'country' relates to the number of lines of code in the function and the relationship to other functions is represented with a 'shared border' between those 'countries'. Similarly, tools like Code Thumbnails [10] provide a spatial representation of a codebase, but with a 'thumbnail' sized version of the source files as opposed to the less explicit 'country' shapes used in MapUI. While these representations provide a high-level view of the codebase, other tools support analysis of control flow by drilling down to the sequence of function calls [11]. Code Bubbles [12] also provides a control flow visualisation where methods can be represented by a 'bubble'. The interface is also interactive, such that when a user selects a static call connection the corresponding 'bubble' is made.

III. DESIGN

The design of ICE leverages the *Model-View-Controller* (MVC) [13] design pattern in which information from any Executable Entity, a binary or intermediate language representing assembly code, is stored in an extensible data model and visualisations act as views of that data model describing the Executable Entity being analysed.

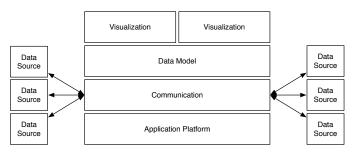


Fig. 1. High-level design of ICE

A schematic representation of the design of ICE is shown in Figure 1. The lowest layer, the *Application Platform*, is responsible for providing all aspects of a graphical user interface and application. This layer includes functionality such as window management, delivering mouse and keyboard events, and a minimal environment for the creation of an application.

Above the Application Platform is the *Communication* layer. The Communication layer enables bi-directional communication between ICE and external applications. Within ICE the external applications are referred to as *Data Sources* and ICE is known as a *Data Sink*. Although this terminology implies that information only flows from sources to the sink, ICE is capable of pushing changes—such as added comments and function name changes—to the sources.

Directly above the Communication layer is the *Data Model*; the "Model" in traditional MVC terms. The Data Model is a core component of ICE and forms the foundation upon which analyses are built, and data pertaining to low-level

representations being analysed is stored. The Data Model is encapsulated in a directed graph that models the structure of the Executable Entity under analysis. Within an Executable Entity function calls are used as the "edges" of this model since they are a ubiquitous mechanism to connect sections of code.

Above the Data Model, ICE provides a mechanism for the development of visualisations that can easily be created by an analyst. Visualisations are the components that an analyst interacts with and are the viewport into the data model, or the "View" in MVC. The MVC paradigm is rounded out with the "Controller" being the logic of ICE that enables a user to switch between the various Views and interact with them to better understand what is being presented.

IV. IMPLEMENTATION

Given the overall modular design of ICE outlined in Section III, the implementation of the current prototype was carried out by composing several existing technologies.

ICE was written in Java using the *Eclipse Rich Client Platform (RCP)* [14] as its foundation. The selection of this environment was guided in part by the prevalence of Java and the Eclipse RCP; however, it also allowed for a large amount of code reuse, specifically in the Tracks visualisation (Section IV-C2) and the Zest framework [15] for rendering graphs. The following subsections delve into the current communication, data model, and visualisations present in ICE.

A. Communication

As previously described in Section III, ICE allows bidirectional communication between data sources and itself. Communication is carried out over a predetermined port on the loopback interface. Taking this approach restricts communication to the localhost which cuts down on network traffic and is beneficial in the context of malware analysis, since machines dedicated to this task are typically separated from all networks to promote security.

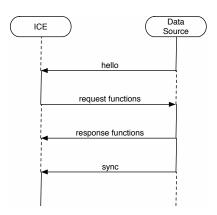


Fig. 2. Message passing between ICE and a data source

The communication protocol used in ICE is built upon the JSON model [16] and is outlined in Figure 2. Communication begins by a Data Source sending a hello message to ICE.

Upon receiving this message ICE creates a new entry in its Data Model that uniquely identifies the sender. A sample hello message is shown in Listing 1.

```
Listing 1. Sample JSON message

{

origin = ''program.exe'',
instance_id = 1234,
action = 'hello'',
actionType = null,
data = null
}
```

The fields presented in Listing 1 have all been chosen to allow for a flexible messaging protocol. The *origin* is the name assigned to the binary by the Data Source. For example, this could be the actual name of the binary or some other identifier such as a project name if Java byte-code was being analysed. The next field, the *instance_id*, is a unique numeric identifier of the Data Source. Currently this is taken to be the process ID of the Data Source since that is guaranteed to be unique due to ICE being restricted to a single machine. The action describes what the message does-it can be thought of as the 'verb' in the message. The messaging protocol defines numerous action values including: hello, request, and response. Similarly, the actionType field can be thought of as the 'adverb' of the message since it describes the action of the message being sent. Finally, the data field is specific to the combination of action and actionType and can contain any valid JSON object.

Once the entry in the Data Model has been created ICE then requests information about the functions contained in the Executable Entity—a binary or intermediate language representing assembly code—under analysis. For each function ICE requests: module name, function name, entry point, starting location, ending location and comment.

Note that the starting and ending location may be either an address or a line number depending the Executable Entity being analysed. Moreover, the *Entry point* is a boolean value indicating if the function is reachable from outside the Executable Entity and the function request triggers the Data Source to return similar information about all calls made within the function.

Upon sending all requested information, the Data Source sends a sync message causing ICE to commit all the data it has received, analyse the data for relationships between functions, and notify any visualisations currently open to update their view.

B. Data Model

The Data Model in ICE, depicted in Figure 3, is based on a directed graph and models the relationships between modules, functions, and instructions. Since ICE is able to support multiple Data Sources, the top-level of the model is an *Instance Map*. The Instance Map serves the purpose of mapping each *Instance* onto its corresponding communication socket.

For each connected Data Source there is an *Instance* object that describes it. The Instance object contains metadata such

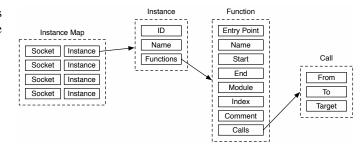


Fig. 3. ICE Data Model containing relationships between modules, functions, and instructions.

as the instance identifier and name, along with a hash table containing a mapping of locations to *Function* objects. Each Function object contains the following metadata:

- Entry Point Boolean value indicating if the function is an entry point.
- Name Name of the function.
- Start Location the function starts at.
- End Location the function ends at.
- Module Name of the containing binary.
- Comment Associated comment (if any).

In addition to this metadata, each Function object contains a list of *Call Sites*. These Call Sites contain pointers to their target Function object, creating a graph of functions.

Lastly, the Function object contains a mapping of locations to *Instructions*. Each Instruction object is comprised of the following attributes:

- Address Location of this instruction.
- Container Location of the function containing this instruction.
- **Flow Type** The "flow" of the instruction (e.g. normal, jump, call).
- Next List containing pointers to the next instruction(s).

Through this directed graph model of an Executable Entity it is possible to apply existing graph analysis algorithms to explore the Executable Entity at the function- or instruction-level as well as analyse the relationships between multiple Data Sources, potentially representing multiple Executable Entities. This also enables ICE to show correspondence between several levels of abstraction such as a high-level code base coupled with its binary.

C. Visualisations

The final major component of ICE are the visualisations. Visualisations are the primary user interface element, and allow an analyst to look inside a program. Currently ICE contains three visualisations: simple call graphs provided by *Cartographer*, sequence diagrams provided by *Tracks*, a *Control Flow Graph (CFG)*, and *Tours*. Due to the extensible design and implementation leveraging the Eclipse architecture, creating new visualisations is a straightforward process discussed further in Section V.

1) Cartographer: At the core of Cartographer is a function call graph [17], [18], generated as ICE receives information about functions from its Data Sources. While the function call graph is a core component of Cartographer, there are two additional central aspects: interactivity, and navigation.

The key to the interactivity of Cartographer is that the call graph is not static, meaning that it accepts modifications by the analyst. Actions that allow the analyst to gain insight and manage the process of program comprehension include:

- Assigning names to functions
- Assigning comments to functions
- Re-positioning nodes in the graph
- Navigating to associated code in the Data Source

With the ability to assign names and comments to functions the analyst is able to better track portions of code that have been analysed as well as assign meaningful names to functions. For example, a default function name in IDA Pro is of the form sub_<address> where <address> is the start address of the function. This name leaves much to be desired and a descriptive name such as decryptCode would provide the analyst a much clearer idea of what the function does. Similarly, with comments the analyst is able to better describe what the purpose of a function is.

Additionally, the analyst is able to place the nodes in a visualisation on the screen as desired, allowing for logical groupings—such as *these nodes have been analysed*—and to manage clutter in complex functions.

With respect to navigation, Cartographer supports navigation within the call graph and navigation to the code. Navigation within the call graph is achieved by double-clicking a node which will cause the call graph of the selected function to be displayed. In addition to displaying the call graph of the current function, Cartographer also keeps track of calls made—by doubling-clicking a function—in a call stack. The call stack displays a sequential view of all the calls made by the analyst along with information such as the address, name, and comment associated with the corresponding function. The call stack also supports navigation. Finally, it is possible to further navigate to the code in a function from Cartographer—in this case the function will be opened up in the containing Data Source.

2) Tracks: In previous work we developed Tracks [3], a visualisation tool that displays function calls within a program as a sequence diagram for both static and dynamic control flow. Through the sequence diagram an analyst is able to gain insight into the functions called as well as the order in which the calls are made. Tracks additionally shows calls to functions in external libraries as well as provides loop detection. Actions from the user are also supported in the connected Data Source, such as navigation to the code (either the function or specific call), setting breakpoints, and syncing renamed functions. Tracks was refactored to adhere to the design discussed in the previous Section, which allowed us an initial analysis of the plugin architecture that is used in ICE.

Tracks was built on top of Diver [11] to support extremely large traces so provides features such as hiding/collapsing call trees and package or module structures, setting new roots of diagrams, navigable thumbnail outline view, and saving the state of the diagram. Previous work has also investigated the use of comment threads within the sequence diagram itself [19].

3) Control Flow Graph (CFG): A Control Flow Graph (CFG) makes it possible to become better acquainted with the inner workings of a function by identifying key structures such as loops and branches. As with Cartographer and Tracks, the CFG visualisation is also interactive and provides filters to help pinpoint how instructions are related.

With respect to interactivity, the CFG visualisation supports zooming in and out, panning, and rotating the nodes of the graph. This is particularly important in this domain, where the number of lines of code in a function may have exploded several orders of magnitude relative to its high-level representation. In addition, individual nodes can be selected and moved to arbitrary locations, once again aiding in clutter management and comprehension.

The novel aspect of the CFG related specifically to comprehension is the ability to select from a set of filters. Each filter highlights the associated set of nodes making them visually discernible and easy to spot relative to the other nodes shown. The CFG in ICE provides filters for: Calls, Joins, and Loops. The Call filter simply highlights all call instructions and can be used to correlate where a call is located within a function without the need to analyse the assembly code. The Joins filter highlights all nodes that have an in- or out-degree greater than one. These nodes represent locations in the function where high-level control flow constructs such as if-then-else, try-catch, switch, and other related statements are found. By identifying these nodes it can be seen if certain instructions have an abnormal number of incident edges aiding in the identification of "interesting" locations in the function. Finally, Loop detection is based on the Tarjan strongly connected component [20] algorithm.

V. EVALUATION

The following evaluation of ICE focuses on the ability of the default visualisations to provide insight into the inner workings of several representations of code for a debugger. Specifically, it highlights the ability of ICE to simultaneously support multiple data sources.

A. Case Study

A common technique in software development is to leverage existing code as much as possible in order to cut down on development time and costs. Consequently, the use of libraries has become quite routine. Due to the ubiquity of libraries the need to be able to analyse the main executable as well as associated libraries is an aspect of program comprehension that must be considered.

As a demonstration of ICE and the default visualisations two open-source projects have been selected under the LLVM umbrella project. The main executable that will be analysed is the *LLVM Debugger (LLDB)* and the associated library is the *LLVM* implementation of the standard C++ library, *libstdc*++. The goal of this analysis is to use ICE to trace through a specific feature of the debugger, the command-line option parsing, and then to drill down to investigate the implementation of a function in the standard C++ library.

The first step of our analysis is to look at the implementation of the main() function since this is the entry point into LLDB and where command-line options are first available. The order in which function calls are made makes it beneficial to look at main() as seen through Tracks.

This combination of Cartographer and Tracks reveals that command-line arguments are passed to a function called <code>Driver::ParseArgs()</code>. Double-clicking this entry will enable Cartographer to display the call graph for <code>Driver::ParseArgs()</code>, Figure 4, which reveals that a large number of functions are called from here. Due to the lack of data-flow in this view it is not possible to determine which command-line arguments result in which function calls; but, it is clear that the arguments result in functions being called that can set the architecture to be debugged and query the version of LLDB among other things. One point of interest in the Cartographer visualisation is that the number of lines connecting two functions indicates the number of times the function is called.

One function call in Figure 4 that stands out is a call to function called push_back(). Although the function displayed contains the C++ name mangling from the compiler, this can be decoded to learn that this function is part of the standard C++ library string implementation. This can be confirmed by searching for push_back in the *Function Selector* of Cartographer. At this point we can leverage the Control Flow Graph visualisation in ICE to get a better idea of how the push_back() function is implemented.

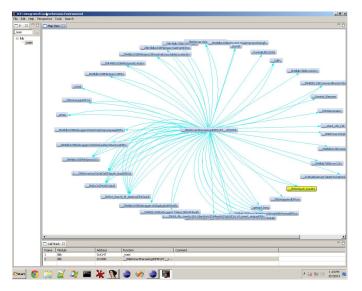


Fig. 4. Screenshot of Driver::parseArgs() in Cartographer

The ability to work with multiple data sources in ICE enables not only the collection of data from a variety of sources such as IDA Pro, debuggers, and even IDEs; but it also enables the data to be consolidated into a single model that can be analysed through the visualisations provided.

VI. DISCUSSION

The results of the preliminary evaluation in Section V show that ICE is capable of meeting the initial needs outlined in Section I through its modular design and implementation as overviewed in Table I.

TABLE I SUMMARY OF REQUIREMENTS MET BY ICE

Operate on multiple binaries	ICE core
Track map of the analysis	Tracks, Cartographer and Tours
Inline documentation through tagging	TagSea
Cross references between binaries	ICE core

Of these requirements, the need to be able to analyse multiple binaries at one time as well as have data propagated between binaries are largely met due to ICE being grounded on the MVC model. Through the graph-based Data Model discussed in Section IV-B, ICE consolidates the various pieces of information that describe any low-level representation into a single model accessible through visualisations. Furthermore, the ability to leverage Tours allows for tracking the progress of an analysis, in particular those that must touch many sections of many files, while creating repeatable documented sequences as the analysis occurs.

Currently development of ICE has been focused on well-formed binaries; however, it is important to consider how the design and implementation might faire when faced with binaries typically associated with malicious software, or even just malformed inlined assembly code. At the Data Model level, some of these situations would be problematic in the current prototype. Specifically, due to the Data Model being largely centred around functions, if the binary under analysis either does not contain functions or contains functions with multiple entry and exit points, ICE would be incapable of displaying those functions in its visualizations.

With respect to the implementation, the flexibility of ICE is realised through the symbiosis of Eclipse, the graph-based Data Model, and the loosely coupled communication mechanism. The flexibility afforded by these technologies permeates ICE, enabling analysts to work with multiple binaries and/or intermediate language flies simultaneously, and integrate visualisations that focus on the many levels of relationships between the levels of abstraction.

The Eclipse-based implementation of ICE is also able to support rapid development of new visualisations, just as Eclipse supports plugins. For example, the initial development of Cartographer took approximately one day of development time; including the time necessary to become familiar with the Zest rendering library. Similarly, refactoring Tracks and Tours took under a day to configure for the ICE environment.

Due to the age of the Tours project, the main challenge with integrating this tool came from finding the correct version of multiple build files. In a robustly supported environment like Eclipse, this task simply takes time and investigation of online resources.

VII. CONCLUSIONS AND FUTURE WORK

Program comprehension is an extremely difficult task where relationships between components, and the manner in which information flows through the program, must be reasoned about at both a high- and low-level in many codebases. The proposed Integrated Comprehension Environment provides a framework for engineers to visualise and share the relationships they find in a program through the use of interactive call graphs and sequence diagrams. ICE is specifically designed to be flexible and extensible, allowing new visualisations to be developed and shared as they are required. We have explored how ICE meets many of the needs in this domain, how it can easily accept new sources, and how it can be used to mitigate issues of scale at this level where the number of instructions is formidable compared to high-level code. Though ICE establishes a good first step as a prototype comprehension environment in this domain, it still has several limitations, which we plan to address in future work.

Currently, ICE relies on external applications for disassembly. Although ICE does not directly display assembly code to the analyst, but instead provides layers of abstraction through visualisations, the Data Sources must be able to disassemble the binary in question in order to provide information on functions and the data required for a control flow graph. Additionally, the Tours tool only works with textual representations of code. In future work we hope to integrate it further into the ICE framework so that Tours may be used with visualisations. We envision a tool where one would be able to use the visualisation of ICE to discover new things about a code set and then walk someone else through their thought process by using Tours. In this way Tours is a highly adaptable tool as it can be used in a variety of contexts with a number of different sources, including high-level abstractions.

Another consideration, when working with object-oriented code bases, is that it is currently challenging to associate functions with the containing class. Additionally, the lack of a class diagram leaves some questions surrounding the code base.

Finally, it would be critical to perform a user study and investigate the potential of a normalised representation of instructions. Through a user study it would be possible to better understand how ICE can fit into an analysts workflow, what cognitive barriers—either broken or found—may be associated with ICE and program comprehension in this domain, and to better understand what limitations may be associated when working with code written using a variety of programming paradigms.

REFERENCES

- J. Baldwin, A. Teh, E. Baniassad, D. van Rooy, and Y. Coady, "Applying social psychology techniques to requirements elicitation within highlyspecialized industry software groups," *In Submission*, 2013.
- [2] I. Guilfanov, "Decompilers and beyond," BlackHat USA 2008, pp. 1–12, Jul 2008.
- [3] J. Baldwin, P. Sinha, M. Salois, and Y. Coady, "Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems," AUIC 2011, pp. 1–10, Nov 2011.
- [4] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," *Information Systems Security*, pp. 1–25, 2008.
- [5] F. Bellard, "QEMU: A Fast and Portable Dynamic Translator," in Proceedings of the USENIX Annual Technical Conference, 2005, pp. 41–46.
- [6] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing CPU emulators," in *Proceedings of the 2009 International Conference* on Software Testing and Analysis (ISSTA). Chicago, Illinois, USA: ACM, pp. 261–272.
- [7] D. Brumley and I. Jager, "The BAP Handbook," 2009.
- [8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 Inter*national Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.
- [9] R. DeLine, "Staying oriented with software terrain maps," in *DMS*, A. Guercio and T. Arndt, Eds. Knowledge Systems Institute, 2005, pp. 309–314.
- [10] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. M. Drucker, and G. G. Robertson, "Code Thumbnails: Using Spatial Memory to Navigate Source Code," in *IEEE Symposium on Visual Languages and Human-Centric Computing*. Brighton, UK: IEEE Computing Society, 2006, pp. 11–18.
- [11] C. Bennett, D. Myers, M.-A. Storey, and D. German, "Working with 'monster' traces: Building a scalable, usable, sequence viewer." in *In Proceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, Vancouver, Canada, 2007, pp. 1–5.
- [12] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. New York, NY, USA: ACM, 2010, pp. 455–464.
- [13] T. Reenskaug, "Thing-model-view-editor: An example from a planning system," Tech. Rep., 1979.
- [14] The Eclipse Foundation. [Online]. Available: http://www.eclipse.org/home/categories/rcp.php
- [15] Zest: The Eclipse Visualization Toolkit. [Online]. Available: http://www.eclipse.org/gef/zest/
- [16] [Online]. Available: http://www.json.org/
- [17] B. G. Ryder, "Constructing the call graph of a program," Software Engineering, IEEE Transactions on, no. 3, pp. 216–226, 1979.
- [18] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy, "Constructing the procedure call multigraph," *Software Engineering, IEEE Transactions* on, vol. 16, no. 4, pp. 483–487, 1990.
- [19] J. Baldwin and Y. Coady, "Social security: collaborative documentation for malware analysis," in *Proceedings of the 12th Annual Conference of the New Zealand Chapter of the ACM Special Interest Group on Computer-Human Interaction*, ser. CHINZ '11. New York, NY, USA: ACM, 2011, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/2000756.2000759
- [20] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM journal on computing, vol. 1, no. 2, pp. 146–160, 1972.