IDA Pro Plugins for CodeHawk-Binary

October 5, 2023

Contents

1 Introduction

The CodeHawk-Binary analysis is a powerful analysis tool for binaries of various architectures. Its core analysis is based on the mathematical theory of abstract interpretation [3, 4]. It includes its own disassemblers, currently supporting x86, arm32 (including Thumb-2), mips, and Power32 (in progress). It takes as input a (possibly stripped) binary, disassembles it, and constructs functions, control flow graphs, and a callgraph. It then creates an over-approximating semantic translation of the functions into CHIF, the CodeHawk Internal Form, for analysis, and translates the resulting (over-approximating) invariants back into the context of the assembly code. This invariant-generation process is performed in a series of rounds, incrementally adding variables discovered in previous rounds. The final set of invariants is saved (in highly compressed form) and forms the basis for all subsequent analyses and liftings.

The CodeHawk-Binary Analyzer comes with a comprehensive command-line interpreter to produce a wide variety of analysis results, including annotated assembly code, annotated control-flow graphs and callgraphs, and potential vulnerability reports. It does not, however, have a graphical user interface. For general reverse engineering tasks a graphical user interface, such as provided by IDA Pro, Binary Ninja, Ghidra, or angr, is often a preferred way of interaction for exploration and navigation. None of these tools, however, provide the deep analysis results that CodeHawk generates. Part of the reason for a more shallow analysis in these tools is, of course, that deeper analysis often takes too much time creating response times that are not acceptable for an interactive tool.

With CodeHawk-Binary Analyzer plugins for these tools we hope to offer users the best of both worlds. CodeHawk analysis can be performed off-line. All analysis results are saved in full, such that they can be used for many different purposes, via a comprehensive python API that is called directly from the python plugin code. During an interactive session with the preferred tool, these analysis results can be accessed and deployed as desired, controlled by interactively invoking plugins, thus injecting analysis results directly into the tool's database and thereby augmenting the tool's own analysis results. Because the plugins only need to extract data, they do not perform any expensive analysis, response times are comparable to other actions typically performed in these tools.

In this report we describe and illustrate an initial selection of python plugins implemented for IDA-Pro.

2 Installation

2.1 CodeHawk

The CodeHawk core analysis is performed by codehawk, an open-source tool available from GitHub [2], implemented in ocaml. Installation instructions for codehawk are provide for Linux and MacOS in the README.

Alternatively Aarno Labs makes available a docker container that provides an easier interface to running codehawk without the need to perform any of the installation steps??

2.2 CodeHawk-Binary

The interaction with the ocaml codehawk analyzer is managed via CodeHawk-Binary, implemented in python, and also available from GitHub [1]. CodeHawk-Binary requires python to be installed with version 3.8 or higher, but does not have other dependencies, except if graphical output is requested directly from the CodeHawk-Binary command-line processor (e.g., control flow graphs or callgraphs), which requires the dot utility to be installed.

2.3 IDA-Pro

Running IDA-Pro python scripts requires a paid license for IDA-Pro. None of the IDA-Pro decompilers (sold separately by Hex-Rays) is required to run the scripts here described, so a minimal IDA-Pro license suffices.

The plugins require IDA-Pro to have CodeHawk-Binary included in its PYTHONPATH. This can be accomplished in a few different ways:

- Setting the PYTHONPATH in the terminal, if IDA-Pro is started from the command-line;
- Copying the full `chb` directory from the CodeHawk-Binary repository to the idabin/plugins directory;
- Making a symbolic link from the idabin/plugins directory to the `chb` directory (??)

The plugins themselves can simply be copied into the idabin/plugins directory from where they will be automatically loaded and initialized when IDA-Pro is started.

3 Analysis Preparation

All scripts described in the next section assume that the target binary has already been analyzed by CodeHawk.

After installation and setting of the appropriate PATH and PYTHONPATH environment variables, analysis is accomplished by the command

```
> chkx analyze <filename-of-binary> --reset
```

Depending on the size and complexity of the binary this analysis can take anywhere from a few minutes to many hours, so should typically be done off-line. It only needs to be done once[1]

The analysis results will be saved in a subdirectory of the directory that holds the binary, with the name `<filename-of-binary>.ch`, from where they will be directly accessed by the IDA-Pro plugins.

[1] In some cases we may want to feed back data from IDA-Pro into the analysis by means of analysis hints, in which case the analys must be rerun. This situation is somewhat unusual, and is described later.

4 IDA Pro Plugins

4.1 Overview

We present the first four IDA Pro plugins using CodeHawk analysis results that have been developed so far, indicated in the menu shown in Figure 1. All of them will be illustrated in more detail in the sections below.

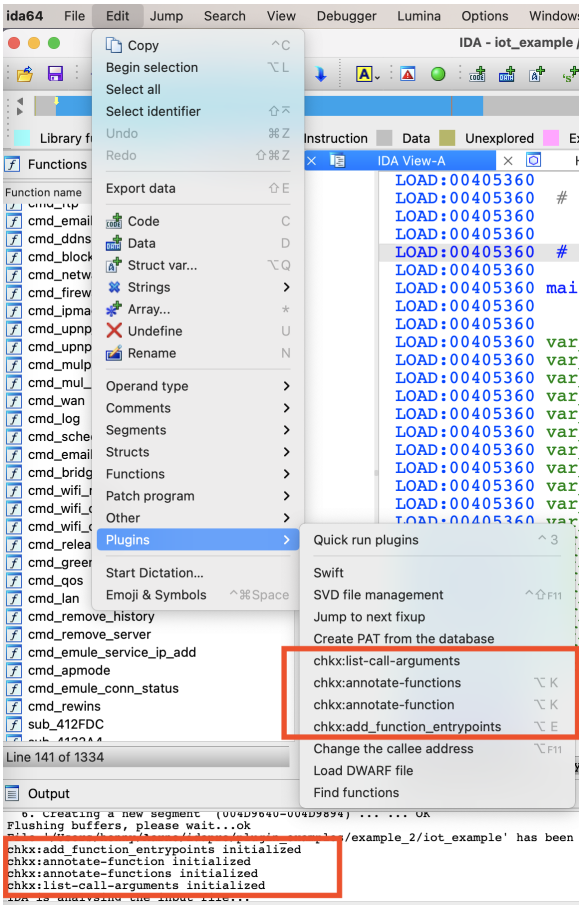


Figure 1: IDA Pro Edit menu with CodeHawk plugins (with chkx prefix)

- `chkx_list_call_arguments`: List all calls to a particular (user-selected) function together with their arguments;
- `chkx_annotate_function`: Annotate the instructions of the function at the current location of the cursor in the IDA-View;
- `chkx_annotate_functions`: Annotate the instructions of all functions;
- `chkx_add_function_entrypoints`: Add function entry points found by CodeHawk but not by IDA Pro.

4.2 Getting Started

The plugins described below are illustrated on a cgi-handling program for an http-server found on a commodity router. The binary is a dynamically linked MIPS binary (632 KB) with more than 1000 functions. The binary can be analyzed with CodeHawk with the command:

```
> chkx analyze iot_example
```

The analysis takes about 30 minutes. When finished load the binary into IDA-Pro, and invoke the plugins as described.

4.3 Adding Function Entry Points

In some cases CodeHawk analysis identifies more function entry points than IDA Pro. The plugin `chkx:add_function_entrypoints` facilitates adding these extra function entry points to IDA Pro automatically.

Figure 2 shows the initial IDA Pro display just after loading the example binary. Notice in particular the bar at the top, showing a colored map of of the code address range, with blue denoting the presence of functions and brown denoting code that is not part of any function. In addition, notice the number of functions found shown in bottom left of the figure: 1005.

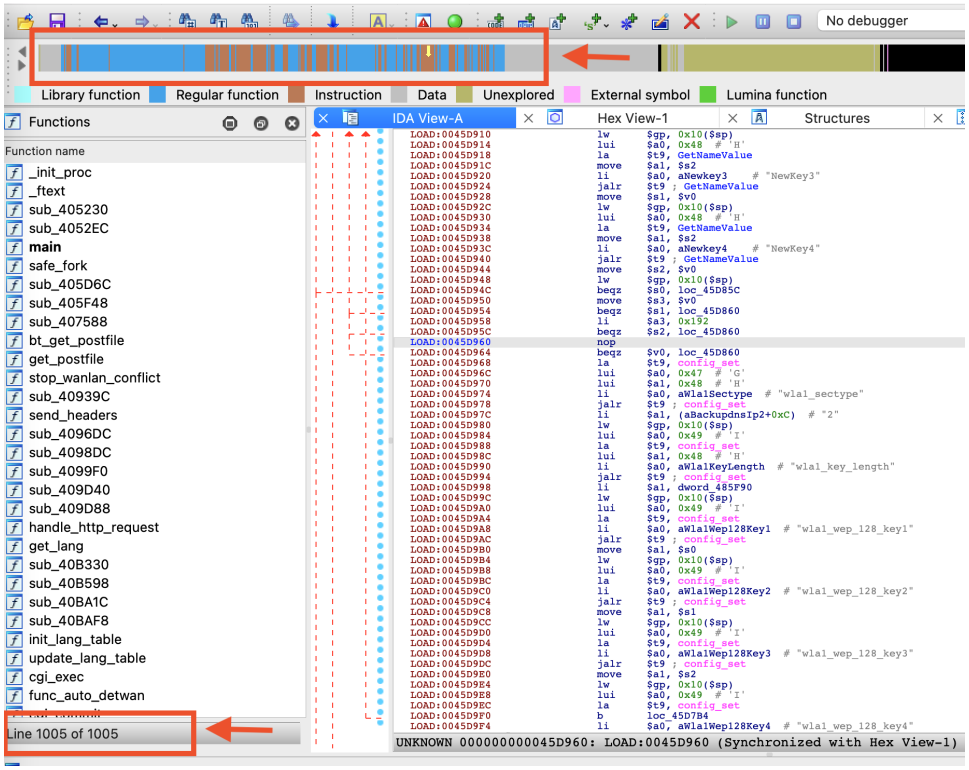


Figure 2: IDA View after initial IDA analysis

Figure 1 shows how to invoke the plugin `chkx:add_function_entrypoints`: Edit → Plugins → `chk:add_function_entrypoints`. The plugin checks for every function entry point in the Code-

Hawk analysis results whether the address is contained in an IDA-Pro-recognized function; if not, it adds a new function entry point at that address and leaves the remaining function construction to IDA-Pro. The plugin takes a few seconds.

Figure 3 shows the same IDA Pro display as shown in Figure 2 after the plugin finished. Notice the effect on the code bar at the top: it now is almost entirely blue, and the number of functions, as shown in the bottom left, now stands at 1363; 358 new function entry points were added by the plugin.

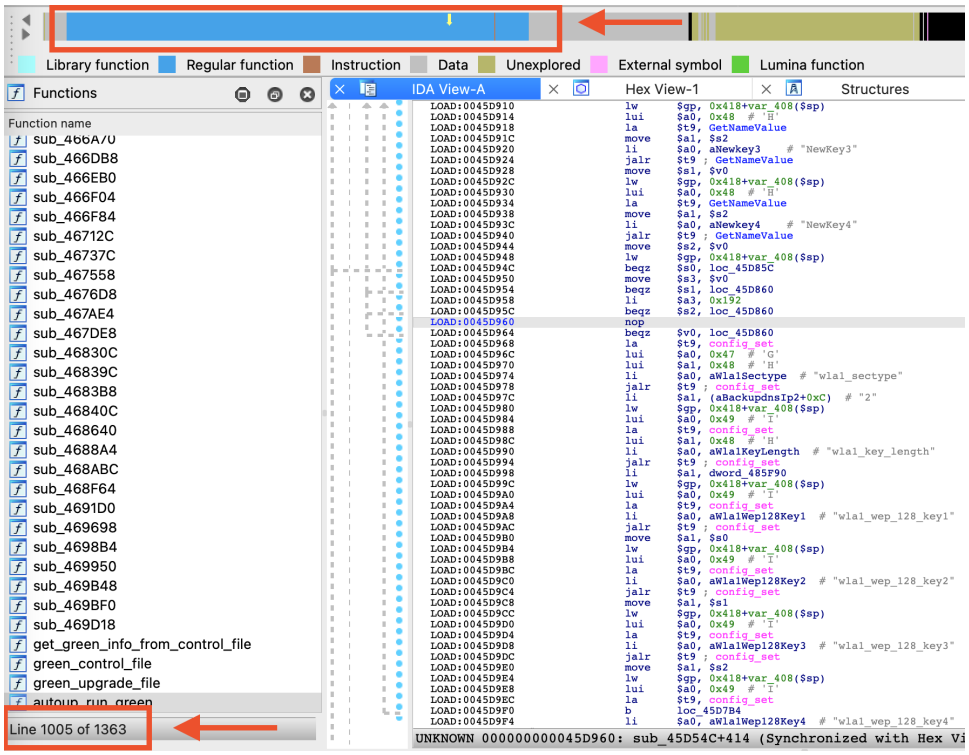


Figure 3: IDA View after invoking the `chkx:add_function_entrypoints` plugin

Note The displays referenced above were obtained with IDA-Pro version 7.6, which was not very good at recognizing MIPS functions by preamble. The latest IDA-Pro version, version 8.3 does a much better job, and the number of functions added (with the same analysis results) is reduced to 25, as shown in Figure 4. Subsequent figures show IDA-Pro version 8.3. However, the plugins should work on any version 7.x and higher.

4.4 Listing Function Calls

IDA-Pro provides a menu option to list all calls to a particular function. Figure 5 shows the resulting display for `strcpy`. For every call it shows just the address of the call and the function in which the call is located. Clicking on the call moves the user to the IDA assembly view, where the call and its surroundings can be further inspected.

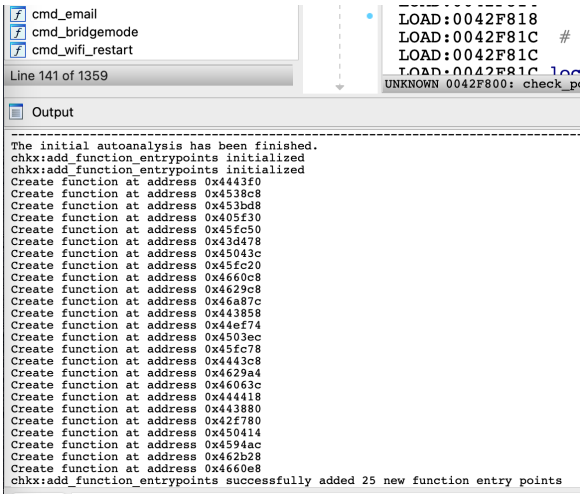


Figure 4: Adding function entry points to IDA-Pro, version 8.3

With CodeHawk analysis results more information can be associated with the calls, in particular, the arguments with which the function is called. Figure 6 shows result of invoking the `chkx_list_call_arguments` plugin (after entering the text `strcpy` in the presented dialog box).

The results are displayed in a standard IDA-Pro display in a new tab. As the roles of the arguments of `strcpy` are known (destination and source), the arguments can be characterized in terms of where they write to (e.g., stack or heap etc) and where the source is read from (e.g., a function return value or argument to the caller function).

The utility of having these extra columns (writes-to, reads-from) is that they can be sorted by clicking on the header of the column, like any other column, thus enabling grouping them by kind, or they can be filtered.

Figure 7 shows the result of clicking on the the `src` header, displaying all calls together that receive their input from the `cgi_value` function (a function that typically returns data that originates directly from an HTTP input form, and thus is especially interesting for vulnerability/exploitation research). Note from Figure 6 that this binary has 266 calls to `strcpy`. Only 33 of these receive input from `cgi_value`. Being able to select these (or group them) with a single click for further inspection, rather than having to inspect all 266, can considerably speed up the search for exploitable vulnerabilities.

Another function of interest for vulnerability research is, of course, `sprintf`. As shown in Figure 8, our example binary has 359 calls to `sprintf`.

Not all of these calls are interesting. In particular calls without any `%s` format specifiers are unlikely to be exploitable, because most other format specifiers result in a formatted argument with a fixed maximum length. Figure 9 shows how we can use the standard IDA-Pro `Modify filters ...` dialog to select only those calls that have a `%s` format specifier in the format string and then highlight those in which `rtn_cgi` appear in one of the varargs as the source of one of those `%s` specifiers. Note that none of these selections or highlightings would be possible without the function arguments supplied by the CodeHawk analysis results.

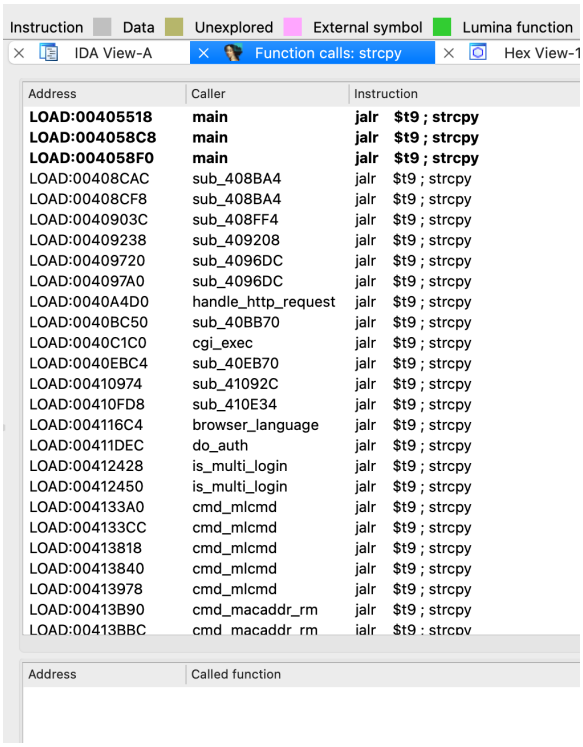


Figure 5: Native IDA-Pro display to show calls to a particular function

4.5 Annotating Functions

Similar to the native IDA-Pro display listing function calls, the displays resulting from invoking the `chkx_list_call_arguments` also allow navigating to the location of a call in the corresponding IDA assembly view, by double-clicking on the call in the display. For example, if we double-click on the `strcpy` call shown in Figure 10, the view is moved to the corresponding function in the IDA assembly view, shown in Figure 11

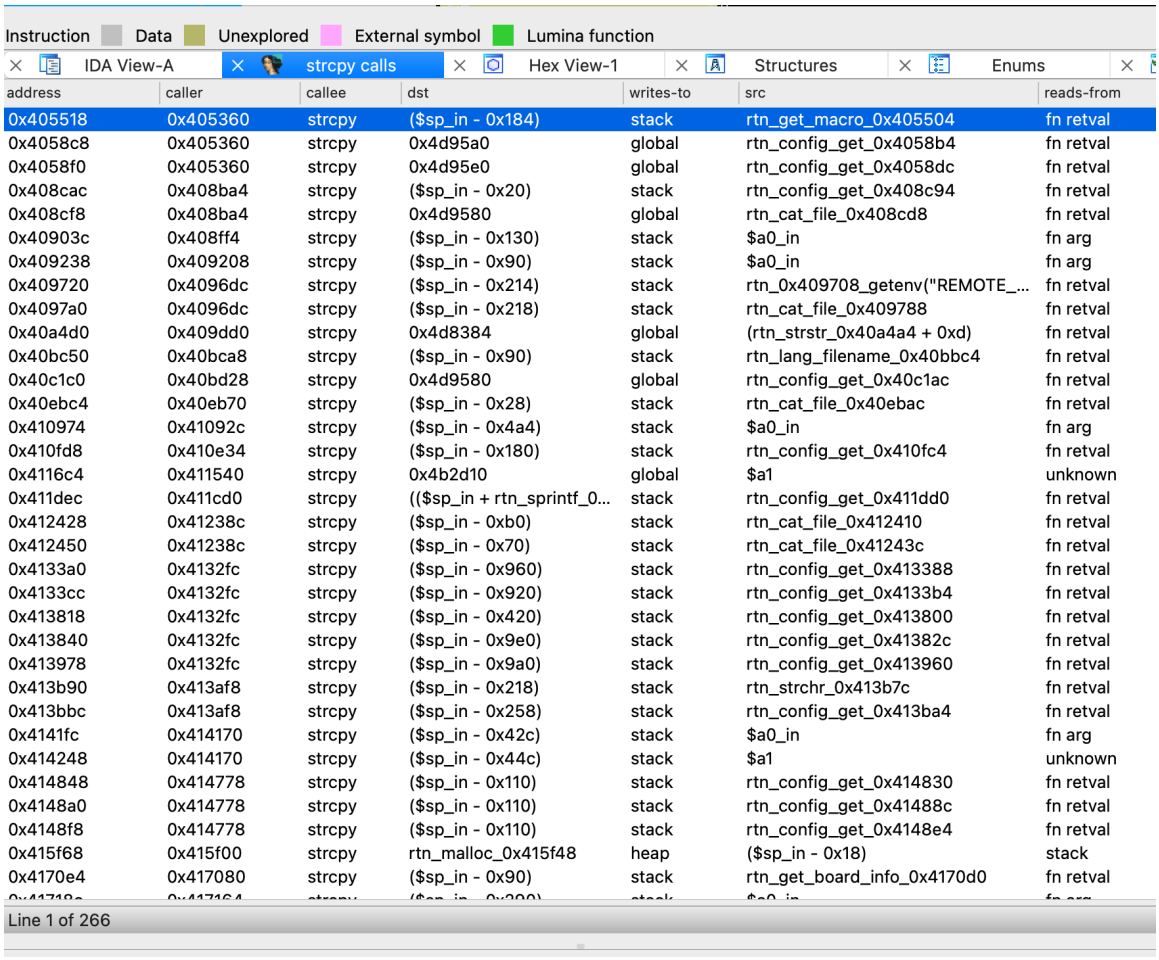
This display doesn't provide a lot of information beyond the bare assembly code. By invoking the `chkx_annotate_function` plugin while the cursor is on this function, we can make the display more informative: the plugin annotates all instructions in the function with CodeHawk analysis results. Figure 12 shows the resulting display. The display allows making a direct connection between the `cgi_value` key associated with the `strcpy` call that copies its value.

The `chkx_annotate_functions` plugin adds annotations to *all* functions. This may take some 10-20 secs. During its execution, progress is indicated as shown in Figure 13.

References

[1] CodeHawk Binary Analyzer. `https://github.com/static-analysis-engineering/CodeHawk-Binary`.

[2] CodeHawk Tool Suite Analyzer. `https://github.com/static-analysis-engineering/codehawk`.

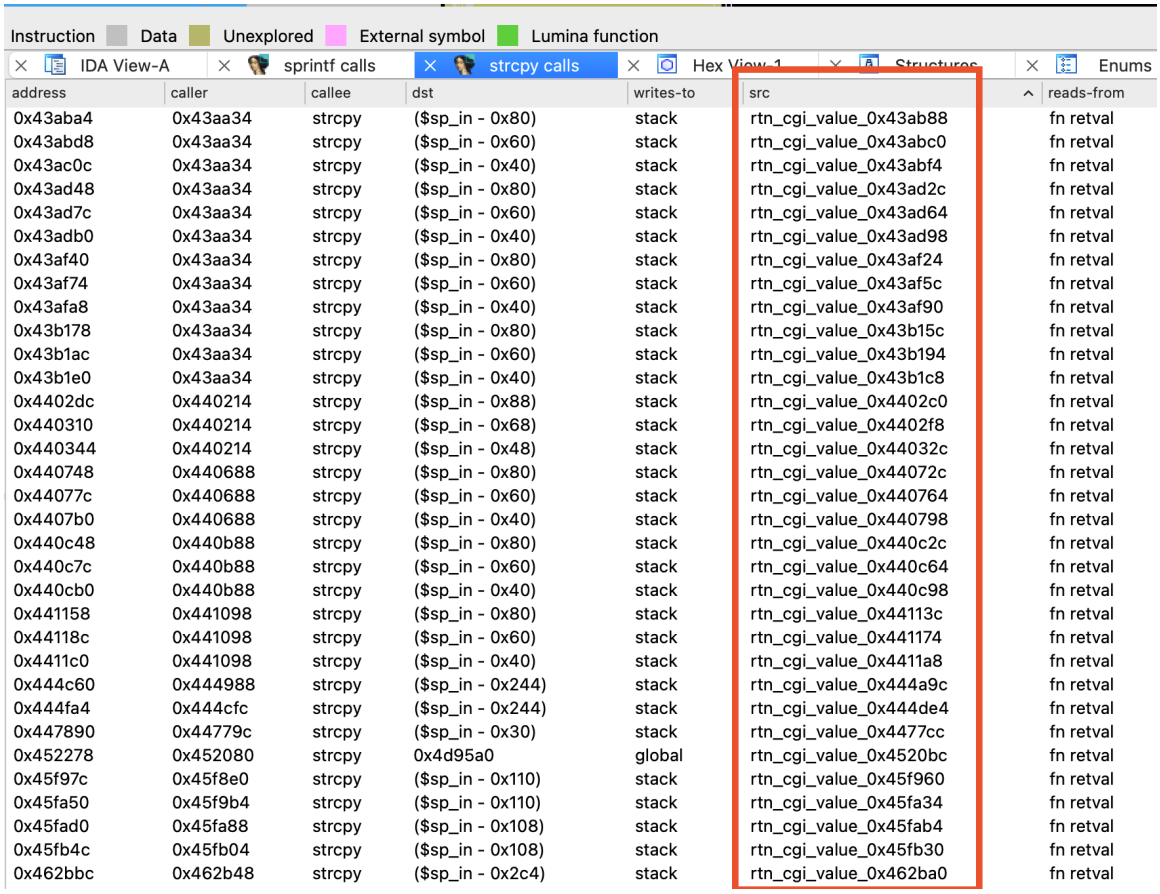


address	caller	callee	dst	writes-to	src	reads-from
0x405518	0x405360	strcpy	($sp_in - 0x184)	stack	rtn_get_macro_0x405504	fn retval
0x4058c8	0x405360	strcpy	0x4d95a0	global	rtn_config_get_0x4058b4	fn retval
0x4058f0	0x405360	strcpy	0x4d95e0	global	rtn_config_get_0x4058dc	fn retval
0x408cac	0x408ba4	strcpy	($sp_in - 0x20)	stack	rtn_config_get_0x408c94	fn retval
0x408cf8	0x408ba4	strcpy	0x4d9580	global	rtn_cat_file_0x408cd8	fn retval
0x40903c	0x408ff4	strcpy	($sp_in - 0x130)	stack	$a0_in	fn arg
0x409238	0x409208	strcpy	($sp_in - 0x90)	stack	$a0_in	fn arg
0x409720	0x4096dc	strcpy	($sp_in - 0x214)	stack	rtn_0x409708_getenv("REMOTE_...	fn retval
0x4097a0	0x4096dc	strcpy	($sp_in - 0x218)	stack	rtn_cat_file_0x409788	fn retval
0x40a4d0	0x409dd0	strcpy	0x4d8384	global	(rtn_strstr_0x40a4a4 + 0xd)	fn retval
0x40bc50	0x40bca8	strcpy	($sp_in - 0x90)	stack	rtn_lang_filename_0x40bbc4	fn retval
0x40c1c0	0x40bd28	strcpy	0x4d9580	global	rtn_config_get_0x40c1ac	fn retval
0x40ebc4	0x40eb70	strcpy	($sp_in - 0x28)	stack	rtn_cat_file_0x40ebac	fn retval
0x410974	0x41092c	strcpy	($sp_in - 0x4a4)	stack	$a0_in	fn arg
0x410fd8	0x410e34	strcpy	($sp_in - 0x180)	stack	rtn_config_get_0x410fc4	fn retval
0x4116c4	0x411540	strcpy	0x4b2d10	global	$a1	unknown
0x411dec	0x411cd0	strcpy	(($sp_in + rtn_sprintf_0...	stack	rtn_config_get_0x411dd0	fn retval
0x412428	0x41238c	strcpy	($sp_in - 0xb0)	stack	rtn_cat_file_0x412410	fn retval
0x412450	0x41238c	strcpy	($sp_in - 0x70)	stack	rtn_cat_file_0x41243c	fn retval
0x4133a0	0x4132fc	strcpy	($sp_in - 0x960)	stack	rtn_config_get_0x413388	fn retval
0x4133cc	0x4132fc	strcpy	($sp_in - 0x920)	stack	rtn_config_get_0x4133b4	fn retval
0x413818	0x4132fc	strcpy	($sp_in - 0x420)	stack	rtn_config_get_0x413800	fn retval
0x413840	0x4132fc	strcpy	($sp_in - 0x9e0)	stack	rtn_config_get_0x41382c	fn retval
0x413978	0x4132fc	strcpy	($sp_in - 0x9a0)	stack	rtn_config_get_0x413960	fn retval
0x413b90	0x413af8	strcpy	($sp_in - 0x218)	stack	rtn_strchr_0x413b7c	fn retval
0x413bbc	0x413af8	strcpy	($sp_in - 0x258)	stack	rtn_config_get_0x413ba4	fn retval
0x4141fc	0x414170	strcpy	($sp_in - 0x42c)	stack	$a0_in	fn arg
0x414248	0x414170	strcpy	($sp_in - 0x44c)	stack	$a1	unknown
0x414848	0x414778	strcpy	($sp_in - 0x110)	stack	rtn_config_get_0x414830	fn retval
0x4148a0	0x414778	strcpy	($sp_in - 0x110)	stack	rtn_config_get_0x41488c	fn retval
0x4148f8	0x414778	strcpy	($sp_in - 0x110)	stack	rtn_config_get_0x4148e4	fn retval
0x415f68	0x415f00	strcpy	rtn_malloc_0x415f48	heap	($sp_in - 0x18)	stack
0x4170e4	0x417080	strcpy	($sp_in - 0x90)	stack	rtn_get_board_info_0x4170d0	fn retval

Figure 6: Result of invoking the the `chkx_list_call_arguments` plugin for `strcpy`

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[4] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.



address	caller	callee	dst	writes-to	src	reads-from
0x43aba4	0x43aa34	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x43ab88	fn retval
0x43abd8	0x43aa34	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x43abc0	fn retval
0x43ac0c	0x43aa34	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x43abf4	fn retval
0x43ad48	0x43aa34	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x43ad2c	fn retval
0x43ad7c	0x43aa34	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x43ad64	fn retval
0x43adb0	0x43aa34	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x43ad98	fn retval
0x43af40	0x43aa34	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x43af24	fn retval
0x43af74	0x43aa34	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x43af5c	fn retval
0x43afa8	0x43aa34	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x43af90	fn retval
0x43b178	0x43aa34	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x43b15c	fn retval
0x43b1ac	0x43aa34	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x43b194	fn retval
0x43b1e0	0x43aa34	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x43b1c8	fn retval
0x4402dc	0x440214	strcpy	($sp_in - 0x88)	stack	rtn_cgi_value_0x4402c0	fn retval
0x440310	0x440214	strcpy	($sp_in - 0x68)	stack	rtn_cgi_value_0x4402f8	fn retval
0x440344	0x440214	strcpy	($sp_in - 0x48)	stack	rtn_cgi_value_0x44032c	fn retval
0x440748	0x440688	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x44072c	fn retval
0x44077c	0x440688	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x440764	fn retval
0x4407b0	0x440688	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x440798	fn retval
0x440c48	0x440b88	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x440c2c	fn retval
0x440c7c	0x440b88	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x440c64	fn retval
0x440cb0	0x440b88	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x440c98	fn retval
0x441158	0x441098	strcpy	($sp_in - 0x80)	stack	rtn_cgi_value_0x44113c	fn retval
0x44118c	0x441098	strcpy	($sp_in - 0x60)	stack	rtn_cgi_value_0x441174	fn retval
0x4411c0	0x441098	strcpy	($sp_in - 0x40)	stack	rtn_cgi_value_0x4411a8	fn retval
0x444c60	0x444988	strcpy	($sp_in - 0x244)	stack	rtn_cgi_value_0x444a9c	fn retval
0x444fa4	0x444cfc	strcpy	($sp_in - 0x244)	stack	rtn_cgi_value_0x444de4	fn retval
0x447890	0x44779c	strcpy	($sp_in - 0x30)	stack	rtn_cgi_value_0x4477cc	fn retval
0x452278	0x452080	strcpy	0x4d95a0	global	rtn_cgi_value_0x4520bc	fn retval
0x45f97c	0x45f8e0	strcpy	($sp_in - 0x110)	stack	rtn_cgi_value_0x45f960	fn retval
0x45fa50	0x45f9b4	strcpy	($sp_in - 0x110)	stack	rtn_cgi_value_0x45fa34	fn retval
0x45fad0	0x45fa88	strcpy	($sp_in - 0x108)	stack	rtn_cgi_value_0x45fab4	fn retval
0x45fb4c	0x45fb04	strcpy	($sp_in - 0x108)	stack	rtn_cgi_value_0x45fb30	fn retval
0x462bbc	0x462b48	strcpy	($sp_in - 0x2c4)	stack	rtn_cgi_value_0x462ba0	fn retval

Figure 7: Sorting the result of invoking the `chkx_list_call_arguments` plugin for `strcpy`

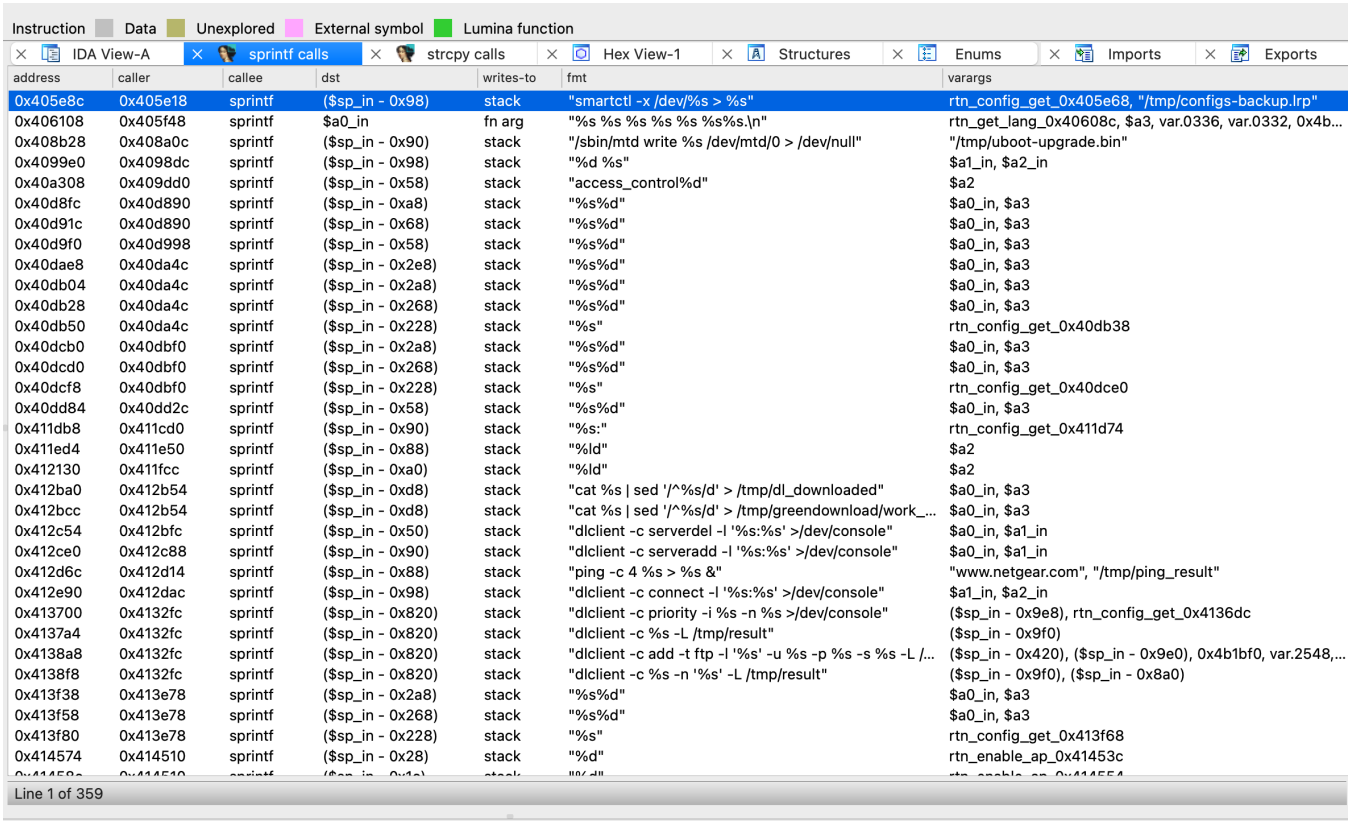


Figure 8: Result of invoking the `chkx_list_call_arguments` plugin for `sprintf`

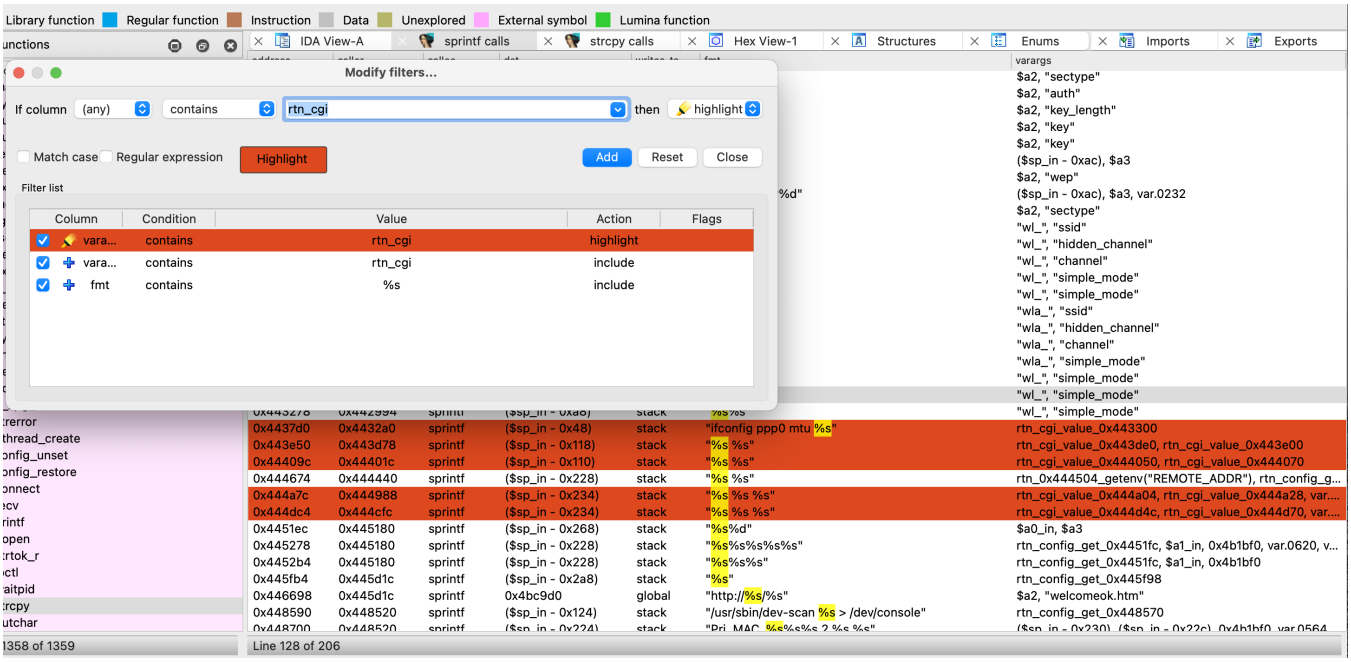


Figure 9: Filtering and highlighting the result of invoking the `chkx_list_call_arguments` plugin for `sprintf`

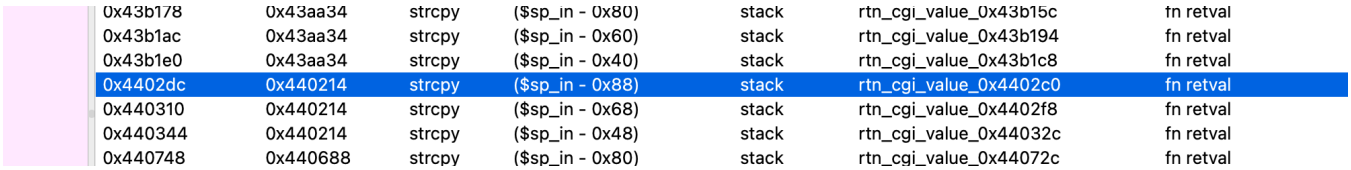


Figure 10: Select one of the calls in the result of invoking the `chkx_list_call_arguments` plugin for `strcpy`

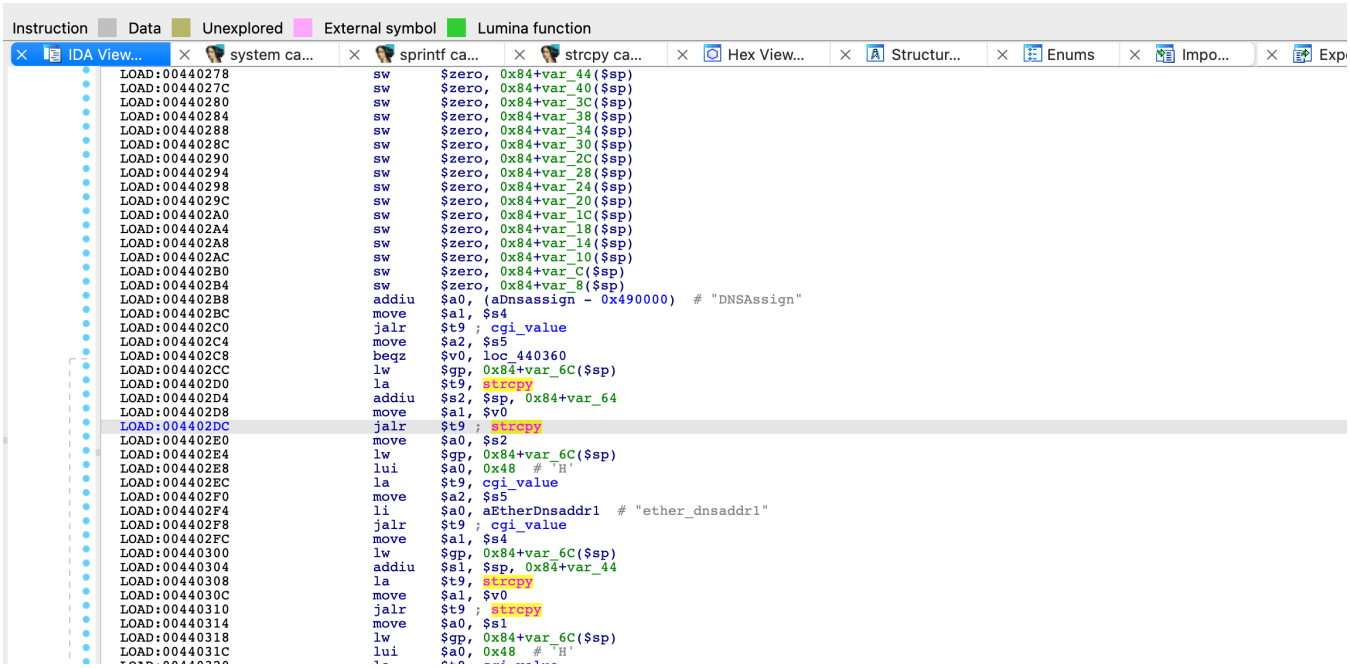


Figure 11: Standard IDA-Pro assembly view for function `sub_440214`

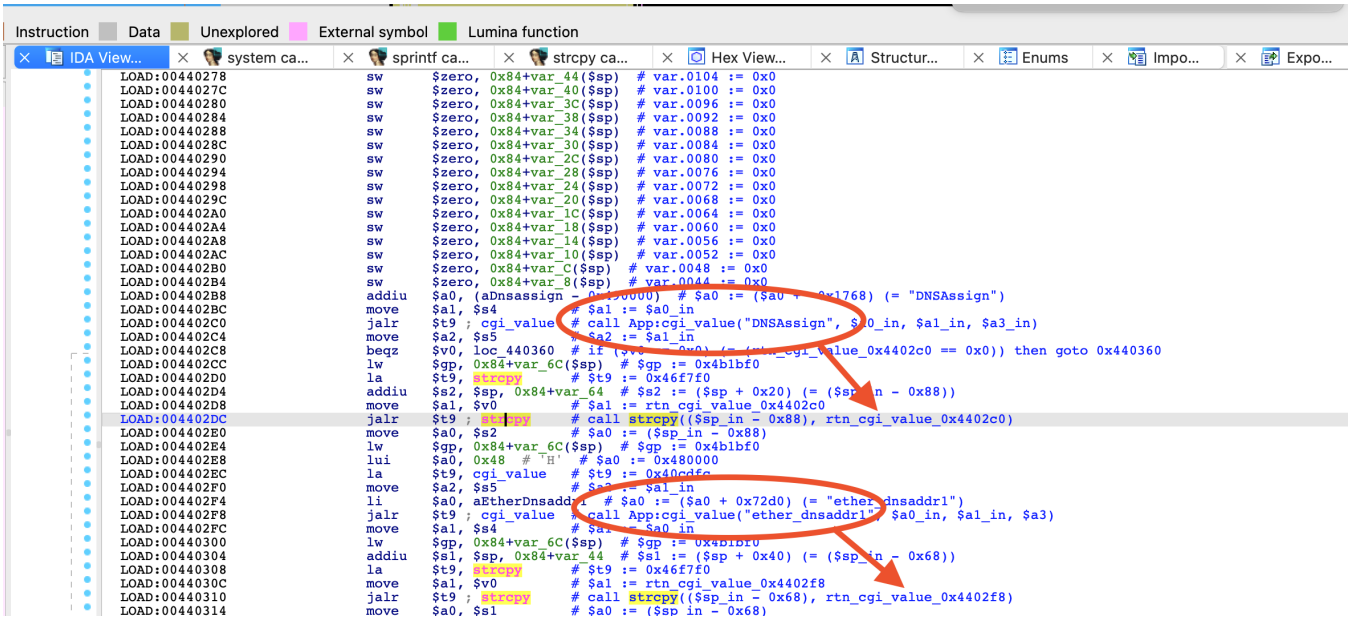


Figure 12: Standard IDA-Pro assembly view for function `sub_440214` annotated with CodeHawk results data

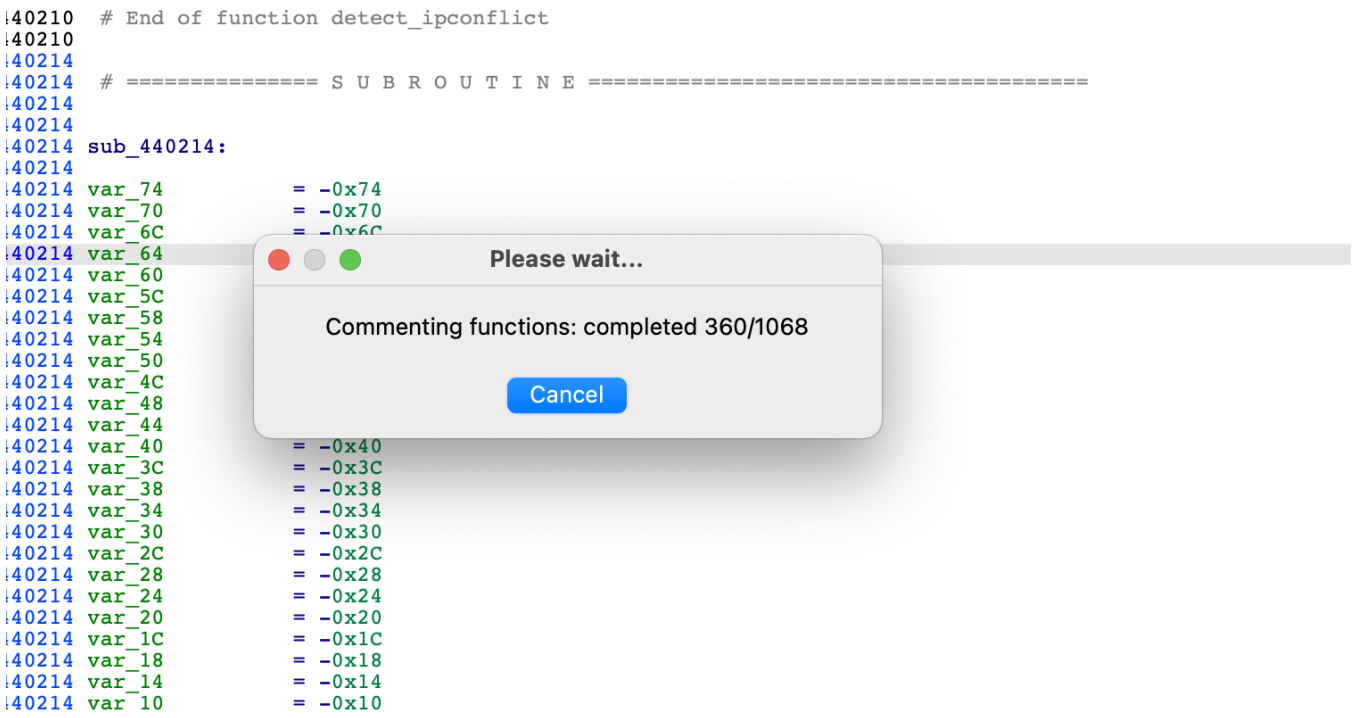


Figure 13: Progress indicator for `chkx_annotate_functions`