A Comparative Binary Analysis Tool (CBAT)
Final Technical Report

Contract # N68335-18-C-0107

Program Name: Total Platform Cyber Protection Innovative Naval Prototype (TPCP INP)
Solicitation #: N00014-17-S-B010

Technical POC
Office of Naval Research – DoDAAC: N00014
Dr. Dan Koller
Phone: 703-696-4212
Email: daniel.p.koller.civ@us.navy.mil

The Charles Stark Draper Laboratory – 555 Technology Square, Cambridge, MA 02139
JT Paasch – 617-258-2577, jpaasch@draper.com
Michael Crystal – 617-258-1039, mcrystal@draper.com

09-November-2022
Draper Reference #: CON03077

Distribution Statement A: Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
14-Nov-2023	Final Technical Report	09-Nov-2017	09-Nov-2022

4. TITLE AND SUBTITLE
Final Technical Report: A Comparative Binary Analysis Tool (CBAT)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
N68335-18-C-0107	N/A	N/A
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
N/A	CLIN 0006	Data Item No. A003

6. AUTHOR(S)
Dr. JT Paasch, Draper

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Charles Stark Draper Laboratory, Inc. (Draper)
555 Technology Sq.
Cambridge, MA 02139-3539

8. PERFORMING ORGANIZATION REPORT NUMBER
CON03077-FTR

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research - DoDAAC: N00014
Dr. Dan Koller
One Liberty Center
875 N. Randolph Street
Arlington, VA 22203-1995

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR Code 311

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Statement A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
None.

14. ABSTRACT

This report describes the research and software development carried out by the CBAT team in the context of ONR's Total Platform Cyber Protection (TPCP) program. The CBAT tool enables automated verification of late-stage software customizations and is sufficiently mature to handle realistic software.

CBAT is now a useful and complete tool, and has been released publicly as open-source software. It has extensive documentation and an in-depth tutorial that walks users through much of its functionality. We also include a guide to the underlying framework BAP that CBAT is built on top of, in order to better serve the power user who wants to contribute to CBAT, and to further adoption. CBAT is already in use by other projects at Draper, and can be directly utilized in other areas such as automated program repair or translation verification.

Overall, the CBAT project advances verification research and demonstrates that formal verification can enable late-stage software customization with evidence for recertification.

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

15. SUBJECT TERMS
Binary analysis, static analysis, program transformation, weakest precondition analysis, formal verification, differential verification, relational verification, SAT solver, SMT solver, OCaml, Binary Analysis Platform (BAP), Comparative Binary Analysis Tool (CBAT)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE	UU	
U	U	U		

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Michael R. Crystal	(617) 258-1039

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Table of Contents

Acknowledgements **4**

1 Introduction **5**

1.1 Differential Program Analysis 6
1.2 Binary Analysis Frameworks 9
1.3 Putting It All Together 10
1.4 Report Structure 11

2 Evaluating Open-Source Binary Analysis Toolkits **13**

2.1 Motivation 13
2.2 BAP and angr: Overview 13
2.3 Testing Environment 15
2.4 Extracting and Using Control Flow Data 16
2.5 Value-Set Analysis 20
2.6 Survey Summary 22

3 CBAT Implementation and Internals **23**

3.1 Weakest Precondition Analysis 23
3.2 Comparative Analysis with WP 31
3.3 Analysis Features 33
3.4 SMT Solver Integration 39
3.5 Testing and Debugging Tools 40

4 Evaluating and Improving CBAT **47**

4.1 Evaluation Criteria 47
4.2 Applying and Improving CBAT: Example 1 52
4.3 Applying and Improving CBAT: Example 2 56

5 Other Applications **58**

5.1 Automated Program Repair 58
5.2 Translation Validation 60
5.3 Specification Synthesis 62

5.4 Conclusion 65

Appendices **66**

A A Gentle Introduction to CBAT **67**

A.1 Overview 67

A.2 Binary Programs 68

A.3 CBAT Usage 70

A.4 Tripping Asserts 71

A.5 Hooking `wp` up to `bildb` 77

A.6 4-Rooks 77

A.7 Find a Null Dereference 80

A.8 Comparing Function Outputs 82

A.9 Comparing Function Calls 85

A.10 Custom Postcondition (One Binary) 90

A.11 Custom Pre and Postcondition (One Binary) 93

A.12 Custom Postcondition (Two Binaries) 95

A.13 Reference Guide 96

A.14 Tutorial Conclusion 96

B CBAT Feature Reference **97**

B.1 Viewing the Man Page 97

B.2 General Options 97

B.3 Single Program Analysis 99

B.4 Comparative Analysis 100

B.5 SMTLIB Cheat Sheet 101

C Using BAP: An Introduction and Reference **103**

C.1 Preliminaries 103

C.2 Extending BAP 110

C.3 The Knowledge Base 139

C.4 Compilation Units 164

C.5 Semantics 191

C.6 KB Analyses 227

D References 235

Acknowledgements

We would like to acknowledge the valuable contributions to the CBAT project and to this report made by the following current and former Draper staff:

John Altidor
Niki Carroll
Chris Casinghino
Michael Dixon
Chloe Fortuna
Dustin Jamner
Sam Lasser
Benjamin Mourad
Jeff Opper
Justin Restivo
Nick Roessler
Cody Roux
Greg Sullivan
Greg Trost
Yu-Jye Tung
Phil Zucker

1 Introduction

Over the last few decades, the scale and complexity of software in commercial and DoD systems have grown enormously. Even in 2012, estimates put the amount of code in an average consumer automobile at over 100M lines [1]. Large software projects are notoriously difficult to build and secure, with some estimates putting the number of bugs as high as 50 per 1,000 lines of code [2]. Further, while some of this size can be attributed to increasingly digital and automated systems, a substantial portion comes from *software bloat*: unnecessary code from sources like large libraries intended for a single function, or unused features left over from previous versions. Software bloat reduces efficiency and can compromise the safety and security of a system.

These issues are complicated by the fact that source code is often unavailable for systems when vulnerabilities or bloat are discovered. Military systems often use COTS software and include legacy code; similarly, large commercial systems are often built with components from many suppliers that may not provide source and may not continue to exist for the full life cycle of the system. Thus, realistic efforts to secure software must enable *late-stage software customization*, where software can be modified after deployment and without source code.

In 2017, the Office of Naval Research (ONR) launched the Total Platform Cyber Protection program (TPCP), motivated by these issues. This effort has resulted in software transformations that de-bloat, de-layer, and add security constructs to existing software. Of course, after applying these transformations, it is necessary to verify and validate that the resulting executables retain the desired functionality from the original software and continue to operate correctly. This verification is critical, both because unexpected changes in the behavior of functioning defense systems can have disastrous results, and because low-level software transformations are particularly error-prone [3]. Rapid verification with strong guarantees and documented results is essential to ensuring that transformed software can be redeployed rapidly, meeting any recertification and reaccreditation requirements.

Draper's Comparative Binary Analysis Tool (CBAT) project, part of TPCP, addresses this verification challenge, as shown in Figure 1. CBAT verifies that a binary still behaves as expected after it has been stripped of irrelevant functionality and undergone security enhancing transformations. The role that CBAT plays in late-stage software customization is that of a rigorous verifier: it confirms that a modified binary program behaves identically to the original, except for the intentionally introduced changes. The tool's results can be used as part of a recertification or reaccreditation process, so that the modified binary can be redeployed.

For a verification effort this critical, it is natural to consider using formal methods. "Formal methods" broadly refers to mathematical techniques for specifying and reasoning about the behavior of software. These techniques offer high levels of assurance by applying mathematical logic to software verification. However, approaches based on formal methods traditionally require substantial human effort to develop formal models of systems, and suffer from limitations in scaling with the size and complexity of modern software. In the context of TPCP, these problems are exacerbated by the need to work directly from binaries, rather than a higher-level source representation.

CBAT addresses these verification challenges with the novel combination of two recent research developments: *differential program analysis* techniques and *open-source binary analysis toolkits*.

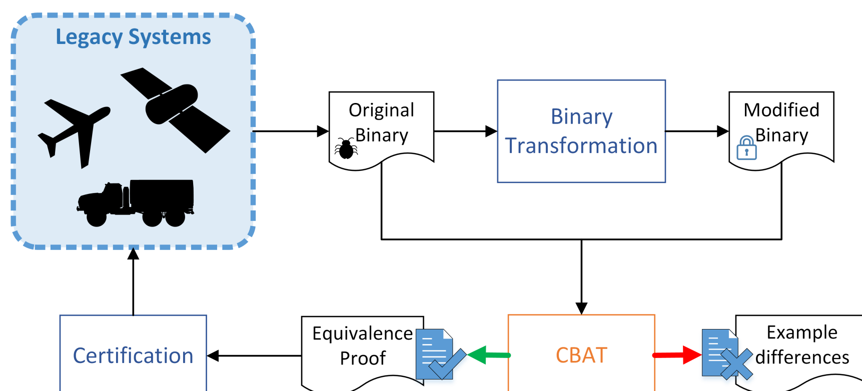


Figure 1: CBAT provides verification evidence that a binary behaves as expected after modification, for use in a certification process, or finds examples of unexpected changes in behavior.

Differential program analysis is an approach to formally verifying *modifications* to programs. This technique observes that some traditional limitations to formal verification can be resolved when a previous version of the program exists and is used to establish "base truth" about correct behavior. This approach has primarily been applied to source code verification in research prototypes. The CBAT project has matured it into a practical binary analysis technique by building on top of open-source binary analysis frameworks. These frameworks lift binaries to a platform-independent low-level intermediate language that is suitable for verification.

In the remainder of this section, we introduce these two techniques in detail (Sections 1.1 and 1.2). In Section 1.3, we describe novel elements of CBAT that overcome scalability and usability challenges of existing differential analysis and binary analysis tools. We conclude with an overview of the rest of this report (Section 1.4).

1.1 Differential Program Analysis

In traditional program analysis, a user begins with a program and some property of that program they would like to check. For example, a user might want to know if the program is memory safe. The user then uses two tools to check whether the property holds. The first tool they use examines the program with the property at hand, and it computes a *verification condition*, which is a boolean proposition that is true if and only if the property holds. Then, the user invokes an SMT solver to check whether the verification condition can be falsified. If so, the solver reports the inputs to the program it has found that cause the program to violate the property. If not, then the user may conclude that the property holds, because the SMT solver has shown that there are no inputs to the program that can cause it to violate the property in question.

This traditional approach to verification often has trouble scaling, however. The kinds of properties one is usually interested in can require verification conditions that are too complex to compute. And even if the verification condition does manage to get computed, its size and complexity can be too difficult for SMT solvers to check.

The core insight of differential program analysis research is that having access to earlier versions of a program can make it possible to build verification conditions that are easier to check [4, 5, 6, 7, 8]. This insight is particularly applicable in the context of TPCP, where the goal is to verify not that the transformed programs behave correctly in every way whatsoever, but rather that they simply function the same as before, modulo some intended changes. These simpler verification conditions

Original program	Version after program modification

<pre>typedef struct {
  int msg_type;
  void* data;
  int status;
  ...
} msg;
 	
void process_message (msg m) {
  switch (m.msg_type) {
    case NAV:
      adjust_heading(m.data);
 	
    case LOG_STATUS:
      log_current_status(m.status);
      break;
 	
    case DEPLOY:
      deploy_payload();
      break;
 	
    ...
  }
}</pre>
	<pre>typedef struct {
	  int msg_type;
	  void* data;
	  ...
	} msg;
	
	void process_message (msg m) {
	  switch (m.msg_type) {
	    case NAV:
	      adjust_heading(m.data);
	
	    case DEPLOY:
	      deploy_payload();
	      break;
	
	    ...
	  }
	}</pre>

Figure 2: Example of bug inserted by bad program transformation.

express *relative* correctness properties that relate the earlier and current versions of a program.

To illustrate this idea in a setting more relevant to TPCP, consider the C code in Figure 2. This is a highly-abstracted version of a UUV component that interprets command messages and sends instructions out along a communication bus. It begins with a partial definition of a "message" struct type, containing a message type, data, and a status code. This is followed by a function that processes the message by examining the message type and performing the appropriate action.

In this case, the programmer has decided that when a navigation message adjusts the UUV's heading, it should be logged. The programmer has implemented this functionality by omitting a `break` statement from the `NAV` case of the `switch` statement. This causes the `NAV` case to "fall through" to the next case, and execute the `LOG_STATUS` case too. By omitting the `break` statement, the programmer has gotten two cases for the price of one, so to speak.

Suppose now that it is many years later, and this software is being re-evaluated. Upon examination, it becomes clear that the operational system never uses the `LOG_STATUS` message, and that the message was in fact intended only for debugging. At this point, it is decided that a late-stage customization tool should be employed to remove this logging functionality, and thereby debloat this software some.

After the debloat pass, the modified version of the program looks as shown on the right side of Figure 2. As can be seen there, the `LOG_STATUS` case has been removed.

However, notice also that the `NAV` case still has no **`break`** statement, and so if a `NAV` message is received, this new debloated program will execute the `NAV` case, and then fall through to the `DEPLOY` case. This is clearly going to be a problem. Subsequent operators will be quite surprised to find that the UUV deploys a payload every time it changes heading.

Differential program verification can detect such a problem. To apply CBAT to a case like this, the user of CBAT identifies a *relative correctness property* for CBAT to check. This property can be selected from CBAT's library of built-in properties that apply to most software, or it can be a custom property based on the requirements of the particular application being verified. The bug shown above can be caught by several simple properties. We highlight two: one broad property that is built into CBAT, and an application-specific property that we built based on the requirements of this example:

1. When considering a debloating transformation whose only job is to remove functionality, one broadly applicable relative correctness property is: "On any given input, the modified program will only call functions that the original program would also call". Because this property often applies when a debloating transformation has been used, we have included it in CBAT's library of common properties.

2. This component of the UUV has requirements describing how the messages it received should be handled. One of those requirements can be formalized as the property: "The `deploy_payload` function is only called after a `DEPLOY` message is received." This property applies specifically to this application, but CBAT lets user provide their own custom properties such as this one.

Given either property, CBAT extracts a corresponding verification condition from both the original and modified binaries. It then validates the verification condition with an SMT solver, which reveals that the property does not hold for this pair of programs. Moreover, in its output, CBAT specifically identifies the problem case that exercises the problem: namely, when the `msg_type` field of the input has the `NAV` value.

This example, which comes from our original proposal for CBAT, is easily handled by CBAT. A version of it is included in Appendix A.9. Both here and in the appendix, we present these programs in high-level source code, but this is only for readability and ease of presentation. CBAT operates directly on the binary, and the principles are exactly the same when applied to the compiled machine code.

At the project's outset, we identified three key challenges for existing differential verification techniques. We briefly revisit those challenges, and note how we addressed them over the course of the project:

- **Property generation.** Many pre-existing approaches to differential verification were designed to check specific relative correctness properties or were restricted in the types of properties that could be expressed. CBAT goes beyond this by letting the user provide any custom property expressible in the industry-standard SMT-LIB format. This support for custom properties is described in more detail in Appendix A.10 to A.12.

- **Scaling.** While pre-existing research suggested that differential program analysis could improve the scalability of verification due to the simpler nature of relative verification conditions, there was little empirical evaluation of the scaling limits of the approach. We measured and improved CBAT's performance over the course of the project, as described in Section 4.1.1, and in many cases, it scales to interesting properties of real-world binary programs.
- **Handling executables.** Many pre-existing differential analysis prototypes work primarily with source-level languages like C. To handle the TPCP case of late-stage software customization, it was essential for CBAT to support binary-to-binary transformations. We achieved this goal by adapting the existing differential analysis literature to modern binary analysis frameworks.

1.2 Binary Analysis Frameworks

The last decade has seen renewed interest in the feasibility of direct analysis of binary code. For example, in the DARPA Cyber Grand Challenge, several teams demonstrated the ability to automatically detect, exploit, and repair some vulnerabilities in the challenge binaries [9]. And some of those teams have since open sourced their binary analysis tool kits.

These and other tools for binary analysis have converged on a broad common strategy. The core idea is to lift from binary to a platform-independent immediate representation (IR). This IR language is designed to make lifting from multiple platforms possible while retaining an execution model that is simple enough to make analysis feasible. Put another way, the IR can serve as a formal specification that is automatically extracted from the binaries. Then, multiple analysis passes can be implemented directly on the IR, resulting in a portfolio of platform-independent binary analysis capabilities.

There are now a number of popular commercial and open-source tools in this space, but there are three that are particularly worthy of mention. There are the open-source tools BAP [10] and angr [11], which were developed and used by the teams that placed first and third in the just-mentioned DARPA Cyber Grand Challenge, respectively. The NSA has also released its own binary analysis framework, Ghidra, as open-source software [12].

At the project's outset, we identified three key challenges for application of the existing binary analysis frameworks. We briefly revisit those challenges, and note how we addressed them over the course of the project:

- **Scaling.** Even the best binary analysis frameworks have considerable scale limitations when it comes to complete formal verification of realistic binaries. They mitigate this in various ways, including limiting the types of analyses that can be performed or focusing on unsound, approximate techniques. CBAT aims to offer strong guarantees about real programs. It employs a combination of techniques to scale better on many programs and offer alternatives when an analysis task is infeasible. First, as discussed above, a focus on relative correctness improves scalability by using the original binary as a kind of oracle about intended behavior for a modified binary. Second, the CBAT project has carefully explored many options for simplifications to the model of program behavior used by our analyses. While a completely

sound analysis that proves facts about very large binaries remains out of reach, we carefully document the assumptions of our model and provide users with many options to customize it according to the needs of their situation (see Section 4). Third, CBAT offers the user the option of lifting only the parts of the binary that they care about. As a consequence, lifting can be done in seconds rather than hours. In effect, the size of the binary program is not a limiting factor for CBAT, since the user only needs to operate on the small part they want to analyze.

- **Diversity.** The diversity of available frameworks and lack of public information about their strengths and weaknesses presented us with the challenge of selecting one. At the time the CBAT project began, no independent group had undertaken a serious comparison of the available tools. Little information was available about differences in areas like supported architectures, scalability, and maturity. To ensure that CBAT built on the strongest foundation, we undertook a study of the advantages and disadvantages of the two most popular frameworks available at the project's outset: BAP and angr (Ghidra had not been released yet). We built prototype analyses with each tool and reported on their relative strengths and weaknesses. Ultimately, we selected BAP as the best tool for our purposes. This study is described in Section 2.

- **Ease of Use.** Like most academic research tools, binary analysis frameworks can be challenging to use for non-experts. Even the designers of the most popular tools have identified usability as a primary gap [13]. For TPCP, where the verification tools must be transitionable broadly across the Navy, requiring a PhD in formal methods to get started is a non-option. Draper built a front-end for CBAT that offers a convenient interface for subject-matter experts to verify their software, without requiring substantial formal methods expertise. We have also written extensive documentation, including a detailed tutorial to get new users up and running quickly (see Appendix A). For those who want to become power users, we have also written an extensive guide to BAP (see Appendix C).

1.3 Putting It All Together

The CBAT tool suite puts together the aforementioned technical solutions into a practical package suitable for real world applications. The high-level user-facing architecture is shown in Figure 3.

To use CBAT, a user supplies two executables as input: the original binary and a modified binary that has undergone late-stage customization. The user then picks a property to be verified which describes the expected relationship between these two binaries. The user can select this property from CBAT's library of built-in properties, they can supply a custom, application-specific property, or they can combine several of the above in order to check multiple properties at once.

When CBAT runs its verification, there can be three potential outcomes:

- Verification is successful. In this case, CBAT successfully checked that the two binaries are related in the expected way (i.e., the modified binary is safe). This result offers a strong mathematical guarantee up to certain assumptions, which depend on user configuration and are described further in Section 4.1.3.

- Verification fails. In this case, CBAT provides a specific program input that will exercise the unexpected difference in behavior. CBAT also describes the execution path that leads to the failure.

- Verification times out, or the solver gives up. This can occur because the binary is too large or complex to be formally verified in the available time, because the verification condition is too complex (e.g., it involves too many quantifiers), or because it utilizes a mathmatical theory that the solver has not yet implemented. When this happens, the user can adjust the tool's parameters and try again, or employ CBAT's tools for exploring and testing the binary's behavior directly.

In all cases, CBAT logs the results of the verification attempt in detail, providing a useful artifact to simplify recertification processes.

Because CBAT can verify such a wide-variety of relational properties, CBAT is in fact useful to other domains beyond TPCP. In essence, wherever there is a need to compare two programs to see if they stand in a particular relationship, CBAT can be applied directly to verify the presence (or lack) of that very relationship. We will say more about these other areas of application in Section 5.

The CBAT tools are primarily developed as plugins for the BAP analysis framework in the OCaml programming language. The tools are available as open source software on GitHub [14].

1.4 Report Structure

The CBAT project was a five year effort with a three year base period and two optional one-year extensions. This document is the final report of the project. It offers an overview of the research work executed over the full course of CBAT's development.

The remainder of the report is structured as follows:

- Section 2 describes a lightweight evaluation of the BAP and angr binary analysis frameworks we conducted early in the project to decide which framework we would use for CBAT.

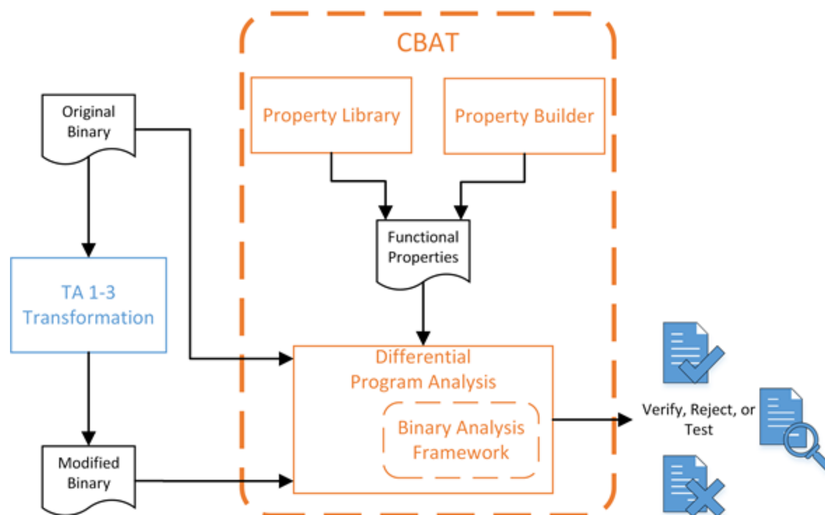


Figure 3: CBAT Architecture

- Section 3 provides an overview of CBAT's implementation and describes some of its more advanced features.
- Section 4 describes how we evaluate CBAT and gives examples of how we have improved it over the course of the project on the basis of that evaluation.
- Section 5 describes other application domains beyond TPCP where CBAT can be applied directly to solve real-world problems.

This report also includes the following appendices:

- Appendix A gives an overview of CBAT's usage, covering its core capabilities in detail.
- Appendix B summarizes the tool's main interface and options.
- Appendix C gives an overview of the BAP framework and its internal ecosystem.

2 Evaluating Open-Source Binary Analysis Toolkits

CBAT brings together two developments: (1) the emergence of open-source binary analysis toolkits that standardize the process of lifting binaries to an intermediate representation (IR) and analyzing them at the IR level, and (2) a line of research in differential program analysis. The goal of CBAT is to bring these differential analysis techniques to binaries with the help of the open-source tools. We began the project by evaluating the open-source tools available at the time (November 2017).

The two most widely used tools in 2017 were CMU's Binary Analysis Platform (BAP) [10] and UCSB's angr [11]. Both are still in wide use today, and have been joined by the NSA's Ghidra tool as a top contender. In this section, we expand on the results of a user study we conducted to evaluate these two tools, previously published in NASA Formal Methods [15]. Our study focused on usability, documentation, and performance, as we were most interested in our ability to confidently build useful tools. Other studies have examined decompilation accuracy [16, 17].

2.1 Motivation

If you want to analyze the version of your program that actually gets executed, you may need to examine its binary code directly. There are a variety of tools to help with this task. Some of these tools are general libraries that can help you build your own custom program analyses.

We compared two popular, open-source binary analysis libraries: BAP [10] and angr [11]. We examine how each library constructs call graphs (CGs) and control flow graphs (CFGs). We have implemented a value-set analysis (VSA) and an algorithm to compare call graphs in both BAP and angr, and assess how easy it is to build real-world program analyses using each.

Our contributions include the following:

- We detail some technical differences in the way BAP and angr identify function starts, as well as how they construct CGs and CFGs.
- We provide a first-hand account of building custom analyses with these libraries, and we profile the tools we built.
- We conclude by identifying the strengths and weaknesses of each tool, and give our impression of their suitability for building sound, static program analyses.

In Section 2.2, we provide a brief overview of BAP and angr. In section 2.4, we discuss some peculiarities of how BAP and angr construct CGs and CFGs. In section 2.5, we explain the custom analyses we built, as well as why we ended up more confident about the analysis built with BAP. Finally, in section 2.6, we summarize the findings of the study.

The data from our analyses can be found in our CBAT GitHub repository [1].

2.2 BAP and angr: Overview

BAP and angr both begin by lifting a binary program to an intermediate representation (IR). Both provide facilities for analyzing this lifted IR program rather than the original binary directly. This

[1] https://github.com/draperlaboratory/cbat_tools/tree/master/bap-angr

permits the reuse of analysis passes across different ISAs (e.g., x86 and ARM), but places a high burden on the lifter to correctly model the behavior of the original program.

BAP lifts to its own IR, the BAP Intermediate Language (BIL), while angr lifts to VEX, which is the IR used by Valgrind. The differences between BIL, VEX, and other potential IR choices are not the focus of this study, but have been studied elsewhere [16].

Once a binary has been lifted to the IR, you can use built-in BAP or angr program analyses, or write your own tools to explore the lifted program. BAP is written in OCaml and angr is written in Python; it is easiest to write your own tools in the host language.

The idiomatic use of each tool is similar: first you load a binary into a "project," and then perform your own analysis. For example, you might begin by generating a CFG. In angr:

```
import angr

exe = "/bin/true"
project = angr.Project(exe)
cfg = project.analyses.CFGFast()
# Now do something with the CFG...
```

In BAP, the process is similar. In the following example, we select byteweight [18] to identify function starts, then we load the program into a project, retrieve the lifted IR program, and generate a CG:

```
open Core_kernel.Std;;
open Bap.Std;;

let exe = Project.Input.file "/bin/true";;
let byteweight = Rooter.Factory.find "byteweight";;
let Ok proj = Project.create exe ?rooter:byteweight;;
let lifted_prog = Project.program proj;;
let cg = Program.to_graph lifted_prog;;
(* Now do something with the CG... *)
```

Both libraries are easy to use in a REPL. For instance, you can import angr in a Jupyter console to explore a particular binary, and you can import BAP into `utop`, or use the `baptop` REPL that BAP provides.

For batch mode, angr analyses can be written as straight-forward Python scripts that import `angr` and proceed from there. BAP offers a modular plugin architecture: each plugin makes a pass over the program, where it extracts information, alters the IR, or performs other tasks. Passes can be chained together.

Both tools offer a reasonably easy point of entry into programmatic binary analysis, with library functions for common tasks such as generating a CG or CFG. The communities for both projects are extremely helpful and responsive—most of our technical questions about the tools were immediately answered.

2.3 Testing Environment

For the experiments below, we worked on an Ubuntu 16.04.4 VM (Linux 4.4.0-87 and GCC 5.4.0) with 16Gb of memory and eight 2.2 GHz cores. We report results for angr 7.8.9 with vanilla Python 2.7, BAP 1.5.0 with OCaml 4.05.0. We also experimented with running angr with PyPy 6.0 rather than Python. We found PyPy to be less efficient for small programs and more efficient for larger ones. We ran BAP with a `-no-cache` flag, but normally BAP caches disassembly and other information, so repeat runs are significantly faster.

For a set of sample executables to run BAP and angr analyses on, we selected 11 programs from GNU utils. Each is listed in Table 1, along with the size of the executables when compiled under different optimization levels.

To profile any particular BAP or angr analysis, we executed the analysis 5 times sequentially. We recorded the running time of each trial, and then we calculated the average running time over all 5 of those time trials. During each of the 5 trials, we also queried the `procfs` for the process's resident-set-size (RSS) every 0.25 seconds. We then computed the following:

- RSS: We compute the average RSS for each trial. Then we compute the average RSS of those 5 averages.
- A.min RSS: We note the smallest (min) RSS that we observe for each trial, then the "Avg min RSS" is the average of those mins.
- A.max RSS: Like "Avg min RSS," but max.
- Min RSS: The smallest (min) RSS observed throughout all 5 trials.
- Max RSS: The largest (max) RSS observed throughout all 5 trials.

Table 1: Test binary sizes

Exe	`-g -O2`	`-g -O1`	`-g -O0`
Bison	1.9M	1.7M	1.1M
Gawk	2.7M	2.3M	1.3M
Gnuchess	1.6M	1.6M	1.6M
Grep	0.7M	0.6M	0.4M
Gzip	0.3M	0.3M	0.2M
Less	0.6M	0.5M	0.3M
Make	0.8M	0.8M	0.5M
Nano	0.8M	0.7M	0.5M
Screen	1.7M	1.5M	1.0M
Sed	0.5M	0.4M	0.3M
Tar	1.9M	1.7M	1.0M

We estimated each library's resource overhead by loading an empty C program into a new project. On average, BAP took a half second with a max resident set size (RSS) of 84MB, while angr took one second with a max RSS of 82MB.

2.4 Extracting and Using Control Flow Data

A basic requirement for analyzing or transforming code in any non-trivial manner is to get data and control flow information. For binary code, this can be complex, and both BAP and angr offer built-in support. In this section, we compare the CFGs and CGs recovered by each tool, and describe a CG-based analysis that we implemented in both BAP and angr as a comparison of their capabilities and performance.

2.4.1 Call Graphs

We compared BAP and angr's features for working with CGs in two ways. First, we developed a script to directly compare the CGs produced by each tool, and report here on their similarity. Second, we selected a CG-based program analysis from the literature and implemented it twice, using each tool as a library.

Comparing CG Accuracy

Both tools make it simple to recover a program's CG and output it in the DOT graph description language. We implemented a simple algorithm for comparing this output:

- Start with the program entry point of both graphs.
- Recursively fetch the reachable nodes from that point, excluding already seen nodes.
- Compare the reachable nodes at step n as sets between the graphs.

While the tools agree well on small examples, differences appear quite early in the CGs of larger programs. For example, we get around 6% difference one step below `main` in the CG for the `grep` executable, and the errors snowball at lower levels up to a significant fraction. The cause for these discrepancies is unclear, but may be related to disagreements between what the tools consider to be reachable function calls during CFG construction (see again [19]).

Implementing a CG-based Program Analysis

One common use of CFGs and CGs is to judge the similarity of two programs. As a basis on which to evaluate the usability and performance of each tool, we selected a well-regarded algorithm for estimating the similarity of two CGs [20] and implemented it both as a BAP plugin and as an angr script. This algorithm was designed for malware indexing, and was judged the best in a survey of several program comparison algorithms [21].

The goal of the algorithm we implemented is to efficiently compare two program call graphs and compute a similarity score between 0 and 1, where 1 indicates that the graphs are identical. In general precise graph matching is an NP-hard problem, and therefore infeasible on the size of graphs produced by realistic programs. Therefore, the algorithm instead computes an *approximate* matching of the two programs' functions, based on their degree and common neighbors. The algorithm has three main steps:

1. Extract the call graphs from the two programs.
2. Construct a matrix where each cell contains the "cost" of matching a function from one program to a function in the other. This cost is an approximation for how different the functions are: lower is more similar.
3. Using the cost matrix, find the matching of functions between the two programs which minimizes the overall cost. Compute a corresponding similarity score.

Implementation of this algorithm was mostly straightforward. Both BAP and angr provide convenient and standard interfaces to the call graph datastructure. One obstacle was that the BAP's plugin interface is designed to manipulate a single program at a time. However, BAP does support saving a program's `Project` data structure to disk. Thus, we designed our plugin to take one binary from the command line and compare with a previously saved `Project` structure. This limitation has subsequently been removed in newer versions of BAP.

For evaluation, we took 11 GNU applications of varying sizes and compiled them on two optimization levels (`-O0` and `-O1`). We used the analysis to compare the two versions of each program. Table 2 contains the results. A long dash indicates that the analysis did not complete within 35 minutes.

The results show that our BAP OCaml implementation runs approximately 15% faster than our angr Python implementation on average, despite constructing larger CGs. Profiling revealed that the running time in both cases is dominated by a standard graph matching algorithm that the analysis uses, and thus speaks more to differences in the efficiency of OCaml and Python code than to differences in BAP and angr. The running time scales with the size of the graphs (reported as a sum of the number of nodes and edges). Substantial differences in graph sizes are a result of the discrepancies in CG recovery described above, and the similarity scores computed by the algorithm also differed as a result.

Table 2: BAP and angr performance on call graph comparison algorithm

Exe	Time (secs)		Max RSS (Kb)		Graph size	
	BAP	angr	BAP	angr	BAP	angr
bison	1181	824	15182	16847	7717	6078
gawk	158	2004	20253	25680	5760	8661
grep	89	581	7184	7528	3339	4002
gnuchess	158	82	20253	10815	5760	868
gzip	58	162	7391	6122	2065	1706
less	113	—	3741	—	4142	—
make	313	552	15812	10440	4835	4436
nano	729	454	8060	10620	6500	4618
screen	699	964	12980	12054	7466	6094
sed	27.6	—	4536	—	2320	—
tar	—	1321	—	8139	—	6520

2.4.2 Control Flow Graphs

Another commonly-used feature of binary analysis tools is control flow graph (CFG) reconstruction.

CFG construction

BAP and angr's CFG construction tools differ in many ways. For example, by default, BAP provides a separate CFG for each function while angr provides an interprocedural CFG.

To better understand the footprint of BAP's and angr's CFG construction processes, we profiled BAP constructing a CFG, angr constructing a CFG using its `CFGFast` module, and angr constructing a CFG using its `CFGAccurate` module, for each of the 11 sample GNU executables. In each case, we executed the construction five times (in parallel) and calculated the average elapsed (wall clock) time and the average maximum resident set size.

Table 3 summarizes the time required to extract a CFG with each tool. Table 4 shows the memory usage of the CFG construction algorithms (average maximum resident set sizes), and Table 5 shows size of the constructed graphs.

Non-isomorphic CFGs

BAP and angr lift binaries to different intermediate representations (IRs), as mentioned in Section 2.1. BAP lifts to its own BAP intermediate language (BIL), while angr lifts to VEX IR, a language originally designed for use with the Valgrind binary instrumentation tool [22] As a result, BAP and angr do not produce CFGs with the same basic blocks or nodes. In particular, their respective IRs (especially BAP's) can contain special synthetic blocks and jumps that are unique to their particular representation.

For example, BAP expands the x86 instruction je to three nodes with two extra jumps: there is one node for the original je jump which jumps to two other synthetic nodes (one for the case where the zero flag is set, and one for the case where it is not). These synthetic nodes do not correspond to any address in the executable, nor do they correspond to any particular nodes in the CFG that angr constructs over its IR of the same executable. Thus, the CFGs that angr and BAP construct over the same executable are not isomorphic in any structural sense.

Table 3: CFG Construction: Time (seconds

Exe	BAP	angr fast	angr acc
bison	52	21	2924
gawk	141	36	2607
gnuchess	24	13	1966
grep	23	10	1341
gzip	8	5	411
less	14	8	1323
make	56	11	2184
nano	26	16	1595
screen	64	27	3226
sed	8	8	647
tar	71	29	2349

Table 4: CFG Construction: Memory usage (average maximum resident set sizes in megabytes)

Exe	Bap	angr fast	angr acc
bison	626	228	1615
gawk	1000	371	2497
gnuchess	450	199	1423
grep	288	163	680
gzip	188	125	562
less	229	157	849
make	419	183	1522
nano	338	193	1192
screen	613	281	2697
sed	1.4	138	748
tar	606	277	2315

Table 5: CFG Construction: Graph size

Exe	BAP		angr fast		angr acc	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
bison	15k	23k	26k	14k	83k	50k
gawk	23k	37k	58k	28k	172k	94k
gnuchess	11k	18k	21k	11k	66k	28k
grep	7k	11k	13k	7k	35k	21k
gzip	3k	5k	7k	4k	22k	13k
less	6k	10k	13k	6k	48k	28k
make	9k	15k	18k	9k	95k	56k
nano	10k	17k	20k	10k	56k	34k
screen	17k	26k	42k	21k	232k	129k
sed	4k	6k	9k	5k	62k	34k
tar	18k	28k	36k	18k	175k	101k

2.5 Value-Set Analysis

As an example of a standard, more complex use of a binary analysis toolkit, we experimented with value-set analysis (VSA) in both BAP and angr [23, 24]. The angr tools include an experimental Value Flow Graph (VFG) module that performs a VSA. It annotates the CFG with sets of values that registers and memory locations can take on at various points during execution. At the time of writing, BAP does not ship with a comparable module, so we implemented our own VSA plugin using BAP's built-in support for abstract interpretation. To compare the results and performance of the two implementations, we used them to discover jump destinations that vanilla BAP and angr CFG constructions missed.

2.5.1 Basics of Value Set Analysis on Control Flow Graphs

A value-set analysis (VSA) [23, 24] is constructed over an executable by calculating an over-approximated set of all possible values that each machine register and memory location can have at each program point in the executable. The possible values are machine words.

Both our BAP plugin and the existing angr pass use *abstract interpretation* to implement the VSA. Abstract interpretation is a standard technique to soundly approximate program behavior [25]. The core idea of abstract interpretation is to select an *abstract domain* that represents possible program values accurately but supports efficient approximations to program operations and permits the computation of fixed points to model program loops and recursion.

The set of possible values for any given variable or register can often be quite large, so the abstract domain of a VSA must represent these values compactly. Our VSA plugin for BAP uses *circular linear progressions* [26, 27]. The implementation found in angr uses an extension of *wrapped strided intervals* [28, 29, 24]. These two representations are similar, and the distinction made little difference for our purposes. This abstract domain is quite powerful in the realm of binary analysis:

- For numerical program values, like variables that have type `int` in the original source, program analysis can often determine that the value lies within some particular interval. For example, it may be easy to see that an index variable is restricted to the bounds of a loop. Strided intervals provide an exact representation of this range. CLPs improve on this representation by providing an accurate model for overflow in arithmetic computation on such values.

- Similarly, pointers often have a set of possible values that can be represented exactly. For example, if a particular pointer always points into a buffer, then it is precisely modeled by a strided interval where where the interval's lower and upper bounds are the beginning and end of the buffer and the "stride" is the alignment of the data fields in the buffer.

- If the value is something else, and has two or fewer possible values, then a strided interval can precisely model those values. Any more than two arbitrary values may not have an exact model as a strided interval or CLP, often resulting in a loss of precision. One common improvement is to combine strided intervals or CLPS with a domain of small finite sets.

2.5.2 Implementing and Comparing VSA in BAP and angr

Writing the VSA plugin for BAP felt straightforward. BAP's heavily documented interfaces make it fairly transparent how to traverse the lifted program and extract information from the various subroutines, blocks, and jumps. In addition, the programmer can attach arbitrary attributes to any IR term. These attributes always have the type of BAP's universal `Value` module, which is a generic data container, so the attributes can contain almost any sort of data structure. This provides a natural mechanism for decorating IR terms with pre- and post-conditions, as one does during a VSA analysis. The most difficult aspect of the implementation was the core mathematics of the CLP abstract domain—BAP itself was a pleasure to use.

The design of angr is geared toward human-driven analysis of binaries. The subset of the interfaces that a reverse engineer would employ to analyze a particular binary is easy to use and well documented. Here, our purpose was to implement a fully automated angr plugin that relied on little input from the user and provided a detailed report of relevant information. To accomplish this, we wrapped the existing VFG construction in a script that dynamically guides the analysis and prints the results. In general, we found the documentation and interfaces for this style of development more challenging to use.

For example, our angr VSA scripts required that we identify which registers were relevant for each basic block. By hand this is simple, but we found it challenging to find the interfaces needed to automatically identify registers in the documentation. Registers can be referenced in several ways, including intuitive names, IDs, and by index into an array of registers. It was typically unclear which variety of reference a function would require as input or return as output. As a result, development often required experimentation in interactive Python environments to understand interfaces, making it challenging to be confident that our code was correct or idiomatic.

To evaluate the two VSA implementations, we used them to resolve indirect jumps that BAP and angr CFG construction missed. We profiled runs on four small test programs that contain indirect jumps that require some insight to resolve. The results are in Table 6.

We found that our BAP VSA plugin resolved all jump targets, while angr's missed one in all but the last case. On further inspection, we found that angr's VFG module had a bug that causes it to discard the contents of previous value sets after successive iterations, thereby resulting in an under approximation. By stopping after each iteration, we were able to observe that angr actually resolved some of the missing jumps before discarding the results for the next iteration. We reported this bug, and it has since been fixed.

The BAP plugin runs faster, but uses more memory at a constant level for our toy programs, while

Table 6: Indirect jump resolution via VSA

Exe	Time (secs)		Max RSS (Mb)		Resolved Jumps	
	BAP	angr	BAP	angr	BAP	angr
Prog A	0.73	1.21	124	88	5 of 5 (100%)	4 of 5 (80%)
Prog B	0.72	1.59	124	91	8 of 8 (100%)	7 of 8 (88%)
Prog C	0.71	1.85	124	93	8 of 8 (100%)	7 of 8 (88%)
Prog D	0.70	4.20	124	104	8 of 8 (100%)	8 of 8 (100%)

angr runs more slowly, but uses less memory. Neither implementation scales well to larger programs. When run on the GNU utilities described in the previous section, we typically encountered issues ranging from memory exhaustion to unsupported constructs before the analysis completed.

As implementors, we found that BAP gave us more confidence in the VSA results than angr. The simple Python interface and VFG module in angr made it easy to get started and obtain initial results. However, the lack of documentation and the presence of apparent bugs made it difficult to verify the correctness of the analysis we built on angr's capabilities. By contrast, since BAP ships with no VSA, it was a fair amount of work to build our own. Nevertheless, BAP's module-based documentation and the static checking provided by its use of the OCaml type system gave us more confidence that we were using it correctly.

2.6 Survey Summary

Both BAP and angr enable analysis of binaries, providing a convenient interface that hides the technical details of the binary formats and ISAs. In addition, they each supply a suite of pre-built analyses to jump start the process.

We compared these tools in several ways. We described the process of implementing program analyses using them, and differences in the call graphs and control flow graphs they recover from binary programs. We implemented two representative program analyses using each tool, and examined their usability and performance.

In terms of resource usage, BAP is often more efficient, but not drastically so. We found that angr was easier than BAP to pick up quickly and begin experimenting with, and includes more-built in analyses. By contrast, BAP required us to do more work to get started, but its comprehensive module-based documentation gave us more confidence that we were using the tool correctly, even as new users.

Based on this experience, we selected BAP for use in the CBAT project.

3 CBAT Implementation and Internals

This section describes the implementation of the CBAT tools and research conducted in the course of their development. We cover:

- The core ideas of weakest precondition (WP) analysis (Section 3.1).
- Adapting WP for comparative analysis (Section 3.2).
- The implementation of key program analysis features in the context of weakest precondition analysis (Section 3.3).
- Integration with SMT solvers (Section 3.4).
- Our testing tools (Section 3.5).

3.1 Weakest Precondition Analysis

The core of CBAT is a *weakest precondition* (WP) analysis, originally introduced by Dijkstra in the 1970s [30]. In this section, we summarize the basics of WP and describe its implementation in CBAT. We focus on WP's traditional use as a single program analysis; Section 3.2 covers how it is adapted for comparative analysis. CBAT's WP implementation has two main phases: computing weakest precondition formulas for an SMT solver, and analyzing the results of the solver.

3.1.1 Computing Weakest Preconditions

WP is used to determine whether a program state of interest can be reached. This final state is a *postcondition*—a property that holds at the conclusion of the program. For example, we may be interested to know whether a particular assert can be tripped. A WP analysis walks backwards through each program statement constructing a formula that encodes what must be true *before* that statement for us to reach the state of interest. This formula is the precondition, as it holds before the program statement. If we are interested in knowing whether there is any way at all to reach the state of interest, we want to find the *weakest* such precondition—i.e., the one that places the fewest requirements possible on the initial state.

As a simple example, consider the following C code:

```
1   y = z * 2;
2   x = y + 10;
3   assert(x > 20);
```

Suppose we want to know what must be true at the beginning of the program for the assert to be tripped (where the assert will be tripped if the expression `x > 20` is false). We see the assert will be tripped if `x <= 20` before line 3. This becomes the postcondition of interest for the previous line (line 2). Then, we see that `x <= 20` will hold after line 2 if `y <= 10` before it. This then becomes the postcondition of interest for line 1, and we see that `y <= 10` after line 1 if `z <= 5` before it. So, `z <= 5` is the weakest precondition—the least we must know about the state at the beginning of the program to ensure that the assert is tripped.

In large, realistic programs, there are more statements and the effects of each may be complicated. This results in a weakest precondition that is too complex for a human to understand directly. It is typically not obvious whether the precondition is *satisfiable*, i.e., whether there is any possible initial state that makes the precondition true. So, program analysis tools based on weakest preconditions typically compute the precondition formula as a *satisfiability modulo theories* (SMT) problem compatible with automated SMT solving tools. In the case of CBAT, we use the standard SMT-LIB format and the Z3 and Boolector SMT solvers.

CBAT implements WP not for C code, but for BAP's intermediate language (BIL). An implementation of WP is essentially a *predicate transformer*—given a postcondition predicate and a particular program statement in BIL, CBAT transforms the predicate to obtain the appropriate precondition for the given statement. By implementing this transformation for all BIL statements, we obtain a weakest precondition analysis compatible with any binary format supported by BAP. This predicate transformer is implemented by the `visit_elt` function[2]:

```
val visit_elt : Env.t -> Constraint.t -> Bap.Std.Blk.elt
              -> Constraint.t * Env.t
```

This function takes three arguments:

- An `Env.t`, which contains various state, including summaries of subroutines (see Section 3.3.1) and a map between BAP variables and Z3 variables. Crucially, this environment also holds *hooks* for certain program constructs that are used to adjust the generated precondition. This feature is essential to enable CBAT's many different built-in properties and support for user-defined properties. As an example, the environment includes a hook that can be used to add additional clauses to the precondition for a memory access. CBAT's `--null-derefs` option, which checks a program for null dereferences, is implementing by setting this hook to add an assertion that the address in an access is not zero. Thus, `visit_elt` need only model basic program behavior that is part of checking any property, and can rely on the environment hooks for property-specific adjustments.
- A `Constraint.t`. This is the postcondition of interest, represented by a custom data type.[3] Rather than constructing SMT formulas directly, we build a tree representing the various constraints derived from different parts of the program, and then convert this to an SMT formula once the beginning of the program is reached. This data structure is useful to capture the relationship between the SMT formula and the program's behavior, and is used again later to interpret the results of the SMT solver. We discuss this datatype in more detail in Section 3.1.2.
- A `Bap.Std.Blk.elt`. This is the type of BAP intermediate language statements.

It has two results: the computed precondition as a `Constraint.t`, and an updated `Env.t`. Once an appropriate `Constraint.t` has been constructed for a given function, the CBAT function `eval`[4] is used to convert it to a Z3 expression suitable for solving:

[2] See `cbat_tools/wp/lib/bap_wp/src/precondition.ml`
[3] See `cbat_tools/wp/lib/bap_wp/src/constraint.ml`
[4] See `cbat_tools/wp/lib/bap_wp/src/constraint.mli`

```
val eval : ?debug:(bool) -> Constraint.t -> Z3.context -> Z3.Expr.expr
```

This function additionally takes an optional `debug` flag that causes useful information to be printed to the terminal, and a Z3 context, which is the main interface to the solver.

An Example

We illustrate the process of computing a weakest precondition with an example. Consider the following C function:

```
void example(uint64_t x) {
  assert (x != 0x42);
}
```

When compiled by GCC with the `-O3` optimization level for x86, we obtain the following machine code:

```
000000000000064a <example>:
 64a:   48 83 ff 42             cmp    $0x42,%rdi
 64e:   74 02                   je     652 <example+0x8>
 650:   f3 c3                   repz retq
 652:   48 83 ec 08             sub    $0x8,%rsp
 656:   48 8d 0d cb 00 00 00    lea    0xcb(%rip),%rcx
 65d:   ba 05 00 00 00          mov    $0x5,%edx
 662:   48 8d 35 ab 00 00 00    lea    0xab(%rip),%rsi
 669:   48 8d 3d ab 00 00 00    lea    0xab(%rip),%rdi
 670:   e8 ab fe ff ff          callq  520 <__assert_fail@plt>
```

As expected, this function begins by comparing its input (stored in register `rdi`) with `0x42`, and contains control flow that results in the assert being called only if they are equal.

BAP lifts this function to the following intermediate language code:

```
000008d1: sub example(example_result)
000008e5: example_result :: out u32 = RAX

0000031d:
00000328: #34 := RDI - 0x42
0000032b: CF := RDI < 0x42
0000032e: OF := high:1[(RDI ^ 0x42) & (RDI ^ #34)]
00000331: AF := 0x10 = (0x10 & (#34 ^ RDI ^ 0x42))
00000334: PF :=
          ~low:1[let $1 = #34 >> 4 ^ #34 in
                 let $2 = $1 >> 2 ^ $1 in $2 >> 1 ^ $2]
00000337: SF := high:1[#34]
0000033a: ZF := 0 = #34
00000345: when ZF goto %0000033f
000008d2: goto %0000068d

0000033f:
```

```
00000357: #36 := RSP
0000035a: RSP := RSP - 8
0000035d: CF := #36 < 8
00000360: OF := high:1[(#36 ^ 8) & (#36 ^ RSP)]
00000363: AF := 0x10 = (0x10 & (RSP ^ #36 ^ 8))
00000366: PF :=
          ~low:1[let $1 = RSP >> 4 ^ RSP in
                 let $2 = $1 >> 2 ^ $1 in $2 >> 1 ^ $2]
00000369: SF := high:1[RSP]
0000036c: ZF := 0 = RSP
00000374: RCX := 0x728
0000037c: RDX := 5
00000384: RSI := 0x714
0000038c: RDI := 0x71B
00000397: RSP := RSP - 8
0000039a: mem := mem with [RSP, el]:u64 <- 0x675
0000039d: call @__assert_fail with return %000008d3

000008d3:
000008d4: call @main with noreturn

0000068d:
00000694: #73 := mem[RSP, el]:u64
00000697: RSP := RSP + 8
0000069a: call #73 with noreturn
```

This function is somewhat longer than the assembly version, because BAP makes explicit each of the effects of complex x86 instructions. In particular, the x86 `cmp` changes many x86 flags, like the zero flag and overflow flag, which BAP represents explicitly by setting `ZF` (statement `33a`) and `OF` (statement 32e). The conditional jump instruction `je` indicates a jump that occurs only when the zero flag is 1, which we see made explicit in the BIL code in statements `33a` and `345`.

CBAT's WP implementation walks backwards over this representation using the previously mentioned `visit_elt` function to capture the behavior of each BIL statement. CBAT includes an option, `--show=precond-smtlib`, that can display the computed precondition in the SMTLIB format. The precondition computed for checking whether the assert can be reached in this function is:

```
(declare-fun RDI0 () (_ BitVec 64))
(declare-fun init_AF0 () (_ BitVec 1))
(declare-fun AF0 () (_ BitVec 1))
(declare-fun init_CF0 () (_ BitVec 1))
(declare-fun CF0 () (_ BitVec 1))
(declare-fun init_OF0 () (_ BitVec 1))
(declare-fun OF0 () (_ BitVec 1))
(declare-fun init_PF0 () (_ BitVec 1))
(declare-fun PF0 () (_ BitVec 1))
(declare-fun init_RAX0 () (_ BitVec 64))
(declare-fun RAX0 () (_ BitVec 64))
(declare-fun init_RCX0 () (_ BitVec 64))
(declare-fun RCX0 () (_ BitVec 64))
(declare-fun init_RDI0 () (_ BitVec 64))
```

```
(declare-fun init_RDX0 () (_ BitVec 64))
(declare-fun RDX0 () (_ BitVec 64))
(declare-fun init_RSI0 () (_ BitVec 64))
(declare-fun RSI0 () (_ BitVec 64))
(declare-fun init_RSP0 () (_ BitVec 64))
(declare-fun RSP0 () (_ BitVec 64))
(declare-fun init_SF0 () (_ BitVec 1))
(declare-fun SF0 () (_ BitVec 1))
(declare-fun init_ZF0 () (_ BitVec 1))
(declare-fun ZF0 () (_ BitVec 1))
(declare-fun init_mem0 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun mem0 () (Array (_ BitVec 64) (_ BitVec 8)))
(declare-fun |init_\|#340\|| () (_ BitVec 64))
(declare-fun |#340| () (_ BitVec 64))
(declare-fun |init_\|#360\|| () (_ BitVec 64))
(declare-fun |#360| () (_ BitVec 64))
(declare-fun |init_\|#730\|| () (_ BitVec 64))
(declare-fun |#730| () (_ BitVec 64))
(assert (let ((a!1
  (and (bvult (bvadd (bvsub #x0000000040000000 #x0000000000800000)
                     #x0000000000000080)
              RSP0)
       (bvule RSP0 #x0000000040000000)
       (= |#730| |init_\|#730\||)
       (= |#360| |init_\|#360\||)
       (= |#340| |init_\|#340\||)
       (= mem0 init_mem0)
       (= ZF0 init_ZF0)
       (= SF0 init_SF0)
       (= RSP0 init_RSP0)
       (= RSI0 init_RSI0)
       (= RDX0 init_RDX0)
       (= RDI0 init_RDI0)
       (= RCX0 init_RCX0)
       (= RAX0 init_RAX0)
       (= PF0 init_PF0)
       (= OF0 init_OF0)
       (= CF0 init_CF0)
       (= AF0 init_AF0)))
  (a!2 (bvnot (bvredor (bvxor #x0000000000000000
                        (bvsub RDI0 #x0000000000000042))))))
(let ((a!3 (=> and (ite (not (= a!2 #b0)) false true))))
  (=> (=> a!1 a!3) false))))
```

Putting these pieces together, we can see that even a very simple C function requires many assembly instruction to express, and results in a precondition that is difficult for humans to understand directly. The SMTLIB code begins by declaring the various variables used in the precondition, which appears in the `assert` clause. We can see in this case that the assert clause primarily sets up initial values of registers, and then performs a comparison involving `RDI` and `0x42`, as expected.

As these formulas are difficult to understand and verify by human inspection, we rely on an SMT solver to automatically check them. How this works exactly is the topic of the next section.

3.1.2 Solving WP Formulas and Analyzing the Results

CBAT relies on an SMT solver to analyze whether the precondition formula is *satisfiable*—i.e., whether there exists an instantiation of the formula's variables that makes it true. The variables in our formula represent the initial state of the system (memory and registers), and the formula is true only if some bad final state is reachable—i.e., if the property we are checking is false. Thus, if the SMT solver succeeds in satisfying the formula, we know the program violates the property of interest. If the SMT solver proves that the formula is unsatisfiable, we know that the program can never reach the bad final state of interest, and thus the correctness property in question holds.

In the satisfiable case, our goal is to report back to the user not only that the property is violated, but *how*. There are many pieces of information that will be useful to the user in debugging the violation. In particular:

- What program inputs cause the violation.
- What initial state of memory leads to the violation.
- What path through the program is taken when the violation occurs.

Continuing our example from the previous section, the final output from CBAT when checking whether the assert can be tripped is:

```
Property falsified. Counterexample found.

Model:
      ZF  |->  0x0
      SF  |->  0x0
      RSP |->  0x000000003f800082
      RSI |->  0x0000000000000000
      RDX |->  0x0000000000000000
      RDI |->  0x0000000000000042
      RCX |->  0x0000000000000000
      RAX |->  0x0000000000000000
      PF  |->  0x0
      OF  |->  0x0
      CF  |->  0x0
      AF  |->  0x0
      mem_orig |-> [
              else |-> 0x00]
```

Here, the `Property falsified. Counterexample found` at the beginning indicates that the formula is indeed satisfiable. That is, the final state of interest (the assert being tripped) can be reached. CBAT then outputs information about the *model*, which is the satisfying instantiation found by the SMT solver. Here we see that `RDI`, which carries the function's argument, is `0x42` as expected. We also see the initial values of state including various processor flags and registers, as well as the initial value of memory. In principle, the SMT solver could pick almost any values for these pieces of program state since they do not affect whether the assert is reached. In practice, the model often contains zeroes for values that are irrelevant, as it does here.

We can also use CBAT's `--show=paths` option to see the path through the program that results in the property violation. This program has only one branch—the if-then-else statement—as seen in CBAT's output when this option is enabled:

```
Path:
      00000345: when ZF goto %0000033f (taken) (Address: 0x64E:64u)
      #34 |->  0x0000000000000000
      ZF  |->  0x1
      SF  |->  0x0
      PF  |->  0x1
      OF  |->  0x0
      CF  |->  0x0
      AF  |->  0x0
```

As we saw in the previous section, the source-level comparison is implemented via the x86 zero flag. This output indicates that the zero flag branch was taken, and it shows us the value of relevant flags and program temporaries. Here we see the value of program temporary `#34` is 0. Examining the lifted BIL version of the program from the previous section, we can see that this temporary holds the difference between the function input and `0x42`. While this program is simple, realistic programs often have complex and nested branching patterns, and the `--show=paths` feature can be essential in understanding how a property violation occurs.

Of course, the relevant path through a program can be much more complex than the path we see here for this simple example. To make it easier to grasp the details of the path, the user can ask CBAT to print it as a `.dot` file which can then be loaded in any graph visualizer and rendered visually. In this way, the user can see a picture of the program's control-flow-graph, with the path of interest highlighted.

To display any of this information to the user, CBAT must analyze the model produced by the SMT solver and translate the instantiation of logical variables in the precondition formula back into terms understandable with respect to the original program. The process to display the original state of registers and memory is relatively straightforward. CBAT creates logical variables to represent each register, using the SMT theory of bitvectors so that each variable has exactly the range of values expected for the width of the register. We also create a single variable to represent the original state of memory as a flat array of bitvectors, using the SMT theory of arrays. When the SMT solver finds a model demonstrating satisfiability, it will contain concrete values for each of these variables. Different solvers have different interfaces to these models—we use the Z3 solver's interface, which makes it straightforward to query the value of specific variables.

Displaying the path through the program is more complex, since it is not directly represented by a specific collection of logical variables in the precondition formula. Instead, there are many intermediate variables that correspond to program states between each statement. CBAT must inspect these and identify a correspondence to the lifted BIL program's branch constructs to interpret what program path is taken.

We solve this problem with the use of an intermediate data structure, **`Constraint`**`.t` that is used both during construction of the precondition formula (as noted above) and while the counter model is being analyzed. This datatype is defined as follows:

```
type t =
  | Goal of z3_expr
  | ITE of Jmp.t * z3_expr * t * t
  | Clause of t list * t list
  | Subst of t * z3_expr list * z3_expr list
```

The first three constructors directly represent different kinds of preconditions.

- `Goal e` is the base case—it represents a specific concrete precondition encoded as a Z3 formula.
- `ITE (j,e,t1,t2)` represents a precondition arising from a branch in the program. The first argument records the branch location in the lifted BIL code, for use when the model is analyzed. The second argument contains the branch condition as a Z3 formula. The final two arguments include the constraints along both branches. By recording the encoded branch condition for each branch in the program, we make it possible to subsequently check which branches were taken and which were not. We can see which program state is relevant to that choice by checking which variables appear in the condition.
- `Clause (ts1,ts2)` represents a compound, hypothetical constraint. Its two arguments are a list `[a1,...,an]` of assumptions and a list `[v1,...,vn]` of verification constraints, and it represents the composed precondition `a1 /`
  `... /`
  `an ==> v1 /`
  `... /`
  `vn`. This is particularly useful to encode *relative* properties of two programs, where we prove some fact about a modified program's behavior contingent on information derived from the first program.

The final constructor, `Subst (t,e1,e2)`, is included as an optimization. It represents the substitution of the values in the list `e1` for the variables in the list `e2` in the constraint `t`. Substitutions of values for variables occur frequently while constructing weakest preconditions, and we have found that performing these substitutions eagerly can have a detrimental impact on performance due to explosion in the size of the precondition. Instead, we represent them abstractly using this constructor, and only perform the substitution when required to construct the final formula. This yields a much better user-experience. Traditional construction of weakest preconditions can often be slow on account of its complexity, but CBAT removes a good deal of the wait time by performing substitutions only at the end.

The logic to query Z3 based on this data structure and provide pretty printed output to the user for program state, paths, and other useful information is implemented by the following function[5]):

```
val print_result : ?fmt:Format.formatter ->
  Z3.Solver.solver -> Z3.Solver.status -> Constraint.t -> show:string list ->
  orig:Env.t * Bap.Std.Sub.t -> modif:Env.t * Bap.Std.Sub.t -> unit
```

[5]See `cbat_tools/wp/lib/bap_wp/src/output.ml`

The arguments to this function are:

- An optional formatter if you want to send the output somewhere other than stderr.
- The interface to the solver for use in querying the model.
- The solver's result—indicating whether the formula was satisfiable, unsatisfiable, or the solver timed out.
- The constraint construction datastructure we have just described.
- Configuration strings indicating what output the user has asked to see.
- The original and (if performing comparative analysis) modified program information.

3.2 Comparative Analysis with WP

The WP analysis, as described in the previous section, is a *single program* analysis—it is used to analyze the behavior of a specific given program in isolation. Of course, the overall goal of the CBAT project is to enable *comparative* analysis, where two programs are compared. In the context of the ONR TPCP program, many teams are building late-stage binary customization tools, and the role of CBAT is to check that the changes introduced by these tools do not introduce unintended changes in behavior, so that the modified binaries can be redeployed.

For comparative analysis, CBAT relies on a key insight from the literature: WP can be used for this purpose by *composing* the two programs to be compared into a single program. As a simple example, consider this program fragment:

```
int x = 3;
if (y) {
  x = 5;
}
```

A verification tool like WP can check various properties of this program, like "`x` < *10 after execution for every possible value of* `y`". Suppose we are given a modification of this program, say:

```
int x = 5;
if (y) {
  x = 7;
}
```

One might want to verify, say, that the second program will always compute a greater value of `x` than the original program does for any given `y`. To verify this fact by analyzing a single program, we build the following composed program:

```
int x1 = 3;
if (y1) {
  x1 = 5;
}
int x2 = 5;
if (y2) {
  x2 = 7;
}
```

Here, we have adjusted the names of all program variables so that the two programs do not conflict—they are essential separate parts of the composed program that run sequentially but do not interact. We can then check the following property for this single program:

> If `y1 == y2` when the program starts, then `x1 < x2` when the program ends.

If this property is true of the composed program, it implies a fact about the original two programs. In particular, it implies that the second program always ends with a greater value `x` than the original program does, if both start with the same value of `y`.

This technique can be adapted to handle procedures with function calls, e.g., by checking that identical functions are called with equal arguments, and assuming that they return identical results in that case.

This composition technique tends to shine when used with *sound* analysis techniques, which is the approach CBAT takes. For unsound techniques like fuzzing, because the composed program's input search space increases with the square of the original program's input search space, missed detections may become more frequent.

The code for building composed programs and using WP to compute preconditions that encode comparative properties lives in CBAT's `Compare` module.[6] The key top-level functionality is implemented by the `compare_subs` function, which computes an appropriate precondition for the composition of two lifted subroutines:

```
val compare_subs
    :  postconds:(comparator list)
    -> hyps:(comparator list)
    -> original:(Bap.Std.Sub.t * Env.t)
    -> modified:(Bap.Std.Sub.t * Env.t)
    -> Constraint.t * Env.t * Env.t
```

This function takes four arguments. The first two are lists of *comparators* that encode the post-condition to be checked and the hypotheses to be assumed. The `comparator` type is an abstract type defined by this module to encode comparative properties. The module also includes an API for constructing comparators, which covers common simple cases and has facilities for arbitrary relative correctness properties. The third and fourth arguments are the original and modified sub-routines to be compared. The function returns an appropriate constraint for SMT solving, using the type described in the last section, and updated environments for the two functions.

[6] See `cbat_tools/wp/lib/bap_wp/src/compare.mli`

3.3 Analysis Features

In this section, we describe how various program analysis features are implemented in our WP analysis and applied by a user.

3.3.1 Function Summaries

As described, CBAT's WP analysis is *intraprocedural*, meaning that it analyses a single function at a time. A key question for an intraprocedural analysis is how to model function calls in the function being analyzed.

CBAT offers two options: inlining and summarization. Inlining a function is to simply include the body of the called function for analysis at the call site. This is easy to understand and maximally precise, essentially making the analysis partially interprocedural. However, it has downsides, chief among which is performance. As WP's performance scales, in part, with the size of the program being analyzed, it is infeasible to simply inline every function encountered.

Summarization, therefore, is the primary technique for handling function calls in WP. The idea behind this technique is to use *function summaries*, which provide a high-level specification of the behavior of a called function, rather than looking at the actual body of called functions. This technique can offer much better performance than inlining because we separate the work of understanding the behavior of caller and the callee, reducing the size of the problems. Often, a summary is used that does not model the behavior of the function in full detail, further enhancing performance. For example, it may make sense to model a function which performs complicated calculations and stores the result in a register as simply storing an unknown arbitrary value in that register, if the details of the computation are not relevant to the property under verification.

More precise and customizable support for function summaries was a focus in CBAT's second phase, as we scaled to larger and more complex programs. Function summaries have been successfully used to verify, for instance, that a modified program calls a function with different arguments. In the remainder of this section, we describe the user-facing interface for function summaries and their implementation.

Using Function Summaries CBAT includes both a library of common summaries and facilities for defining custom summaries. Users can choose to summarize any number of called functions with a combination of these options using command line flags. If a function is encountered for which the user has not provided or selected a summary, a default summary is used. The default summary is based on the ABI of the platform, and says that the function may arbitrarily change the values of caller-save registers and that it does not change the value of callee-saved registers.

Users may apply CBAT's built-in library of function summaries with the `--fun-specs` flag. Commonly used summaries include:

- `chaos-caller-saved`, which assumes the function may arbitrarily change the values of caller-saved registers.
- `verifier-nondet`, which models non-deterministic functions—i.e., those whose output is not determined by their inputs. This is useful for functions whose result depends on some state that is not being modeled in full detail, as is often the case for memory management functions like `malloc`.

- `verifier-error`, which assumes the function will trigger an error and can be used to see if the function is reachable.

A complete list of built-in summaries can be found in Appendix B.

The built-in summaries can be convenient for quickly modeling a function with approximate behavior. To capture application-specific behavior and provide a more fine-grained model, the user can instead provide *custom summaries* for functions of interest. Custom summaries are provided with the `--user-func-specs` flag. For example, to give a summary to a single function `foo`, a user would write:

```
--user-func-specs="foo,<precondition>,<postcondition>"
```

Or, in cases where the function specs for the original and modified binaries need to be different, a user can specify distinct respective function summaries with the `--user-func-specs-orig` and `--user-func-specs-mod` flags.

The pre- and post-conditions are written in the same SMTLIB2 format used throughout the tool. The WP analysis will build a property that checks the precondition holds whenever this function is called, and assumes the postcondition holds after the call for the purposes of verifying the remainder of the function under analysis.

This simple interface is extremely flexible, allowing to model a wide variety of functions and scenarios. For example:

- Suppose a function `int div(int x, int y)` is used to compute the integer division of `x` by `y` in an x86_64 program. To check that the program can not attempt to divide by zero, the following summary could be used:

  ```
  div, (assert (RSI (_ bv0 64)))), (assert true)
  ```

  Here, the precondition checks that the second argument to the function (stored in the register `RSI`) is not equal to zero (the expression `(_ bv0 64)` is SMTLIB shorthand for a 64-bit bitvector with value 0). We use the trivial postcondition `(assert true)` because in this example we are interested only in the precondition.

- Suppose the program calls a function `int perfect_square(int x)`, which returns 1 if its argument is a perfect square and 0 otherwise. If we want to model the result of this function precisely, we could use the function summary:

  ```
  perfect_square, (assert true),
      (assert
        (ite (exists ((y (_ BitVec 64)) (= init_RDI (bvmul y y))))
             (= RAX (_ bv1 64))
             (= RAX (_ bv0 64))))
  ```

  Here we use the trivial precondition, because `perfect_square` may legally be passed any integer value. Our postcondition is a more complex "*if-then-else*" statement, written `ite` in

SMTLIB syntax. Intuitively, the meaning of this property is "If there is some integer y such that the input to the function equals $y * y$, then the result of the function is 1. Otherwise, the result is zero."

To model this property, the scrutinee of the "if" statement uses the existential quantifier `exists` to check whether there exists a 64-bit bitvector `y` such that the initial value of the function's first argument (`init_RDI`) is equal to the result of multiplying `y` by itself (`bvmul` is bit vector multiplication). On x86_64, the function's output will be stored in the register `RAX`, so our summary states that in the "if" statement's true case `RAX` will be a 64-bit vector with the value 1, and in the false case `RAX` will be a 64-bit vector with the value 0.

- As a final example, we consider a situation from our ongoing analysis of the RAZOR debloating tool developed by the TPCP team from Georgia Institute of Technology, which illustrates the flexibility of the function summary mechanism. RAZOR attempts to remove code paths from a binary based on an experimental analysis of what code is reached during execution of the program's test cases. One property we are interested in verifying about RAZOR is that the original and debloated programs behave identically *except* along code paths that were intentionally removed.

  To check this property, we somehow need to tell CBAT to "ignore" the behavior along the code paths that were intentionally removed. The way RAZOR works is that it inserts a call to an error function `error()` at the points where the program would branch to a removed code path. This is useful for RAZOR users, because the `error` function can print a helpful message to let them know they have reached removed code. It is also helpful for CBAT, as we can achieve the desired property with a simple function summary:

  ```
  error, (assert true), (assert false)
  ```

  This summary is simple, but has a slightly subtle meaning. Our goal is essentially to tell CBAT to stop verification when it reaches the `error` function, because we are interested only in verifying the pieces that were *not* removed. We achieve this with the postcondition `(assert false)`. This tells CBAT that after the `error` function is called, it may assume `false` while verifying the remainder of the code path. But assuming `false` is a contradiction, and therefore any property is trivially verifiable. Thus, when we use this function summary, WP will consider the code paths that reach the `error` function trivially verified, while still checking the correctness property for all other paths.

Implementation of Function Summaries

At initialization time, CBAT builds a specification for each function that might be called by the current function and stores it in the environment. These specifications have the type `Env.fun_spec`, which is defined as follows:

```
(** The type that specifies whether a fun_spec should calculate the
    precondition based off a summary or by inlining the function and
    visiting it with {!Precondition.visit_sub}. *)
type fun_spec_type =
  | Summary of (Env.t -> Constr.t -> Bap.Std.Tid.t -> Constr.t * Env.t)
  | Inline
```

```
(** Type that specifies what rules should be used when calculating
    the precondition of a subroutine. *)
type fun_spec = {
  spec_name: string;
  spec: fun_spec_type
}
```

We see that a `fun_spec` comprises the name of the function and a `fun_spec_type`, which is either the directive to inline the function when encountered, or a summary.

At a high level, the goal of a summary is to compute the appropriate precondition for a function given a postcondition. Recall that WP walks backwards through the program, so when it encounters a function call it will have computed some property representing the weakest precondition of the program after that point. That property becomes the function's postcondition, and the summary must give us an appropriate precondition to continue analysis. Summaries have the type:

```
Env.t -> Constr.t -> Bap.Std.Tid.t -> Constr.t * Env.t
```

The arguments are the current environment (an `Env.t`), the function's postcondition (a `Constr.t`), and the function's identifier (a `Tid.t`). The summary returns the precondition and an updated environment.

For our library of built-in summaries, these specification are simple functions defined directly in ocaml.[7] For user-defined function summaries, we must turn the user's specification into a function of this type. Recall that a user-specified summary has the form:

```
name, fun_pre, fun_post
```

Here, `name` is the name of the property being specified, while `fun_pre` and `fun_post` are SMTLIB formulas representing the function's precondition and postcondition. WP must create an ocaml function that takes the computed weakest precondition just after the function call, `wp_call`, and builds a new formula which both checks that `fun_pre` holds prior to the call and that `wp_call` will hold if `fun_post` is true after the function. Therefore, we define the `fun_spec_type` for user-provided function summaries to return the formula:

```
fun_pre /\ (fun_post => wp_call)
```

This is implemented by the function `user_func_spec` in `precondition.ml`, which also handles updating the registers and memory state that may be modified by the function call. These steps are crucially linked: for example, if the user mentions `init_RDI` in their specification, referring to value of the register `RDI` at the moment the function is called, `user_func_spec` will adjust this name to match up with the generated name representing the state of `RDI` at this point in the program.

[7] See `cbat_tools/wp/lib/bap_wp/src/precondition.ml`

3.3.2 Loop Invariants

Loops present a well-known, key challenge for static program analysis. In the general case, loops run for an unknown number of iterations, which may depend on user input and other program state. This uncertainty makes it infeasible or impossible for a static analysis to model the complete effects of a loop with full precision.

The most common solution to this problem, which we support in CBAT, is loop unrolling, a bounded model checking technique. The idea is to model a loop's behavior by iterating it some fixed, pre-determined number of times. This model may be inaccurate because the number of iterations of the loop in a real execution may differ from the number of unrolled iterations. However, this approach has the advantage of being straightforward to implement and has been shown in practice to be good for detecting many classes of errors.

But CBAT also supports a more sophisticated solution: invariant checking. Invariants are properties that are true before and after every iteration of the loop. For example, a loop that sums an array into an accumulator variant may respect the invariant that the accumulator holds the sum of the first i locations in the array, where i is a program variable incremented in the loop. This is true before the loop begins (when i is 0), and also after any number of iterations. Further, it implies that when the loop has finished executing the accumulator holds the complete sum of the array, because at that point i will equal the array's length.

Invariant checking is particularly tricky in binary analysis because of the lack of structured control flow. In a C program, the "beginning" and "end" of the loop are clearly marked in the syntax of the program. In a binary program, a loop is a cycle in the control flow graph, and may potentially have many entry points and exits. We use a graph analysis to find the entry and exit blocks, and our weakest precondition analysis checks that the invariant implies the postconditions of the exit blocks and is implied by the preconditions of the entry blocks.

This approach is implemented by the function `loop_invariant_checker` in `precondition.ml`. The user specifies the loop invariant as an SMTLIB string, and identifies the loop by one of its basic blocks. The `loop_invariant_checker` function confirms that this block corresponds to a loop using the Ocaml **`Graphlib`** library to find the strongly connected components of the function's control flow graph. A strongly connected component is a subgraph where every node is reachable from every other node—all loops have this property.

Once the strongly connected component containing the user-specified block is identified, we run WP on it as if it were a stand-along program, finding the weakest precondition at the entry block that must hold for the invariant to be true at the exit blocks. Finally, the computed property for the loop is that the invariant itself implies this weakest precondition—thus, we check that if the invariant holds at the entry points to the loop, it must hold at the exits.

We built a number of examples of using CBAT with loop invariants.[8] One important note for users is that recent features of Z3 make it much easier to define invariants with native support for recursive specifications. For example, in a specification for a loop that sums an array, we might use the following SMTLIB function to compute the correct sum:

```
(define-fun-rec sum_array ((start (_ BitVec 64)) (end (_ BitVec 64)))
```

[8] See `cbat_tools/wp/resources/sample_binaries/loop_invariant/`

```
                          (_ BitVec 8)
      (ite (= start end)
           #x00
           (bvadd (select mem start)
                  (sum_array (bvadd start #x0000000000000001) end)))
    )
```

This defines a function `sum_array` with two 64-bit parameters `start` and `end` representing the addresses of the beginning and end of the array. It sums all bytes from `start` to `end` and returns the resulting 8-bit value.

Unfortunately, the initially released support for such specifications in Z3 was buggy [9]. This bug has subsequently been fixed in Z3, so it is important to ensure one has the latest version when defining such invariant.

3.3.3 Analyzing Whole Programs

The uses of CBAT described to this point involve a single function or a single pair of functions. But end-users will sometimes be interested in applying WP to a whole program (or pair of programs).

To help a user analyze whole programs, we developed a script that collects information about the binary or binaries under test, automatically invokes CBAT with common parameters on every function or pair of corresponding functions, and displays summarized, intuitive results to the user. It also stores detailed analysis results to disk for further investigation.

The script is included with the CBAT release.[10] It offers a simple interface for the common case:

```
$ bash run.bash --help
bash run.bash [-j|--jobs] [-t|--timeout] [-o|--output] -- <original> <modified>

- jobs: How many jobs to run in parallel (default: 1)
- timeout: Timeout for each job (default: 1000s)
- output: Location of logs and results of each subroutine
          (default: output-<date>)
```

The scripts is also easily editable to fine-tune WP parameters. When invoked, it prints status updates followed by a brief results summary. For example, here are the results from one of the examples discussed in Section 4.

```
  Comparing 'brittle-tar/tar' and 'brittle-tar/tar.compact.exe'.
  ----------------
  SATs: 74
  UNSATs: 1208
  UNKNOWNs: 229
  TOTAL: 1511
  ----------------
  Elapsed time: 06:13:16
```

This shows the number of counter examples found (SATs), the number of functions verified (UNSATs) and the number of tests that timed out with the provided time limit (UNKNOWNs). The

[9] See https://github.com/Z3Prover/z3/issues/5305
[10] See `cbat_tools/wp/scripts/run.bash`

detailed analysis results for each function examined are stored in individual files at the user's specified output location.

3.4 SMT Solver Integration

As discussed above, CBAT's WP analysis relies on an SMT solver to check whether the computed weakest precondition formula is satisfiable. Many open source SMT solvers exist, ranging from academic experiments to robust tools used in production environments. By default, CBAT uses Z3, an open source SMT solver from Microsoft [31].

Z3 is the most mature and widely used SMT solver. It offers excellent performance on a wide variety of problem types. However, alternate SMT solvers, primarily from academia, often offer better performance on certain classes of formulas. Here, we describe a novel technique we developed for integrating CBAT with alternate solvers (Section 3.4.1) and our performance results with one such solver, Boolector (Section 3.4.2).

3.4.1 Novel integration path for alternate solvers

Obstacles to seemlessly swapping solvers come from two key points of interaction between our weakest precondition analysis and the solver: passing the solver a formula to check and analyzing the results. At the first interaction point, the formula itself, we use the solver-agnostic SMT-LIB format. Sometimes CBAT uses by default certain SMT-LIB features that are not supported by every solver, in order to cleverly encode certain cases. However, when using CBAT with a solver other than Z3, these optimizations can be turned off, in which case CBAT will use the more standard encoding of program behavior.

The second interaction point, analyzing solver results, is more challenging. In the cases where the solver refutes the formula, it produces a "counter model" that shows a concrete falsifying instantiation of the logical variables in the formula. This model encodes a way in which the program can violate the property being checked, but CBAT must do quite a bit of work to "interpret" this encoding and show the user what goes wrong in terms of the original program, as described in Section 3.1.2. This stage of CBAT heavily depends on a Z3-specific library for examining counter models and performing additional queries based on an existing counter mode, because it would be a substantial undertaking to rewrite this code for each new solver. But how then does one "interpret" the results of a non-Z3 solver, using a Z3-specific library?

Our novel approach is to use a *combination* of Z3 and a second solver in a way that retains the potential performance benefits of the alternate solver, yet avoids the need to rewrite our counter model analysis code for it. The approach has four steps:

1. We first call the alternate solver to check the formula.
2. In the case that the alternate solver found a counter model, we have it output this model in a solver-agnostic SMT-LIB S-expression format. This output provides concrete values for the formula's logical variables that show how it can be falsified.
3. We then *also run Z3 on the original formula*, but add additional assertions setting the logical variables to the values found by the first solver. This serves as a check that the first solver found a valid counter model, but, more importantly, it allows Z3 to find that same countermodel instantly.

4. We then use the existing Z3 integration in CBAT to translate the counter model back into information about the original program for display to the user.

In cases where an alternate solver is faster than Z3, this approach offers the performance benefits of that solver but still allows us to use Z3's tools for manipulating the counter model. Because Z3 is provided variable instantiations from the alternate solver's counter model, it is able to reconstruct the counter model instantly.

3.4.2 Evaluating Boolector

Boolector is an SMT solver from the academic community that has shown promising results in SMT competitions [32]. We evaluated how CBAT performed with Boolector compared to Z3.

Using an example of verifying the equivalence of each function in a lifted and recompiled GNU tar binary, we found that Boolector is substantially faster than Z3 on average, but that Z3 is faster on some functions.

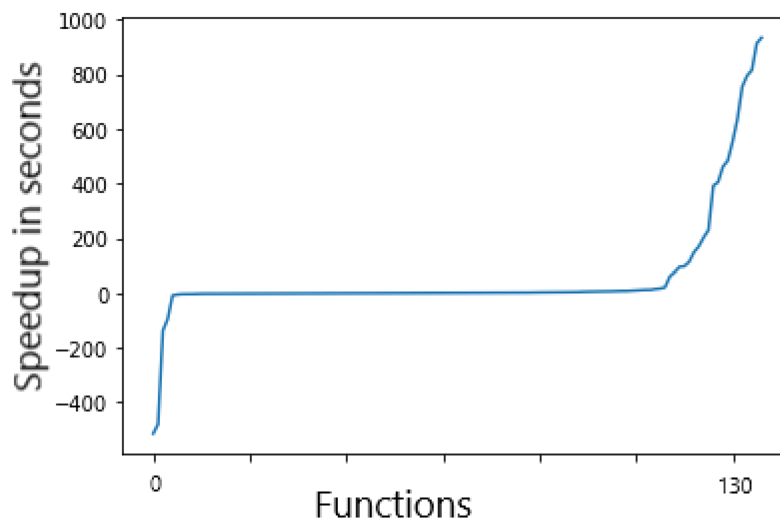


Figure 4: Per-function speedup offered by Boolector over Z3 on sample of 130 randomly selected functions.

Figure 4 shows the speedup in seconds offered by Boolector over 130 randomly chosen functions from this binary. We see that, for most functions, the speed difference is near zero. However, Z3 is substantially faster on some functions (the negative numbers) and Boolector is substantially faster on others (the positive numbers). Boolector offers a substantial performance boost on average—only four functions are 20s slower with Boolector, while 21 functions are more than 20s faster with Boolector. This includes nine functions where Z3 timed out with a 1000s limit, but Boolector was able to find a solution.

3.5 Testing and Debugging Tools

3.5.1 BIL DB

It is often useful to interactively explore the behavior of a lifted binary using CBAT's BIL DB tool. We use this tool in many circumstances, including when exploring the behavior of a new program, to observe the behavior that leads to a property violation found by WP, and to test binaries in

circumstances where WP times out. In this section, we describe the BIL DB implementation, and its interface to WP. The BIL DB implementation is found at `cbat_tools/bildb`.

BIL DB runs as a Read-Eval-Print Loop (REPL) that accepts input from the user and incrementally executes the lifted program, providing feedback about the results and allowing the user to inspect machine state at any moment. It is built on BAP's microexecution engine, Primus. Microexecution is the ability to execute code fragments in a virtual environment without user-provided input ([33]). Primus simulates the behavior of the underlying machine, and supports abstract machine state for unknown program data. BIL DB crucially relies on Primus to represent machine states[11] and to execute lifted BIL statements.[12]

BIL DB also exploits Primus's lightweight forking to keep a copy of every state it moves through. The user can then move both forwards *and* backwards through a program's execution, since under the hood, BIL DB can simply move forwards or backwards to the relevant copy of machine state. This is a very useful and somewhat unique feature in a debugger. When the user encounters an error or crash state, they do not need to start over from the beginning of the program as they would have to do with, say, GDB. Instead, they can just back up a step or two, and then try to figure out what led to the error.

To run a program with BIL DB, invoke it like this:

```
bap /path/to/exe --run --bildb-debug
```

In essence, you start BIL DB by telling bap to "run" the program with Primus. By adding the `--bildb-debug` flag, you trigger the debugger.

When BIL DB starts up, it prints some information about the architecture, and then it stops at the first instruction. For example:

```
BIL Debugger
Starting up...

Architecture
Type: x86_64
Address size: 64
Registers:
R10    R11    R12    R13    R14    R15    R8     R9     RAX    RBP    RBX
RCX    RDI    RDX    RSI    RSP    YMM0   YMM1   YMM10  YMM11  YMM12  YMM13
YMM14  YMM15  YMM2   YMM3   YMM4   YMM5   YMM6   YMM7   YMM8   YMM9

Entering subroutine: [%000007c6] _start
Entering block %000000df
000000ea: RBP := 0
>>> (h for help)
```

You are next prompted to enter a command for the debugger. To step to the next instruction, type `s` (for "step") and hit `enter`. You will see the next instruction, folowed by another prompt:

[11]See `cbat_tools/bildb/lib/state.mli`
[12]See `cbat_tools/bildb/lib/cursor.mli`

```
000000ed: AF := unknown[bits]:u1
>>> (h for help)
```

You can repeat the last command you typed by hitting `enter`. So, to step again, hit `enter`. That will take you to the next instruction:

```
000000f0: ZF := 1
>>> (h for help)
```

To see more instructions at a time, say the nearest 5 instructions before and after the current one you're looking at, type `show 5`, and hit `enter`. You will see something like this:

```
   %000000df
   000000ea: RBP := 0
   000000ed: AF := unknown[bits]:u1
-> 000000f0: ZF := 1
   000000f3: PF := 1
   000000f6: OF := 0
   000000f9: CF := 0
   000000fc: SF := 0
>>> (h for help)
```

If you prefer to alway see the nearest, say, 5 instructions before and after, type `always show 5` and hit `enter`. To return to seeing no extra instructions beyond the current one, type `always show 0` and hit `enter`.

To set a breakpoint, e.g. at the TID `000000f9`, type `b %000000f9`, and hit `enter`. It will tell you that you've set the breakpoint:

```
Breakpoint set at %000000f9
>>> (h for help)
```

After setting a break point, try looking at the nearest +/- 5 instructions by typing `show 5`:

```
   %000000df
   000000ea: RBP := 0
   000000ed: AF := unknown[bits]:u1
-> 000000f0: ZF := 1
   000000f3: PF := 1
   000000f6: OF := 0
 b 000000f9: CF := 0
   000000fc: SF := 0
>>> (h for help)
```

You can see the `b` next to TID `000000f9`, which indicates that there is a breakpoint there. To see all the breakpoints, type `breaks`, and hit `enter`.

To remove a breakpoint, e.g., the one just set, type `clear %000000f9` and hit `enter`.

```
Breakpoint cleared at %000000f9
>>> (h for help)
```

To skip to the next breakpoint (or the next basic block, whichever comes first), type `n` (for "next"), and hit `enter`. You will see that you have moved to, say, TID `000000f9`, where we set a breakpoint:

```
000000f9: CF := 0
>>> (h for help)
```

To move backwards one instruction, type `-s` ("minus s" for "skip back") and hit `enter`. You will see that you have moved back one step, for instance to TID `000000f6`:

```
000000f6: OF := 0
>>> (h for help)
```

To move all the way back to the nearest breakpoint or basic block (whichever comes first), type `-n` ("minus n" for "back to next") and hit `enter`. If there are no other basic blocks or break points in the program before we get back to the beginning, BIL DB will stop back at the first instruction:

```
At program start
000000ea: RBP := 0
>>> (h for help)
```

To see the value of a register, type `p RAX` ("p" is short for "print"). You will see the value printed out:

```
RAX    : 0
>>> (h for help)
```

To set RAX to some particular value, for example `0xabc`, type `set RAX=0xabc`, and hit `enter`. You will see that it has updated RAX with the new value:

```
RAX    : 0xABC
>>> (h for help)
```

To see the byte stored at a memory address, say `0x3fffff1+`, type `p 0x3ffffff1` and hit `enter`. You will see the value stored there:

```
0x3FFFFFF1: 0xD9
>>> (h for help)
```

To see, say, the next 4 consecutive bytes stored in memory starting at `0x3fffff1`, type `p 0x3ffffff1 4` and hit `enter`. You will see each byte at each address printed out:

```
0x3FFFFFF1: 0xD9  0x3FFFFFF2: 0x9E  0x3FFFFFF3: 0x7F  0x3FFFFFF4: 0x4C
>>> (h for help)
```

To store a byte at an address, e.g., to store `0x44` at `0x3ffffff1`, type `set 0x3ffffff1=0x44` and hit `enter`. You will see the new value:

```
0x3FFFFFF1: 0x44
>>> (h for help)
```

To see the full list of commands available to you, type `h` and hit `enter`. This will diplay the help menu. To quit at any time, type `q` and hit `enter`.

If you want BIL DB to initialize certain variables and memory locations before it starts running through the program, you can create an init file with this format:

```
Variables:
  R8: 0x0000000
  R9: 0x7fffffff
  RAX: 0x00000abc
  RCX: 0x00000000
  RDI: 0x00000000
  RDX: 0x00000000
  RSI: 0x00000000
Locations:
  0x3ffffff1: 0x00000012
  0x3ffffff9: 0x000000ab
```

Save it as `init.yml`. Then start BIL DB with `--bildb-init=init.yml`, like this:

```
    bap /path/to/exe --run --bildb-debug --bildb-init=init.yml
```

When BIL DB starts up, it will then print a message informing you of which variables and memory locations it initialized:

```
BIL Debugger
Starting up...

Architecture
Type: x86_64
Address size: 64
Registers:
R10    R11    R12    R13    R14    R15    R8     R9     RAX    RBP    RBX
RCX    RDI    RDX    RSI    RSP    YMM0   YMM1   YMM10  YMM11  YMM12  YMM13
YMM14  YMM15  YMM2   YMM3   YMM4   YMM5   YMM6   YMM7   YMM8   YMM9

Initialized state
Variables:
RSI    : 0            RDX    : 0            RDI    : 0            RCX    : 0
RAX    : 0xABC        R9     : 0x7FFFFFFF   R8     : 0
```

```
Locations:
0x3FFFFFF9: 0xAB   0x3FFFFFF1: 0x12

Entering subroutine: [%000007c6] _start
Entering block %000000df
000000ea: RBP := 0
>>> (h for help)
```

You can see from the output that it set `RAX` to `0xABC`, and `R9` to `0x7FFFFFFF`, just as was specified in the init file. Also, the memory locations are initialized too, in accordance with what was specified in the init file.

This ability to start BIL DB with a particular state enables a key integration between BIL DB and WP. Often, when WP finds a counterexample to a given property, it is helpful to interactively observe the problematic program execution. But, instead of having to manually configure the registers and memory in BIL DB to witness the problem, you can have WP automatically generate the countermodel as a BIL DB init script, and then you can just start BIL DB with that initialized state. This functionality of WP is implemented by the `output_bildb` function in WP's `Output` module[13]:

```
  val output_bildb :
    Z3.Solver.solver -> Z3.Solver.status -> Env.t -> string -> unit
```

Note that this function has a similar interface to the `print_result` function described in the previous section. This makes sense, as its purpose and implementation are similar. The purpose of `print_result` is to show the user the the initial program state that leads to the property violation, while the purpose of `output_bildb` is to provide the same information to BIL DB. Both functions are implemented by identifying the logical variables that correspond to initial program state and querying the SMT solver to find their values in the generated model. The `print_result` function then pretty-prints this information to the terminal for a human, while the `output_bildb` function prints it to a YAML file for consumption by the BIL DB tool.

CBAT also includes integration with GDB for a similar purpose. BIL DB is the recommended path for exploring the behavior of a discovered property violation, because microexecution can simulate any architecture and does not require the host machine to be able to execute the binary under analysis. However, in the case that the binary being analyzed and the machine doing the analysis are compatible, it can sometimes be useful to explore the behavior of the original machine code program itself. Support for this is implemented by the function `output_gdb` in the same module:

```
val output_gdb :
     Z3.Solver.solver -> Z3.Solver.status -> Env.t -> func:string
  -> filename:string -> unit
```

This function outputs a script at the given location that can be loaded into GDB to initialize the machine state. Its implementation is similar to `output_bildb` and `print_result`, except that it

[13]See `cbat_tools/wp/lib/bap_wp/src/output.mli`

must take an additional step. Where those functions compute initial program state in terms of BAP's intermediate language, `output_gdb` must translate this back down to the native architecture level.

4 Evaluating and Improving CBAT

In this section, we discuss the evaluation of CBAT's weakest precondition analysis (WP) in the context of application to modified binaries constructed by other TPCP performers. We begin by discussing metrics for evaluation in Section 4.1. We then look at a two case studies where we applied CBAT to examples provided by other performers and evaluate CBAT, our ability to improve it, and the generality of those improvements (Section 4.2).

4.1 Evaluation Criteria

There are three main criteria for evaluation of tools like CBAT: performance, false positives, and false negatives. We consider each in detail and discuss our evaluation approach.

4.1.1 Evaluating Performance

Static program analysis algorithms like WP are computationally expensive, typically requiring substantial CPU time and memory consumption. Binary analysis problems are particularly challenging for reasons including the requirement to model processor state in great detail to accurately capture behavior at the assembly level, and the way binary programs explicitly represent control flow information as data on the stack.

These factors can easily lead to state space explosions that exhaust available time or memory, and prevent tools from analyzing realistic programs. As such, designing algorithms that can scale effectively in terms of program size and property complexity are key goals for the CBAT project.

In the case of WP in particular, both computing the precondition formula and solving it with an SMT solver are potentially expensive, and must be optimized:

- To achieve performance in computing the precondition formula, we carefully select of intermediate data structures and program representations. For example, as described in Section 3.1.2, we found that substitutions of complex expressions for formula variables can explode the size of the formula. This makes subsequent manipulations progressively more time-consuming. We mitigated this by holding substitutions as abstract operations until the final formula is needed.

- To ensure the SMT solver will be be able to solve our formulas with acceptable time and memory requirements, we continuously measured its performance and experimented with alternate encodings of program behavior. For example, we initially represented an equality test operation with an SMT if-then-else construct that returned 1 if the two values are equal and 0 otherwise:

```
let binop (ctx : Z3.context) (b : binop)
      : Constr.z3_expr -> Constr.z3_expr -> Constr.z3_expr =
  match b with
  ...
  | EQ -> fun x y ->
              Bool.mk_ite ctx (Bool.mk_eq ctx x y) one zero
  ...
```

  However, we later discovered that the same logic can alternatively be encoded by taking the

bitwise exclusive or of the operands and using a vector reduce operation to check whether any bits remain 1:

```
let binop (ctx : Z3.context) (b : binop)
    : Constr.z3_expr -> Constr.z3_expr -> Constr.z3_expr =
  match b with
  ...
  | EQ -> fun x y ->
          BV.mk_not ctx @@ BV.mk_redor ctx @@ BV.mk_xor ctx x y
  ...
```

Experimentation determined that this alternate encoding results in a speed boost when checking forumlas with Z3, our most frequently used solver.

As a representative example, we consider a transformed binary provided to us by another TPCP project—RetroWrite [34]. The RetroWrite team took an instrumentation transformation they use for integration with the American fuzzy lop (AFL) fuzzing framework and applied it to the Linux `base64` utility. One potential risk in such a transformation is that the added function call code will change register values that are used by the original program, so we used CBAT to prove that functions in the modified program result in the same register values as the original.

The results of this experiment, including performance analysis, are summarized in Figure 5. We ran CBAT's comparative WP-based analysis using the property described in the previous paragraph and a 10 minute time out. As shown, 138 of 145 functions complete SMT formula construction in this time (95% success), and 129 of the remaining 138 additionally complete SMT solving in this time (93% success). This demonstrates that CBAT is sufficiently performant to handle checking useful properties on many real-world functions. The results of the SMT solving step (satisfiable or unsatisfiable) are further discussed in the next section.

Our experiments have found that, as expected, time to generate a precondition formula is closely correlated with function size—large functions take longer. The runtime scales superlinearly, due to the increasing size of intermediate data structures and the potential for explosion in the size of the formula itself with complex control flow. In future work, we intend to investigate techniques to break functions up into discrete chunks that can be analyzed individually, which has the potential to dramatically improve CBAT's ability to scale due to the superlinearity.

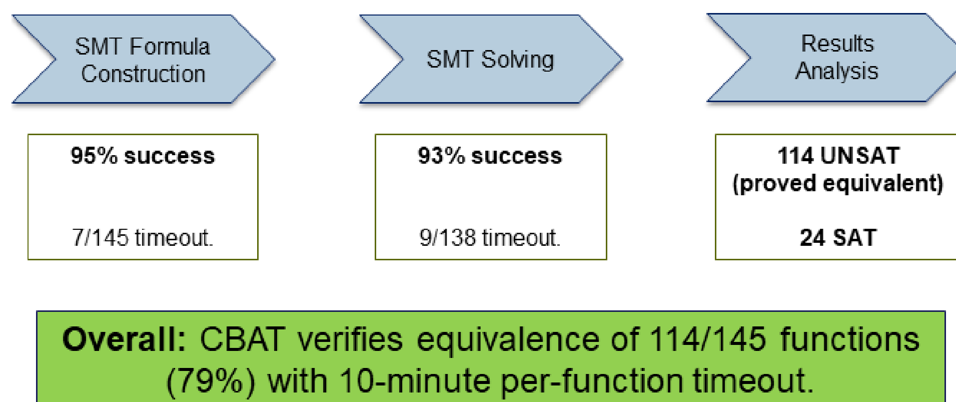


Figure 5: Summary of results on RetroWrite-transformed `base64` binary.

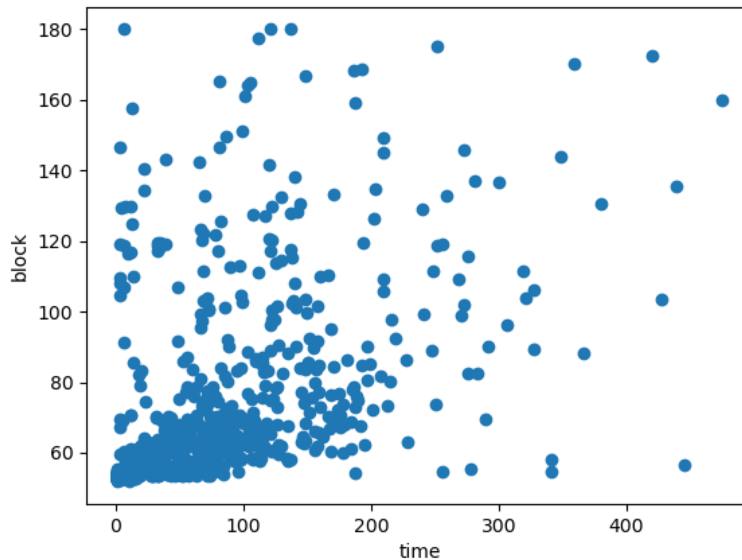


Figure 6: Overall CBAT WP runtime does not closely correlate with function size.

On the other hand, SMT solver time is *not* closely correlated with the size of the functions, and thus neither is WP's overall runtime. Figure 6 shows the (lack of a) relationship between function size (in basic blocks) and solving time over several hundred functions from an example provided by GrammaTech. We also found that other common program complexity metrics, like branching factor, are not closely correlated with solving time. One hypothesis is that, when performing *comparative* analysis, the complexity of the formula may scale with the amount of change between the two functions rather than their size.

We also experimented with integrating different SMT solvers to improve performance, described in more detail in Section 3.4.

4.1.2 Evaluating False Positives

A "false positive" is a situation where a tool reports that a program violates a property, but the program is in fact safe. False positives are inconvenient because they typically require a human to investigate the reported issue, but they do not represent a potential safety risk—they represent a tool erring on the side of caution, rather than the alternative. In practice, it is computationally infeasible for an automated analysis to quickly analyze complex properties of large binary programs with complete fidelity to the underlying machine. Tools like CBAT can select safe model simplifications to gain speed and handle larger programs at the cost of some false positives.

As a simple example, suppose we are analyzing a fragment of code that calls another function. A fully faithful model of program behavior would include a complete model of the behavior of that function, as well as all functions it calls, and so on. However, this can be prohibitively expensive, resulting in a model that is too complex to analyze in finite time. As an alternative, one can use a lightweight analysis to determine what system state can be affected by the function, and assume the function changes that state to hold abstract, unknown values.

This behavior is simple and does not complicate the model. It is also safe, because any possible behavior of the function is covered. However, it may result in false positives, in the case where some possible values cause a property violation, but these values never occur as a result of calling

the real function. Much of the work of a project like CBAT is exploring and evaluating these kinds of tradeoffs, with the goal of reaching a sweet spot where the analysis is fast enough to run in the available time and results in few enough false positives that they can be manually examined.

Figure 5 above provides a representative example of CBAT WP's false positive rate. In this case, the SMT solver provided an answer for 138 of the binary's functions. Of these, it proved that the property being checked holds for 114 of the 138 (83%) and reported a property violation for 24 of the 138 (17%). These 24 reported violations are either genuine bugs in the transformation or false positives. We have manually inspected the majority of them, all of which are false positives.

A core goal for the CBAT project is to achieve a false positive rate sufficiently low that a human and traditional testing tools can check the remaining portions of the binary. In Section 4.2, we describe this process in detail, covering six months of improvements that substantially reduced our false positive rate on a key example.

4.1.3 Evaluating False Negatives

A "false negative" is a situation in which a program can violate a property, but a tool incorrectly reports that the program is safe. Our goal is for WP to be *sound*, which means that it does not have false negatives. Soundness is an important goal, as it means that when a tool reports a program as safe, that result can be trusted.

There are two potential sources of unsoundness: simplifications in a tool's model of program behavior, often made for performance, and bugs in the tool's implementation. A related, but distinct, question that often arises in the area of security is to understand the attack model that a verification tool defends against. For example, even a tool with a completely faithful representation of architectural behavior might admit Spectre and Meltdown style attacks [35, 36], because these depend on microarchitectural implementation details. Similarly, even a verification strategy that fully modeled the microarchitecture might not be able to verify absence of Row Hammer [37] or physical attacks where radiation is use to flip bits. In the case of CBAT, we do not model microarchitectural or physical features, and instead aim for a faithful model of the architecture specification, via BAP's translation of the assembly instructions to its intermediate language. That is, the goal for CBAT comparative analysis is to detect any unintended change in behavior that can be observed with specific function inputs in any compliant processor.

Our experience with CBAT confirms the literature's findings that, for binary analysis, some unsound simplifications are necessary to achieve acceptable performance in an automated analysis [38, 39]. We chose standard simplifications that we believe are unlikely to result in common false negatives. However, this is challenging to empirically evaluate, as such an evaluation requires an "oracle" that can report when a tool misses a potential violation of the property being checked.

While the existence of potential unsoundness in any tool is a cause for concern, one useful perspective is to compare formal verification with other testing and validation approaches. A key observation is that any verification regime makes assumptions. Traditional software testing, for example, relies on assumptions including that the test environment matches the deployment environment, that the test cases cover all possible behaviors, and that all possible requirements and failure modes have been captured in the tests. One advantage of formal verification, even with potential false negatives, is that our assumptions are well defined. With other approaches, it is impossible to provide assurance that the assumptions made by the verification regime have been

captured, while with formal verification we can understand mathematically the possible scope of failures, and can treat these as risks to be analyzed and quantified or discharged with other approaches.

To this end, we documented the assumptions in our WP model[14], and provided flags to the user to adjust these assumptions. We summarize the key assumptions here, and discuss how they may be eliminated or reduced in future work:

- Reliance on BAP: CBAT assumes that BAP correctly lifts binaries to intermediate language programs with the same semantics, and that the datastructures it computes (like call graphs and control flow graphs) are accurate. BAP generally does a good job in this respect, and has the advantage over other tools that its intermediate language is formally specified, so we can be sure our tools correctly capture its behavior. However, the problem of control-flow and call graph reconstruction is undecidable in the general case, and there are some areas where BAP and other tools have known deficiencies, including function identification and indirect control flow recovery [17].

  There remains substantial "low hanging fruit" in improving binary lifting, like targeted analyses for common sources of indirect control flow. It is also possible to go further: using interactive verification techniques to prove the correctness of a binary lifter itself would be an exciting and important research project. Because BAP is written in OCaml, which is compatible with the Coq theorem proving environment, it is a natural target for this approach.

- Loops: CBAT's default model of loops unrolls them a finite, configurable number of times. This has the potential to miss changes that occur only after a greater number of loop iterations. Loops are a traditional challenge for automated analysis tools, and unrolling is a standard "low-cost" approach.

  In Phase 2 of the project, we extended CBAT with suppor for verifying *loop invariants*, which are properties that apply after an arbitrary number of unrollings and thus support sound reasoning about loops with unknown bounds. The implementation of this feature is discussed in Section **??**.

  In the present implementation, users must supply invariants for any loops in the program to use this feature. CBAT could be improved by adding support for *inferring* invariants. This is challenging and undecidable in the general case, but there is an increasing body of literature suggesting approaches that handle many programs and may be applied for binary analysis. An empirical evaluation of these approaches in a realistic setting like CBAT would be informative.

- Function calls: By default, CBAT implements an intra-procedural analysis; it does not descend into called functions. To handle function calls, CBAT supports *function summaries*, which provide specifications for known functions, and uses a default summary for unknown functions. The default summary assumes functions may arbitrarily modify registers, but does not attempt to model memory effects—a potential unsoundness.

[14] See `cbat_tools/wp/plugin/design_decisions.md`

In Phase 2 of the project, we have substantually improved our support for user-specified summaries with several new related features (see Section 3.3.1). Still, there are several ways this could be improved upon. For example, we could automatically compute more accurate summaries for unknown functions by using a light-weight analysis to bound their memory effects. We could iteratively apply CBAT's WP analysis to generate these summaries, repeatedly analyzing functions impacted by updated summaries until a fixed point is reached. We could also expand the built-in function summaries to completely cover widely used standard functions in libraries like `libc`. CBAT also includes an option to *in-line* the bodies of particular functions rather than summarizing them. This approach is sound, but appropriate only for small called functions, due to issues of scale.

4.2 Applying and Improving CBAT: Example 1

As described in the previous section, the development process for CBAT has involved using various examples provided by other TPCP performers as test cases for CBAT's performance and its false positive rates. Typically, we will work with a performer to understand the transformation applied, and select or build an appropriate relative correctness property to capture the intended relationship between the original and modified binaries.

We then run CBAT in “batch mode”, asking it to prove that the relative correctness property holds between each function in the modified binary and its counterpart in the original. Each comparison can have three outcomes: The property is proven to hold, a countermodel is found, or the tool runs out of time. Our goal is to get the number of functions in the latter two categories as small as possible—reducing manual verification and testing burden for end users in a recertification process. So, we manually examine these functions, identifying improvements to CBAT's WP computation for increased performance and accuracy. We focus on improvements that will apply across a broad range of programs and transformations, and we have observed this to be effective in practice—the improvements we have built, motivated by specific examples, improve results on future examples.

In this section, we describe the first of two representative applications of this process. In this first case, we are working with a binary provided by the GrammaTech team. The program in question is a version of GNU tar from the CMU/SEI testbed. GrammaTech applied their lifting and recompilation framework to the program, without otherwise modifying its behavior. The process of disassembling and lifting a binary to an intermediate representation and then recompiling it introduces pervasive changes throughout the program, including modifications to the locations of sections and globals, alternate choices for assembly-level encodings of higher-level logical structures, and reordered basic blocks in functions.

The GrammaTech transformation is not intended to change the semantics of the binary, so the relative correctness property we chose for this experiment is register value equivalence: we check that if a function in the original and modified binary are called with the same arguments, all machine registers hold the same values when they end.

4.2.1 Example: Baseline Results

The baseline results are shown in Figure 7. The tar binary has 1171 functions, and at the outset the CBAT tools could verify equivalence of 484 (41%) with a 15-minute timeout. Of the remaining functions, 259 (22%) timed out, and counterexamples were found for 428 (37%). Our next step

was to dig into examples from both of the latter two categories, identifying what was causing slowdowns in the tools, and how computed precondition might be adjusted to eliminate some false positives with a higher fidelity model of program behavior.

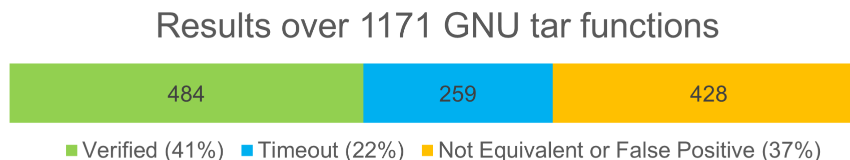


Figure 7: Baseline results on GrammaTech-transformed tar example.

4.2.2 Improvement 1: Performance and handling memory offsets

Our initial evaluation identified several potential performance improvements to reduce the number of timeouts. As an example: the `eval` function described in Section 3.1 maintains a collection of variable substitutions that is updated incrementally. This collection is essentially a map, and we use an association list to represent it, which is potentially inefficient compared to other map implementations, like tree or hashmap. Unfortunately, it is not possible to move directly to a more efficient map implementation because the SMT solver API we use requires this information to be provided as a list, and the cost of converting back and forth would outweigh the performance benefits of using an efficient map implementation internally. However, we still found that the performance could be improved by more aggressively pruning duplicate entries and sorting the list so that the most recently modified variable appears at the front (as recently modified variables are likely to be used again).

Additionally, we observed that the recompilation process has the effect of moving all of the binary's sections to new addresses. Our weakest precondition model initially assumed that when comparing two versions of a function, they were called with the same initial state of memory. This is particularly problematic in the case of data and bss sections, which contain program data like global variables accessed by many functions, and which have been shifted to new locations in this particular transformation. So, we improved CBAT to examine the two binaries, determine the relative offset of key sections, and apply this offset to the model of memory. We added this feature as an optional flag, called `--mem-offset`.

Figure 8 shows the results after the performance improvements and addition of the `--mem-offset` flag. We see that the number of functions verified has increased from 484 (41%) to 616 (53%), while the number of timeouts has decreased from 259 (22%) to 174 (15%) and the number of likely false positives has decreased from 428 (37%) to 381 (33%).

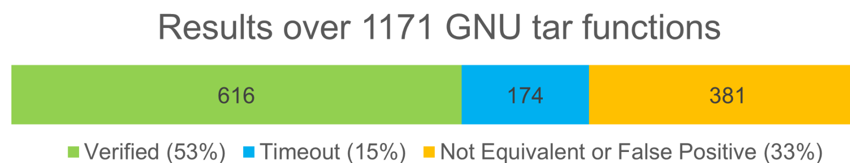


Figure 8: Results after performance improvements and addition of `--mem-offset` flag.

4.2.3 Improvement 2: Handling interprocedural pointers

The property we are checking in this example is that the behavior of the original and modified functions, as visible through the final value of registers, is identical when passed identical inputs. In practice, most functions cannot be called with all possible input values in a realistic execution of the program. For example, in the case of tar, an integer argument might be used to represent a compression level option that has only a few possible settings. The function will only ever be called with a few specific options for this argument.

In a case like this, CBAT still checks that the two function versions behave identically on all possible integer values, by default. This is a conservative approximation: it is safe, but might lead to false positives if the original and modified functions behave differently on inputs that are never used in practice.

Our experience suggests that false positives resulting from this choice are rare except in one case: pointer arguments. There are regions of memory that CBAT does not assume to be identical between the original and modified program, like the region below the stack pointer, code memory, and memory not used at all by the program. However, programs that pass values as function arguments that are used as pointers will, in practice, only user pointers to certain specific regions of memory (e.g., the stack and heap). Since our goal in comparative analysis is not to find bugs in the original program, but rather to find differences introduced by the applied binary transformation, we can rule out pointer values to uninitialized or otherwise bad memory on the basis of the correctness of the original program.

The results of CBAT after adding assumptions that passed pointers are valid if they are used in the original program are shown in Figure 9. We see that the number of likely false positives decreases from 381 (33%) to 236 (20%). Some of these functions are now verifiable, with the number of verified functions increasing from 616 (53%) to 682 (58%), and some now time out, with the number of timeouts going from 174 (15%) to 253 (22%).

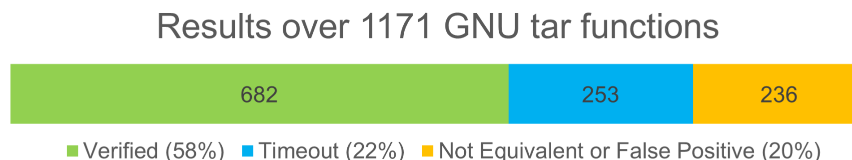


Figure 9: Results after improvements to handle pointers in function arguments.

4.2.4 Improvement 3: Handling hard-coded addresses for globals

We previously described the issues caused by shifted memory regions in the modified binary, and the `--mem-offset` flag we implemented to handle them. This flag applies to memory accesses—when the program accesses certain regions of memory, it enables CBAT to make sure the comparative analysis occurs at an offset.

While this adjustment allows CBAT to see that the *values* in the shifted regions are the same, the *addresses* of these values are still different. This can cause various issues not only in analyzing whether the modified binary has unintended changes, but also in defining what an "intended" or an "unintended" change is. Even in simple cases like this example, where we are checking programs for equivalence, defining this property is complicated by address changes. As a simple example,

consider a C program that simply defines a single global variable and prints its address. If that address has changed in the modified binary, should we say the programs are equivalent, because they both print the address of the same global variable, or not equivalent, because they print two different addresses? Or, in the case of the program and property we are currently considering, how should we handle a situation where the address of a global is stored in a register when a function ends?

To handle this issue, we added the `--rewrite-addresses` flag. Rather than just adjusting the model of memory accesses to account for offset regions, we adjust the modified binary itself so that its globals are at the same locations as the original, and update the references to these globals in the code. This essentially clarifies that for the purposes of comparing program behavior, global variable addresses matter only as a reference to the variable's value. Their numerical values are not interesting in themselves. We believe this approach is safe, in the sense that it will not result in new false negatives where WP misses bugs, as long as we do not change the location of any variables that have external linkage.

The results when adding this flag are shown in Figure 10. We see that the number of potential false positives has decreased from 236 (20%) to 122 (10%). The number of verified functions increases from 682 (58%) to 765 (65%), and the number of timeouts increases from 253 (22%) to 284 (24%).

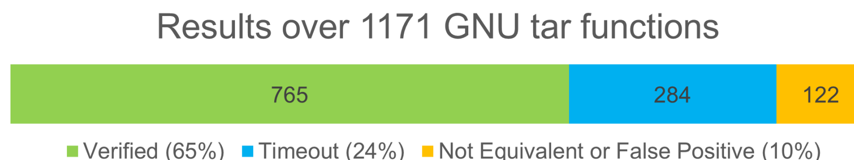


Figure 10: Results after improvements to handle hard-coded addresses of global variables

4.2.5 Improvement 4: Non-argument pointer validity

Improvement 2 described the observation that pointers passed as function arguments must point into valid memory regions if they are used successfully in the original program. Subsequent observation of the remaining potential false positives determined that other pointer, such as those read from memory, sometimes resulted in the same issues as argument pointers. Thus, we extended the logic previously applied in that setting to any unknown pointer successfully used in the original program.

The results after the addition of this feature are shown in Figure 11. We see that the number of potential false positives has decreased from 122 (10%) to 42 (4%). The number of verified functions increases from 765 (65%) to 827 (71%), and the number of timeouts increases from 284 (24%) to 302 (26%).

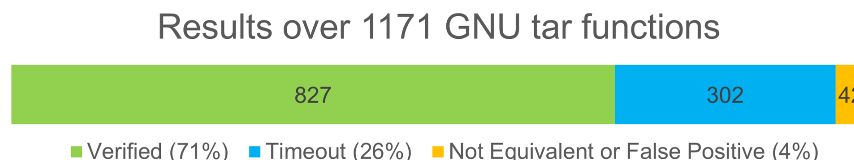


Figure 11: Results after improvements in non-argument pointer validity model

4.2.6 Example summary

The improvements described above were developed over the course of six months, as part of the CBAT project's research. Figure 12 summarizes the improvement in results over that time. Our experience on other examples shows that the improvements described in this section are not specific to the particular example in question—they provide benefits across a wide range of binaries and transformations.

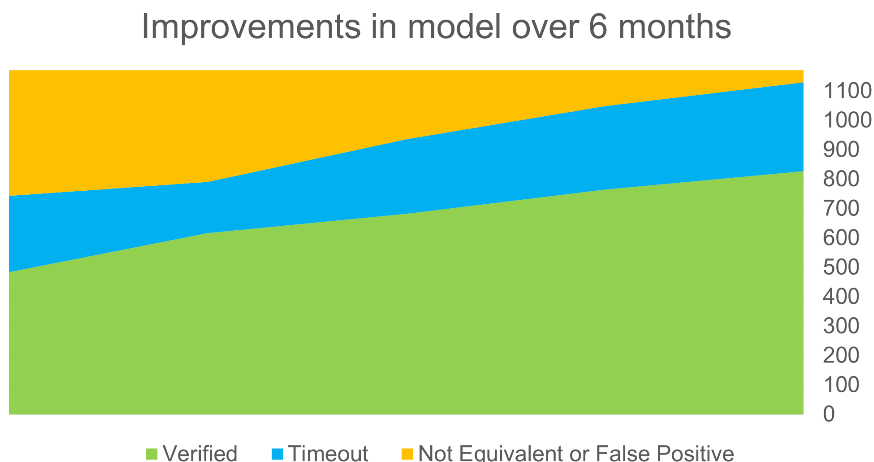


Figure 12: Improvement overtime on the lifted and recompiled GNU tar example.

We see that the number of verified functions increased from 484 (41%) to 827 (71%). Of particular note is the decrease in the number of functions for which WP finds a counter-model: from 428 (37%) to 42 (4%). Combined, these results mean that CBAT can substantially reduce verification burden, and has a false positive rate where it is becoming realistic to manually inspect all countermodels arising from a particular program. Of course, work remains to understand the root causes of the remaining countermodels, improve performance to reduce the number of timeouts, and measure effectiveness across a broader range of examples.

4.3 Applying and Improving CBAT: Example 2

After completing the application of CBAT to the Grammatech example described in the previous section, we moved on to a transformation developed by the Galois TPCP team, called "Embrittle". Like the GrammaTech transformation, Embrittle is intended to preserve the semantics of the program, so we can use the same relative correctness property in both experiments.

The Galois team also started from a similar binary—a slightly different version of GNU tar. The Embrittle tool introduces binary diversity with transformations like shuffling blocks and renaming program elements. At the time of our experiments, Embrittle worked on statically linked binaries. The GrammaTech experiement was performed on a dynamically linked version of tar, so the binaries differ slightly in that there are approximately 500 additional functions in the Galois binaries, largely from libc. These additional functions are included in our analysis.

This second application provided a good opportunity to evaluate whether the improvements we built for the GrammaTech example are specific to that case, or are general enough to apply to other transformations. Our results appear in Figure 13. As shown, our initial results on the Galois example were slightly better than our final results on the GrammaTech example, providing good

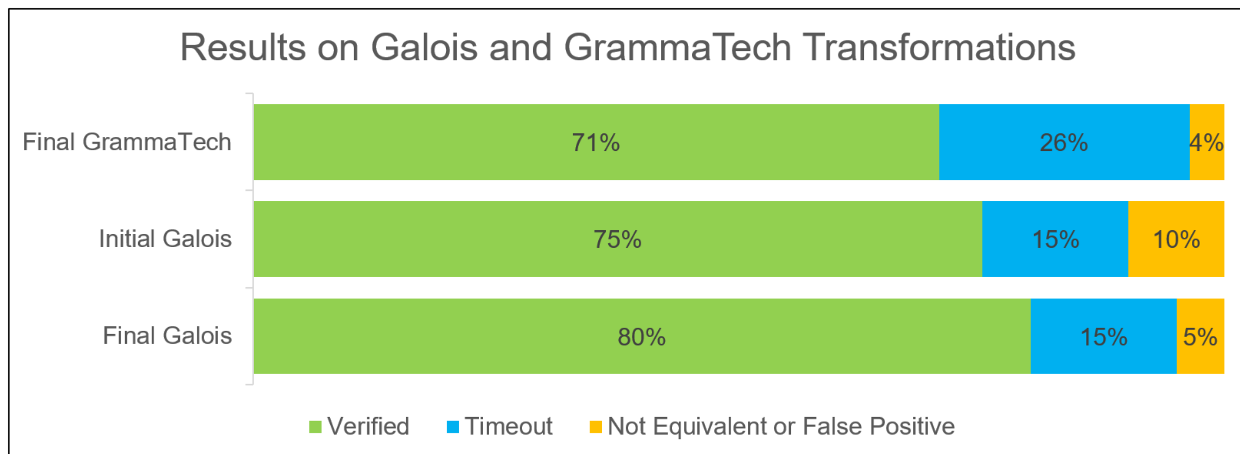


Figure 13: Results on application to Galois's Embrittle transformation, compared with previous results on GrammaTech example

evidence that the improvements we developed are applicable accross transformations.

As shown in Figure 13, we also used this as an opportunity to develop additional improvements. In particular, the Galois transformation renames functions in ways that initially made it difficult for CBAT to find the transformed version of a function in the modified binary in some cases. This motivated the addition of a new flag, `--func-name-map`, which maps names of functions in the original binary to names in the modified binary according to user-provided regular expressions for comparative analysis. With this improvement, we achieved the improved results shown in the figure.

5 Other Applications

At bottom, CBAT is a relational verification tool: it compares whether or not a particular relationship holds between two programs. This is of course useful for the TPCP case, because it can be used to verify that late-stage customizations did not break anything.

Yet because CBAT can verify such a wide-variety of relational properties, CBAT is useful to other domains beyond TPCP. Wherever there is a need to compare two programs to see if they stand in a particular relationship, CBAT can be used to verify the presence (or lack) of that very relationship.

In fact, CBAT has proved so useful that it is already being used by other projects at Draper. For instance, CBAT is a core part of Draper's contribution to DARPA's Assured Micro Patching program (more on this below), and it is being considered for other projects too.

In this final section of the report, we will illuminate CBAT's potential by describing three further research areas to which CBAT can be applied: automated program repair, translation validation for optimizing compilers, and automated specification synthesis.

5.1 Automated Program Repair

CBAT can be used as part of an automated program repair (APR) toolchain. Most APR tools operate as follows. The tool takes as input a program and a test suite. The tool then takes a failing test, and it constructs a patch that makes the test pass. It then repeats the process until all tests are passing.

There are two basic flavors of APR. One is heuristic-based (also called a generate-and-validate approach). The tool tries a series of different patches until it finds one that makes the tests pass. This is essentially a try-until-you-succeed process, which can be guided by various heuristics to make the guesswork less random (for an overview of heuristic-based techniques, see [40]).

A second and generally more effective flavor of APR takes a semantics- or constraint-based approach (for examples, see [41]; [42]; [43]; and [44]). It usually consists of the following process:

1. The tool uses symbolic execution to find different paths through the buggy program, noting branching conditions that make it go one way or the other.
2. The tool then eliminates any path where the test fails, and it picks one of the remaining paths where the test passes (the idea being, "to fix the failing test, we should make the program go down this path").
3. The tool then takes the branching conditions that caused the symbolic executor to travel down the chosen path and hands them off to a code synthesizer, which generates a patch that satisfies those constraints.

With the above techniques, APR tools face a serious problem: the patches that they generate tend to *overfit* the test data. This is a problem for both heuristic- and semantics-based tools (see [45]; [46]).

To illustrate, consider the following function, inspired by [40]. This function is meant to check if at least two out of three integers are equal:

```
bool are_two_equal(int x, int y, int z) {
  if (x == y || y == z) {
    return true;
  } else {
    return false;
  }
}
```

The programmer made a mistake in the `if` statement: they forgot to check for the third case, namely whether `x == z`.

Now consider the following test suite for the above function:

Test ID	x	y	z	Expected output	Pass/fail
01	-1	3	3	true	pass
02	3	3	0	true	pass
03	2	0	2	true	fail
04	-1	0	1	false	pass
05	-2	0	2	false	pass

To fix the failing test, a semantics-based APR tool might put together the following replacement for the `if` statement:

```
...
if ( (x + y + z) > 0 ) {
...
```

If we apply that patch to the above function, we get this code:

```
bool are_two_equal(int x, int y, int z) {
  if ( (x + y + z) > 0 ) {
    return true;
  } else {
    return false;
  }
}
```

With this new version of the function, all tests pass, since the tests that expect `true` have inputs whose sum is a positive number, and the tests that expect `false` have inputs whose sum is zero or less.

But of course, this solution badly overfits the tests. It certainly makes the *tests* pass, but it does not generalize to other cases, and it is certainly not a fix that a human would make.

This is a toy example, but it illustrates the chief problem. APR tools use test cases as implicit specifications, but test cases are too weak as specifications to guide correct program repair.

APR tools attempt to remedy the problem in various ways. Some focus on generating more tests (e.g., [47]; [48]; or [49]). Others focus on formulating better constraints, e.g., through variable analysis, or mining documentation and prior fixes ([50]). The approach taken in [51] bases patches

on human written specifications, and [52] synthesize fixes only for failing tests that contain certain kinds of recognizable errors that have pre-written fix specifications.

Since CBAT can verify very general relational properties, it could be used to verify synthesized patches for relational correctness. In this way, CBAT could be used to filter out high-quality from low-quality patches.

CBAT is already being used in a similar way in the realm of micro-patching (see DARPA's Assured Micro Patching (AMP) program[15] and Draper's VIBES contribution to the program[16]). The goal of micro-patching is to surgically make one or more tiny alterations to a binary in order to fix a bug. To accomplish this task, Draper's VIBES tool takes as input (i) a buggy program, (ii) a proposed fix, and (iii) a relational property describing how the fixed program should behave, relative to the original pre-patched program. Then, VIBES synthesizes a micro-patch at the binary level that satisfies the relational property requirements.

5.2 Translation Validation

CBAT can also be used to validate compiler optimizations. After a compiler translates human-readable code into an intermediate representation (IR), it can then make a series of optimization passes over the IR, doing things like removing dead code, replacing unnecessary calculations, and so on. But one might ask: how can we be sure that these optimizations do not introduce any bugs?

Intuitively, it seems promising to think that an optimization would turn out to be "bug-free" if the optimized code merely *refines* the original: i.e., if the new code's possible behaviors are a subset of the original code's behaviors. So, can we check a compiler's optimizations for refinement?

One way to do this is to formally verify the compiler's internal optimization logic. That way, you verify once and for all that the compiler will always produce optimized code that refines the original. CompCert ([53]) is a state-of-the-art example of just such a verified compiler.

Another approach is called "translation validation" (or TV, for short). Translation validators (or TVers) are separate tools that stand outside the compiler and validate the transformations externally.

A typical TVer takes as input two IR fragments—the original and the optimized fragment—and it checks whether the optimized fragment refines the original If it discovers that the optimized fragment is not a refinement, then it raises the alarm, since it has discovered an optimization that has in fact introduced *new* behavior that wasn't there before.

The concept of translation validation was first introduced by [54], and many new techniques have been developed since (for instance, see [55]; [56]; [57]; [58]; [59]; and [60]). While these techniques check for refinement automatically, their specification of refinement is usually worked out manually. For instance, the rules described in [60] constitute Alive2's refinement specification, and the authors likely had to devote extensive time to developing those rules. Moreover, since the refinement specifications are constructed manually, the checkers are too.

Since CBAT can check relational properties like refinement, CBAT could instead be employed to verify any of these refinement relationships directly. In this way, CBAT can serve as an off-the-

[15] https://www.darpa.mil/program/assured-micropatching
[16] https://github.com/draperlaboratory/VIBES

shelf refinement checker for TVers.

There is another way in which CBAT's ability to check very general relational properties can be useful in this domain. We can illustrate with a well-known example. The following function deals with sensitive data (an authentication token) stored in a buffer. For security purposes, the buffer is scrubbed clean (it is zeroed out) at the end of the function:

```
int check_auth(int *payload) {

  int *token = read_first_field(payload);
  int status = OK;

  if (!valid(token)) {
    status = FAIL;
  }

  memset(token, 0, sizeof(token)); // scrub the buffer
  return status;

}
```

An optimization pass might look at the `memset` and see an orphaned write: data is written, but never looked at again. Hence, it might consider that `memset` to be dead code and eliminate it altogether, yielding the following in its place:

```
int check_auth(int *payload) {

  int *token = read_first_field(payload);
  int status = OK;

  if (!valid(token)) {
    status = FAIL;
  }

  return status;

}
```

This is clearly not what the programmer intended. Nevertheless, it can easily pass a refinement check and get the TVer's stamp of approval.

Since translation tools check merely for refinement, they allow this unwanted optimization to occur. In order to prevent this, we need a stronger relational property that compares the original and optimized programs in terms of the data they leave around in memory (i.e., an optimization should not make a program leave information lying around in memory that wasn't left there before).

This example suggests that it might sometimes be advantageous to avoid hard-coded refinement checks, and instead work with more tailored specifications that attend to the specifics of different cases. In this way, one could do optimizing compilation, or "secure compilation" ([61]) as needed. Since CBAT can verify both refinement and security properties, it is again an ideal off-the-shelf validator for these more general TV pipelines.

5.3 Specification Synthesis

The final research area to which CBAT can be applied is the autamoted synthesis of specifications. This problem area is much more abstract than the previous two discussed above, but it has great potential.

At bottom, most verification tools check whether programs satisfy specifications. Some tools check just one program at a time. Other tools compare two or more programs at a time, in order to check that a certain relationship holds between them. To perform this task, comparative verifiers rely on techniques like differential checking ([62]) or relational verification more broadly ([63]), particularly relying on the construction of "product" programs ([64]; [65]). As has been described in this report, CBAT is just such a tool, with the power not only to verify one program at a time, but also to compare programs relationally.

Still, CBAT and other such verification tools must be given a specification to check, and for many advanced cases, that specification must be written by hand. Sometimes writing a specification is straightforward, but as is well known to practitioners, more often than not it is a tedious and labor-intensive task. It often needs to be done on a case-by-case basis, and it requires a good deal of expertise in formal methods.

These difficulties naturally lead to the following question: is it possible to automate the process of producing a specification? If so, can the entire process be automated, or only parts of it?

The utility of a positive answer to these questions is obvious. Given the difficulty of writing specifications by hand, even partially automating the process would have significant payoff.

Moreover, many users shy away from verification tools because of how difficult they are to use, and that difficulty is anchored at least partly in writing specifications. Specifications need to become more user-friendly, and automation would certainly go a long way towards achieving that goal.

However, it is unclear to what extent automating specifications is possible. To mechanically identify what a program *does* is one thing, but to mechanically identify what it *should* do is another. A specification stipulates what a human intended, and it is not obvious how a machine could mechanically deduce what a human meant, at least not without some level of human input.

In addition, the notion of "correctness" is often ill-defined. For instance, many verification tools check for refinement, but as noted above when discussing translation validation, it is possible to refine a program and yet violate security properties ([66]; [67]). So when is refinement the desired outcome, and when is security? It is not obvious how a machine could mechanically deduce the answer to questions like this one, at least not without some level of human input.

5.3.1 The Literature

Despite the difficulty of the topic, researchers have made some progress over the last few decades.

Specifications need to be formally precise, and this has both advantages and disadvantages. On the one hand, the formality makes specifications amenable to mathematical analysis, and that is where much of their power lies. On the other hand, the formality makes specifications difficult to write and understand (see [68]; [69]).

To help make it easier to work with formal specifications, some researchers developed hierarchical specification frameworks, as well as graphical representations (e.g., UML diagrams). For exam-

ples, see [70]; [71]; [72]; [73]; or [74]. Although these frameworks are now quite dated, their goal remains important. Specifications involve different levels of abstraction that are difficult for humans to manage. Making those levels perspicuous aids the specification writer.

Another problem is this: how can one combine specifications in a sound way? [75] established mathematical foundations for composing specifications, and this project was carried forward in a variety of ways (e.g., [76]; [77]; [78]; [79]; [80]). Much of that work led to the development of a family of flexible algebraic specification languages, including Clear ([81]), OBJ ([82]), CASL ([83]), and Glider ([84]).

The success of these languages made it possible to create tools like CAT ([85]), VDN ([86]), DTRE ([87]), KIDS ([88]), SpecWare ([89]), or Spec# ([90]). These tools synthesize software through a process of specification refinement: users begin by designing high-level specifications of the system they want, and then through a series of refinements, an implementation is gradually synthesized. A similar tool is REFINITY ([91]), which utilizes abstract execution (symbolic execution over abstract programs) to synthesize refactored Java code from user specifications. Rhodium ([92]; [93]) is also similar in spirit: it can automatically verify program transformations written in its custom specification language.

Still, the tools listed above require human-written specifications in some form or other. In the last two decades, a number of techniques have been developed in an attempt to synthesize specifications directly. For a sample of some of these techniques, see [94]; [95]; [96]; [97]; [98]; [99]; [100]; [101]; and [102]. Whether any of these techniques are mature enough to be utilized to automate the process of writing specifications is an open question.

One interesting area of research concerns those parts of the C language that are underspecified. In order to verify underspecified C code, [103] construct an ActiveObject model for the underspecified code, and they use that to deduce (and then verify) the possible behaviors that different implementations of the underspecified code can have. Although the goal here is not to directly infer a specification, this approach does automatically construct a picture of program behavior without a specification, and that same idea could perhaps be used to infer specifications more generally.

Another relevant research area is focused on the task of inferring specifications for unknown library procedures (for instance, see [104]; [105]). This task is useful in a variety of contexts, but what matters for us are the techniques used to infer the specifications. Perhaps these techniques could be used to generate a candidate specification for any block of code that a user might write, which the user could then refine.

In a similar vein, researchers have developed forms of abductive inference that can be used to infer hypotheses for weakest precondition analysis. For example, see the papers by [106]; [107]; [108]; [109]; [110]; and [111]. These techniques could perhaps also be used to generate candidate specifications for arbitrary blocks of code.

Another area of interest concerns a certain class of properties known as "hyperproperties." Traditionally, safety and liveness properties are understood as properties of execution traces ([112]; [113]). However, many security properties cannot be properly formulated as properties of execution traces. Instead, they need to be formulated as properties of entire systems, and these system-level properties are called *hyperproperties* ([114]).

Various verification techniques have been developed to verify hyperproperties. For instance, one can compose a program with itself k times, and then perform a relational analysis on the composite ([115]). So hyperproperties can be used to both express and verify system-level security properties. This suggests that much could be gained by allowing specifications to speak about the hyperproperties of a program, rather than just its execution properties. For more on hyperproperties and verification, see for instance [116]; [117]; [118]; and [119].

One other possibility to note arises if we apply a topological technique from the pattern-matching domain to the task of combining specification fragments to form a larger specification. [120] showed that pattern matching has a natural interpretation as a sheaf, and that one can construct a larger match by systematically stitching together smaller matches (see a similar sheaf-theoretic approach to network analysis and algorithm construction in [121]). Perhaps the same idea could be applied to construct specifications? The execution traces of a program form a natural topology, which we might think of as the "space of executions" of the program ([122]; [123]; [124]). If one were to assign specification fragments to each piece of that space, one could in principle construct a larger specification by systematically stitching together smaller fragments.

Next Steps

The research community is still a long way from being able to automatically generate useful specifications for arbitrary programs. But there are many ways that useful progress can be made.

One can ask whether the entire process of generating a spec can be automated in its entirety. At the most general level, this problem can seem intractable at present. A more promising approach is to focus on developing interactive tools that can help guide the writer in constructing specifications, perhaps similarly to the way that interactive theorem provers interact with their users today.

Imagine a tool integrated into, say, Ghidra and other binary analysis tools, which engages the spec writer in a kind of back-and-forth exchange. As the user examines various pieces of the binary program, the tool could guide them through the process of building up a specification piece by piece.

In fact, CBAT is based on an idea that could be used to help generate candidate relational specifications: assume that the original program is correct, and then generate a diff of the two programs' behavior. That diff can then be presented to the user as a candidate specification. The user could then accept, deny, or modify the specification, which CBAT could then verify.

In order to accomplish tasks such as these, CBAT is well suited to incorporate and extend any of the above-mentioned techniques from the literature, including but not limited to:

- Many of the above techniques construct a specification for, say, a library function by formulating the weakest preconditions that are needed for that library function to do its job. Since CBAT already formulates weakest preconditions, CBAT could be extended so that instead of solving weakest precondition, it could alternatively propose weakest preconditions.
- Abduction is also similar to a weakest precondition analysis, and so CBAT could in principle be extended to use abduction or other similar techniques to generate candidate specifications for any selected piece of a program or programs. The user could then accept, reject, or modify such candidates.

- CBAT already uses a more general form of self-composition under the hood to compare programs. CBAT can thus already verify various 2-properties, but it could be extended to handle k-properties for any integer k.
- CBAT could be used to construct the weakest precondition for a small piece of a program, and that could be presented to the user as a candadite specification. In a back-and-forth interactive process, CBAT and the user could then widen their scope to stitch together a specification for larger and larger covers.

5.4 Conclusion

The above demonstrates CBAT's wide applicability to other active areas of research in the field. Because it is such a general tool, CBAT is an excellent off-the-shelf verifier that can be used whenever one needs to verify relational properties of programs.

This report has described the research and software development carried out by the CBAT team in the context of ONR's Total Platform Cyber Protection (TPCP) program. For TPCP, CBAT enables automated verification of late-stage software customizations and is sufficiently mature to handle realistic software.

CBAT is now a useful and complete tool, and has been released publicly as open-source software [14]. It has extensive documentation and an in-depth tutorial that walks users through much of its functionality. It also offers a guide to the underlying framework, BAP, that CBAT is built on top of, in order to better serve the power user who wants to contribute to CBAT, and to further adoption. CBAT is already in use by other projects at Draper, and can be directly utilized as useful off-the-shelf verifier in other areas such as automated program repair or translation verification.

Overall, the CBAT project has advanced verification research and demonstrated that formal verification can enable late-stage software customization with evidence for recertification. As the issues of insecure legacy and COTS products in critical systems continue to grow, CBAT and the broader TPCP program offer a realistic and exciting path forward.

Appendices

A A Gentle Introduction to CBAT

In this section, we introduce readers to the use of CBAT. This material is intended as a stand-alone tutorial that can be read independently of the rest of the report, to provide self-contained documentation and usage guidance. It is adapted from a tutorial presented at the 2020 Software Security Summer School. It is updated for the most recent version of CBAT as of the delivery of our final report, which is included with our base period deliverables. The CBAT website contains a version updated for the latest release.[17] All instructions in this introduction are designed to be run from the following directory in our repository:

```
/path/to/cbat_tools/docs/tutorial
```

A.1 Overview

Let us begin by providing a brief overview of what CBAT is, and what it does.

A.1.1 What does CBAT analyze?

CBAT is a tool that helps you analyze the behavior of binary executables (i.e., compiled programs). It does not analyze the source code you have written in a language C or Rust. Rather, CBAT analyzes the assembly-level machine instructions that the computer executes directly when it runs your compiled program.

This is useful because the executable is the real program. High level programming languages offer many features, but these features are compiled down to the same basic machine instructions—these are what CBAT analyzes.

A.1.2 What does CBAT do?

At its heart, CBAT checks whether a function in your program (or programs) behaves in a certain way. In other words, it can verify that a function in your program has certain properties.

For example, suppose there is a function `foo` in your program, and you want to know if it always produces the number `10` as its output. You can ask CBAT to check this for you. CBAT will explore all logical possibilities, and find if there is any possible way that `foo` could produce anything other than `10`. If it finds a way, it will give you an example of inputs you can feed into `foo` to make it do what you didn't expect.

It is worth emphasizing that CBAT does not perform a probabilistic or fuzzy sort of check here. CBAT performs a mathematical check of all logical possibilities. CBAT does make some simplifying assumptions for the sake of performance. But, up to our assumptions, if there is any way to make `foo` produce anything other than `10`, CBAT will find it. By contrast, if CBAT cannot find a way, then CBAT has essentially found a logical proof that `foo` *always* produces `10`.

A.1.3 Comparing programs

CBAT can analyze functions in a single binary program, but it can also compare functions in two different programs and check whether their behavior is the same. This is useful for checking whether a newer, modified version of a program still works the same way as an older version.

[17] https://draperlaboratory.github.io/cbat_tools/

For example, suppose you make some improvements to `foo`, and you want to know whether your improvements have broken anything. CBAT can analyze both your original and your improved program, and make sure that the modified version of `foo` always produces the same output as the original version of `foo`.

A.2 Binary Programs

CBAT is a family of command line tools built on top of Carnegie Mellon's Binary Analysis Platform (BAP). CBAT and BAP do not analyze high-level code in a source language like C or Rust. Rather, they analyze the compiled, "binary" version of such programs.

When you compile (say) a C program or a Rust program, the compiler converts your source code into machine code, which the computer can execute directly.

A computer has a collection of *registers* that you can stash values in, often with well-known names like `RAX`, `RDI` and `R9`:

```
+----------+---------------+
| Register | Slot          |
+----------+---------------+
| RAX      | 0x000000      | <--- values go in the slots
+----------+---------------+
| RDI      | 0x000023      |
+----------+---------------+
| RBP      | 0x000000      |
+----------+---------------+
| R9       | 0x000000      |
+----------+---------------+
| R10      | 0x000000      |
+----------+---------------+
| ...      | ...           |
+----------+---------------+
```

There are also a number of smaller registers, often call *flags*, which can only contain a one or zero, to indicate that they are "on" or "off." They often have names like `ZF` and `OF`:

```
+------+--------+
| Flag | On/off |
+------+--------+
| ZF   | 0x0    | <--- Flag is off
+------+--------+
| OF   | 0x1    | <--- Flag is on
+------+--------+
| AF   | 0x0    |
+------+--------+
| ...  | ...    |
+------+--------+
```

Finally, there are regions of *memory* you can stash values in too. Memory is essentially just another array of slots you can put bytes in, and each slot has an address.

```
+----------+-------+
| Address  | Slot  |
+----------+-------+
| 0x000000 | 0x00  | <--- bytes go in the slots
+----------+-------+
| 0x000001 | 0x23  |
+----------+-------+
| 0x000002 | 0xaf  |
+----------+-------+
| 0x000003 | 0xa2  |
+----------+-------+
| ...      | ...   |
+----------+-------+
```

The machine doesn't let you do very much with all of these slots. You can put values in them, copy values out of them, compare values in them, and do basic arithmetic on them.

The addresses and values used by the machine are always binary numbers, i.e., sequences of ones and zeros. These are called *bitvectors* (or sometimes just "binary numbers").

There is also a GOTO instruction, which tells the machine to jump to a different instruction in your program. At a high level, machine code is simple GOTO-style programming, with a bunch of slots to put your (binary) values in.

One of the problems with binary analysis is that every computer architecture is different. The slots are laid out differently and the machine instructions take different forms. For this reason, programs compiled for, say, `x86_64` look different than those compiled for `ARM`.

BAP is a big help here. It takes most of these variations in machine instructions, and it lifts them into a unified and simpler form of machine code, which it calls BIR (short for the "BAP Intermediate Representation"). We'll just call it the "IR" for short.

A.2.1 Function arguments and return values

As was noted already, when code written in, say, C is compiled down to machine code, everything gets converted into simple machine instructions. Of course, functions in C code take arguments, and they return values. At the level of machine code, function arguments are return values are handled by putting them into specific, pre-defined slots.

In `x86_64` programs (which is what we'll be looking at), the first argument to a function is always placed in a register called `RDI`, and the function's return value is always placed in a register called `RAX`.

For example, suppose we have a function in C that returns the value `7`:

```
int foo(int x) {
  ...
    return 7;
}
```

In BAP's IR, the return value would be handled like this:

```
00000436: subroutine foo... <-- Start of the function foo
00000443: ...
00000450: RAX := 0x07       <-- Put the result in RAX to return
00000458: return...         <-- Return to the caller
```

The numbers at the start of each line are unique identifiers, one for each line. The instructions come after those numbers, to the right. Instruction identifiers in this tutorial may differ from the ones you see when running CBAT interactively. That's okay; different versions or runs of CBAT can produce different numberings.

You can see here that the value `7` is placed in the register `RAX`, and then the function returns. This is how the function returns `7`. Return values are always placed in `RAX`.

Similarly, the argument to a function always goes in a register called `RDI`. For example, a call the above `foo` function with `3` as the parameter looks like this in BAP's IR:

```
00000467: RDI := 0x03        <-- Put the argument in RDI
00000476: call foo           <-- Call the function
```

You can see here that the value `3` is placed in the register `RDI`, and then the function `foo` is called. A function's first argument always goes in `RDI`. If your function takes more than one argument, the second argument always goes in a register called `RSI`, and there are other established slots for additional arguments.

For our examples, it is sufficient to know that a function's first argument goes in `RDI`, and the return value goes in `RAX`. To make the examples in this tutorial easy to follow, we have provided C code for each program. However, remember that CBAT actually operates on the machine code–no source code is necessary to use CBAT. On occasion, we will pay attention to the machine code, particularly the argument register (`RDI`) and the return value register (`RAX`).

A.3 CBAT Usage

BAP itself is a framework for analyzing binary programs. It has a command line interface, which you can extend with your own plugins. BAP takes a binary program and lets you run a pass over the program, during which time your plugin can dig in and analyze the internals of that program.

Our CBAT tools are implemented primarily as BAP plugins. The particular plugin that we will focus on here is called `wp`, which is short for "weakest precondition". This is the technical name in the literature for the kind of analysis that `wp` performs.

The `wp` plugin has two modes of operation. In the first mode, it can make a pass over a single binary program and analyze a function in that program. In the second mode, it can make a pass over two programs and compare a function that appears in both. Here we cover the first mode–the second mode will be covered when we reach the first example that uses it.

A.3.1 Analyzing one program

The command to analyze a single program has this basic form:

```
$ bap wp \              <-- The command is wp
    --func=FUNC \       <-- The function to examine
```

```
        [options] \           <-- Any extra options
        /path/to/exe          <-- The path to a binary program
```

This will tell CBAT to go into the program at `/path/to/exe` and analyze the function `FUNC` that occurs in that program.

There are various `[options]` you can specify to tell CBAT to do different things as it analyzes your program. We will illustrate the most important options throughout this tutorial.

A.4 Tripping Asserts

The `wp` tool is quite general. You can tell it to verify that your functions have certain properties, and you can specify your own custom properties that `wp` should check for. The tool provides some predefined properties. One of them is this: `wp` can check if it is possible to trigger an assert in a function in your program.

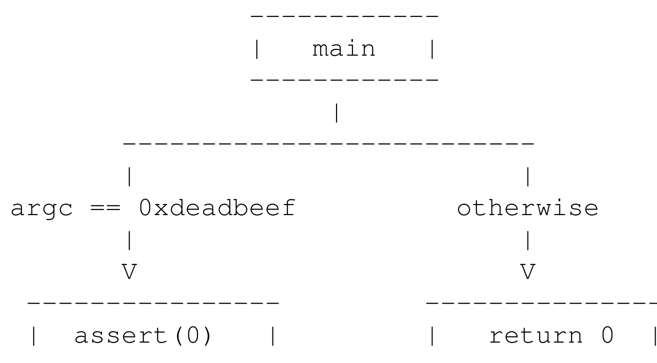
Consider the following simple C program (which lives at `tutorial/01/binary/main.c`):

```c
#include <assert.h>

int main(int argc, char** argv) {

    if (argc == 0xdeadbeef) {
        assert(0);
    }

    return 0;

}
```

Here we have a `main` function, which takes an argument, `argc` (we can ignore `argv`).

- If the value of `argc` is `0xdeadbeef`, then the program trips an assert, which causes it to halt.
- Otherwise, it returns `0`, which means success, and it exits cleanly.

Visually, here's the control flow:



As you can see from this diagram, if `argc` is `0xdeadbeef`, then the program travels down the left branch, where an assert is tripped, and the program fails. Otherwise, it goes down the right branch, where the program returns `0`, and exits cleanly.

Now, this is a C program, and as we noted earlier, `wp` does not analyze the C code itself. Rather, `wp` analyzes the compiled version of this program. We started with the C code to make it easier to read, but the real program is what you get when you compile it.

Let's see what the machine instructions look like just for the branching part of the `if` statement in our example program. We can ask BAP to show us the machine instructions as it sees them, like this:

```
$ bap 01/binary/main -d --print-symbol=main
```

You don't need to do that right now. But if you did, you could look through the output, until you find the relevant instructions, which should look something like this:

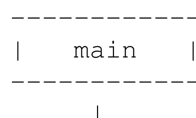
```
00000481: when #5 goto %0000047b  --------- When #5 is 1, go here ---+
00000879: goto %000006f9          ---- Otherwise, go here --+        |
                                                             |        |
000006f9: ...  <--------------------------------------------+        |
00000722: call @__assert_fail ...  <--- trip the assert              |
                                                                      |
0000047b: ...  <-----------------------------------------------------+
000004aa: call #52 with noreturn  <--- exit cleanly
```

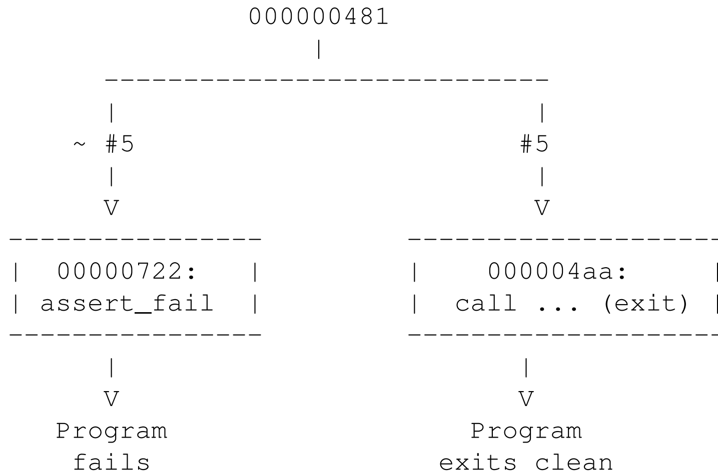
At the first instruction printed here (numbered `00000481`), you can see that the machine is going to jump to instruction `0000047b` when the value of a virtual variable called `#5` is true (i.e., `1`). You can think of a virtual variable as a temporary variable at an unspecified location.

So if we suppose that `#5` is `1`, we can see what happens next, by following the machine down to the instruction at `000047b`. There, we can see that the program exits cleanly.

Alternatively, if `#5` is not `1`, then the machine doesn't jump at `00000481`, and instead moves down to the next instruction, at `0000879`. What happens there? The machine jumps directly to instruction `0000006f9`, and if we look there, we can see that it trips the assert we are looking for.

Even though these are machine instructions and not C code, there is still a clear control flow here, similar to what we saw for the C code:





Of course, just because there is a branch in the code where an assert is tripped, that doesn't mean the program can actually travel down that branch. Programs can have "dead" branches, which are impossible to travel down:

```
if true:
    do_something()
else:
    never_called() # Dead code, never called
```

What we want to know in our example is whether there is any way that the `main` function can actually get to the branch that trips the assert.

We can ask `wp` to find out if this is possible. To do that, invoke `wp` like this:

```
$ bap wp \
      --func=main \
      --trip-asserts \
      01/binary/main
```

Here we ask `wp` to try to trip an assert inside the `main` function of the program at `01/binary/main`. When we run this command, the relevant output comes at the end, which looks like this:

```
Property falsified. Counterexample found.

Model:
  ZF  |->  0x0
  SF  |->  0x0
  RSP |->  0x000000003f800081
  RSI |->  0x0000000000000000
  RDX |->  0x0000000000000000
  RDI |->  0x00000000deadbeef
  RCX |->  0x0000000000000000
  RBP |->  0x0000000000000000
  RAX |->  0x0000000000000000
```

```
R9  |->  0x0000000000000000
R8  |->  0x0000000000000000
PF  |->  0x0
OF  |->  0x0
CF  |->  0x0
AF  |->  0x0
mem_orig |-> [
  else |-> 0x00]
mem_mod = mem_orig
```

The first thing it says is `Property falsified. Counterexample found`, which means that CBAT was able to find a way to trip the assert.

Next, it says `Model`, after which it lists some information. There, you can see a bunch of registers on the left, and a bunch of values on the right (and a mention of memory at the bottom, which we can ignore for now).

What this tells us is how to trip the assert. The idea here is that if we set the registers (and memory) to these particular values at the start of the `main` function, then `main` will trip the assert. (There's more than one correct solution; your CBAT installation might find a different solution.)

In this particular example, we can see that most of the registers are set to zero. The one that is interesting here is `RDI`. Remember that `RDI` is where the argument to the function goes. And here, `wp` tells us that this register should be set to `0xdeadbeef`. That is to say, `wp` is telling us that if we set `RDI` to `0xdeadbeef`, then our function `main` will trip the assert. And that of course makes sense, given the control flow that we examined in this function.

A.4.1 BAP's IR

It can be instructive to step through the machine instructions and actually see the function exhibit this behavior.

One of the tools in the CBAT family is an interactive debugger that lets you do just that: step through a program, one machine instruction at a time. The tool is called `bildb` (short for "BAP Intermediate Language DeBugger"), and the command to invoke it looks like this:

```
$ bap 01/binary/main \
      --pass=run \
      --run-entry-point=main \
      --bildb-debug
```

This will start up `bildb` at the `main` function in `01/binary/main`. You should see something like this:

```
BIL Debugger
Starting up...

Architecture
Type: x86_64
Address size: 64
Registers:
```

```
R10    R11    R12    R13    R14    R15    R8     R9     RAX    RBP    RBX
RCX    RDI    RDX    RSI    RSP    YMM0   YMM1   YMM10  YMM11  YMM12  YMM13
YMM14  YMM15  YMM2   YMM3   YMM4   YMM5   YMM6   YMM7   YMM8   YMM9

Entering subroutine: [%00000868] main
0000088d: main_argc :: in u32 = RDI
0000088e: main_argv :: in out u64 = RSI
0000088f: main_result :: out u32 = RAX
Entering block %00000413
0000041a: #46 := RBP
>>> (h for help)
```

You can see that at startup `bildb` prints some information about the architecture and its initialization state, then it enters the function (subroutine) `main`, where it stops at the first instruction in the first block, namely:

```
0000041a: #46 := RBP
```

The instruction's identifying number is on the left of the colon, and the instruction itself is on the right of the colon.

Below that, `bildb` gives you a prompt, where you can type a command:

```
>>> (h for help)
```

There are various commands you can enter here, but they are fairly straightforward. For example, to see more than just one instruction, type `show 5` and hit enter (to see the nearest +/- 5 lines):

```
>>> (h for help) show 5
   %00000413
-> 0000041a: #46 := RBP
   0000041d: RSP := RSP - 8
   00000420: mem := mem with [RSP, el]:u64 <- #46
   00000427: RBP := RSP
   00000435: #47 := RSP
   00000438: RSP := RSP - 0x10
```

Another thing you can do is see which values are stored in registers. For instance, to see the value stored in `RBP`, type `p RBP` (short for "print `RPB`") and hit `enter`. You should see something like this:

```
>>> (h for help) p RBP
RBP    : 0
```

This tells you that the `RBP` register has the value zero stored in it.

Remember how `wp` told us that `main` will trip the assert if it begins with `RDI` set to `0xdeadbeef`? Let's set that value, and see if we can trip the assert.

To set `RDI` to `0xdeadbeef`, type `set RDI=0xdeadbeef` and hit `enter`. You'll see that `bildb` sets the value:

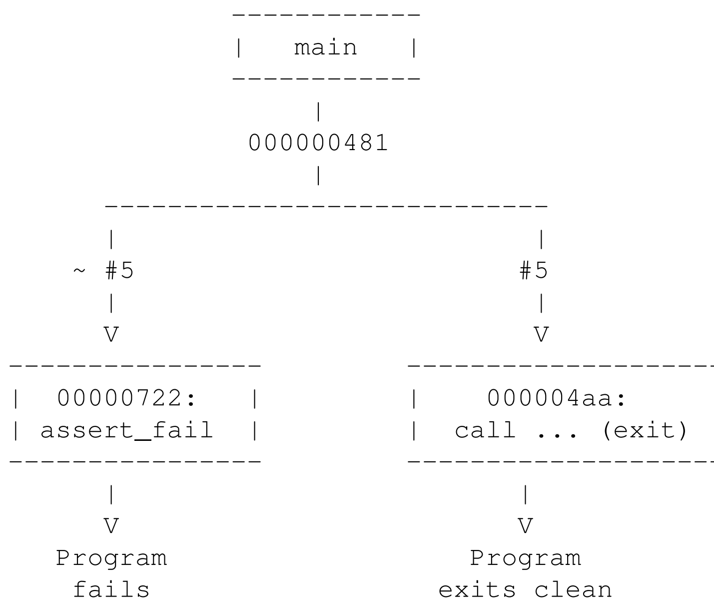
```
>>> (h for help) set RDI=0xdeadbeef
RDI    : 0xDEADBEEF
```

Now you can to step through this program, instruction by instruction, and you should end up triggering the assert. To move to the next instruction, hit `s` (for "step"), and hit `enter`:

```
>>> (h for help) s
0000041d: RSP := RSP - 8
```

You could keep stepping through this program, one instruction at a time, but you can also just set a breakpoint at the instruction you want and skip ahead to it.

Let's set a breakpoint at the assert, to see if we reach it. Recall the control flow graph we saw before:



You can see that the assert happens at instruction `00000722`, so let's set a breakpoint there. To set a breakpoint, type `b %00000722` (short for "breakpoint at instruction `00000722`"):

```
>>> (h for help) b %00000722
Breakpoint set at %00000722
```

Now let's see if we get to that breakpoint. To tell the debugger to move forward, type `n` (for "next") and hit `enter`. The debugger takes us to the next block:

```
Entering block %000006f9
000006fe: RCX := 0x4005ED
```

Type `n` (and `enter`) to skip forward again, and you will see that you hit the breakpoint at `00000722`:

```
>>> (h for help) n
00000722: call @__assert_fail with return %0000047b
```

You have now confirmed that the program does indeed trip the assert if `RDI` is set to `0xdeadbeef` at the beginning of `main`. You could keep stepping through the program, but there is no need at this point. To quit the debugger, hit `q` and then `enter`.

A.5 Hooking `wp` up to `bildb`

Instead of manually setting the values of registers in `bildb` to see the behavior that `wp` tells you about, you can have `wp` dump its output into a YAML file, and then have `bildb` read that in at startup. To have `wp` dump its output like this, run the following command:

```
$ bap wp \
      --func=main \
      --trip-asserts \
      --bildb-output=init.yml \
      01/binary/main
```

Notice how you simply added the option `-bildb-output=init.yml` to the command. That tells `wp` to dump the appropriate output into a file called `init.yml`.

Once you have that file at hand, you can start `bildb` with it, like this:

```
$ bap 01/binary/main \
      --pass=run \
      --run-entry-point=main \
      --bildb-debug \
      --bildb-init=init.yml
```

Notice how we added the option `-bildb-init=init.yml`. When `bildb` starts up, it will read `init.yml`, and set the registers to the values listed there. From there, you can set the breakpoint at `00000722` and run through the program exactly as before.

A.6 4-Rooks

As we saw in the last example, `wp` can find a way to trip an assert if it's possible to do so, and it will tell us how to do it. This can be used to solve puzzles of various kinds.

As an example, consider the 8-Queens chess puzzle. The task is this: on a standard 8x8 chess board, place 8 queens on the board in places where none of them can be captured by any of the others in a single move. Queens can move up/down, sideways, and diagonally, so you have to be careful to place the queens out of each other's paths.

To keep things simple, we'll do a smaller version: the 4-Rooks puzzle. It's just like the 8-Queens puzzle, except we have 4 rooks, and the board is a 4x4 board. Rooks can move up and down, and sideways.

Since this version of the game is less complex, it is somewhat easy to think up a solution. For example, here is one:

```
+---+---+---+---+
| R |   |   |   |
+---+---+---+---+
|   |   | R |   |
+---+---+---+---+
|   | R |   |   |
+---+---+---+---+
|   |   |   | R |
+---+---+---+---+
```

As you can see, each of these rooks is out of the path of the others, so none can be captured by any of the others in a single move.

Now suppose that you have a function (in C) that checks solutions to the 4-rooks puzzle. Imagine that the function looks something like this:

```
/* This function checks if solution is correct. */
int check(int solution) {

    bool correct = true;

  // Check if solution is correct...

  if (correct) {
    assert(0);
  }

  return 0;
}
```

This function takes a solution to the 4-rooks game as an argument, it then checks the solution, and if the solution is correct, it trips an assert.

How do we encode a proposed solution so that we can pass it into this function as an argument? Well, since machine code really only works with binary numbers, we'll need to encode proposed solutions as a bitvector.

How can we do that? There are 16 squares on a 4x4 chess board, so let's say that we'll represent the board with a 16-bit binary number, where each bit represents one of the squares. Let's also say that, for each bit, if it's `1`, that means there is a rook at that position, and if it's `0`, that means there is no rook at that position.

To encode a proposed solution as a 16 bit binary number, we mark each position on the board with a `0` or `1` to indicate whether there is a rook at that position. For example:

```
+---+---+---+---+            +---+---+---+---+
| R |   |   |   |            | 1 | 0 | 0 | 0 |     ==>    1000
+---+---+---+---+            +---+---+---+---+
|   |   | R |   |            | 0 | 0 | 1 | 0 |     ==>    0010
+---+---+---+---+     ==>    +---+---+---+---+
|   | R |   |   |            | 0 | 1 | 0 | 0 |     ==>    0100
```

```
+---+---+---+---+            +---+---+---+---+
|   |   |   | R |            | 0 | 0 | 0 | 1 |     ==>    0001
+---+---+---+---+            +---+---+---+---+
```

Then we line up those bits, one after another, into one 16-bit binary number:

```
solution = 1000 0010 0100 0001 (or 0x8241 in hex)
```

So, the `check` function takes a 16-bit number as the argument, it then checks the proposed solution to see if it is correct, and it trips an assert if it is correct.

The full code for this function can be found at `tutorial/02/binary/main.c`. The compiled version of this program is at `02/binary/main`. You can try it out by passing it a hex version of a possible solution. For example:

```
$ ./02/binary/main 0x2212
$ ./02/binary/main 0x8421
```

Since we have a function that trips an assert if the solution is correct, we can ask `wp` to try and find a way to trip that assert. If `wp` can find a way to trip the assert, it will give us an example of an input to the function that will cause this behavior.

Let's try it. Here is the command. We only want to analyze the `check` function, which we specify with `-func=check`:

```
$ bap wp \
      --func=check \
      --trip-asserts \
      02/binary/main
```

After a moment, `wp` returns a result. At the end of the output, you should see something like this:

```
Property falsified. Counterexample found.

Model:
  ZF  |->  0x0
  SF  |->  0x0
  RSP |->  0x0000000040000000
  RSI |->  0x0000000000000000
  RDX |->  0x0000000000000000
  RDI |->  0x0000000000004281  <-- Argument to function
  RCX |->  0x0000000000000000
  RBP |->  0x0000000000000000
  RAX |->  0x0000000000000000
  R9  |->  0x0000000000000000
  R8  |->  0x0000000000000000
  PF  |->  0x0
  OF  |->  0x0
```

```
CF   |->  0x0
AF   |->  0x0
mem_orig |-> [
  else |-> 0x00]
mem_mod = mem_orig
```

Here, we can see that `wp` did indeed find a way to trip the assert, and it provided an example of how to do it.

Look at the registers listed under `Model`. Remember that the first argument to a function is always stored in `RDI`, and here `wp` tells us that if we set that to `0x4281`, then our function `check` will trip the assert.

What is `0x4281`? In binary, it is:

```
0100 0010 1000 0001
```

If we break that up into a 4x4 matrix, we can see the chess board:

```
                +---+---+---+---+              +---+---+---+---+
0100    ==>     | 0 | 1 | 0 | 0 |              |   | R |   |   |
                +---+---+---+---+              +---+---+---+---+
0010    ==>     | 0 | 0 | 1 | 0 |              |   |   | R |   |
                +---+---+---+---+      ==>     +---+---+---+---+
1000    ==>     | 1 | 0 | 0 | 0 |              | R |   |   |   |
                +---+---+---+---+              +---+---+---+---+
0001    ==>     | 0 | 0 | 0 | 1 |              |   |   |   | R |
                +---+---+---+---+              +---+---+---+---+
```

We can see that this is indeed a solution to the 4-rooks problem. Since none of the rooks are in each other's way, none of them can be captured by any of the others in a single move.

This technique applies to many different kinds of puzzles or problems whose solutions can be encoded as a binary number. For example, solving a sudoku puzzle, reversing a hash, and so on.

A.7 Find a Null Dereference

Another predefined property that `wp` can check is to find null dereferences in a function. To illustrate, consider the following C program (`tutorial/03/binary/main.c`):

```c
#include <stdlib.h>

int main() {

    // Allocate a byte of memory, at address `addr`
    char *addr = malloc(sizeof(char));

    // Store the character 'z' at that address
    *addr = 'z';

}
```

In this program, we first allocate a byte of memory, and then we take the address of the byte that gets allocated and we store it in the pointer called `addr`. Next, we attempt to store the character `z` in the slot at that address.

Suppose I run this program. If the byte of memory I ask for here is successfully allocated, then everything works as expected. I get back an address for the allocated byte, and I can then store the character `z` at that address.

But things needn't turn out that way. It's always possible that `malloc` won't be able to allocate the memory I've asked for, in which case it can't return an address to me. If that happens, it will return `NULL` (i.e., `0`). And then, when I try to store `z` at that address, the program will segfault.

We can ask `wp` to find null dereferences like this one. To check this particular `main` function, we can use the following command (note that we add `--check-null-derefs` as a flag):

```
$ bap wp \
      --func=main \
      --check-null-derefs \
      03/binary/main
```

If you run this, you'll see output that looks something like this:

```
Property falsified. Counterexample found.

Model:
  ZF  |->  0x0
  SF  |->  0x0
  RSP |->  0x0000000040000000
  RSI |->  0x0000000000000000
  RDX |->  0x0000000000000000
  RDI |->  0x0000000000000000
  RCX |->  0x0000000000000000
  RBP |->  0x0000000000000000
  RAX |->  0x0000000000000000
  R9  |->  0x0000000000000000
  R8  |->  0x0000000000000000
  PF  |->  0x0
  OF  |->  0x0
  CF  |->  0x0
  AF  |->  0x0
  mem_orig |-> [
    else |-> 0x00]
  mem_mod = mem_orig
  malloc_ret_RAX016 |->  0x0000000000000000
```

First, it says `Property falsified. Counterexample found.`, indicating that it did in fact find a way to trigger the null dereference. Second, it gives us a `Model`, which shows us how to do it. Notice in particular the very last line:

```
malloc_ret_RAX016 |->  0x0000000000000000
```

This is tells us that if `malloc` returns `0`, then we'll end up with a null dereference. And that's exactly right. As we saw, if `malloc` returns `NULL` (i.e., `0`, which indicates that no memory could be allocated), then trying to store a character there will segfault.

Note that we could rewrite the program at `tutorial/03/binary/main.c` so that it checks for the `NULL` memory address before trying to store something there. There is a corrected version at `tutorial/03/binary/main_fixed.c`:

```
#include <stdlib.h>

int main() {

    // Allocate a byte of memory, at address `addr`
    char *addr = malloc(sizeof(char));

    // Don't proceed if we got no address
    if (addr == NULL) { return 0; }

    // Store the character 'z' at that address
    *addr = 'z';
}
```

Let's have `wp` check *this* version of the function. The compiled program lives at `03/binary/main_fixed`. So:

```
$ bap wp \
      --func=main \
      --check-null-derefs \
      03/binary/main_fixed
```

Now `wp` gives back output that looks like this:

```
Evaluating precondition.
Checking precondition with Z3.

No counterexample found.
```

This time, `wp` says `No counterexample found`. In other words, `wp` has found that there are no null dereferences in `main`. And that is correct. There is indeed no possible way to dereference a null pointer in this version of the program, which `wp` confirms.

A.8 Comparing Function Outputs

In the previous examples, we used `wp` to examine a function in a single binary program. But, as mentioned earlier, `wp` can also be used to compare two versions of a function from two different programs.

A.8.1 Analyzing two programs

To analyze two programs, the command you want to invoke has this basic form:

```
bap wp \                        <-- The command is wp
    --func=FUNC \               <-- The function to examine in both programs
    [options] \                 <-- Any extra options
    /path/to/exe1 \             <-- The path to the first binary
    /path/to/exe2               <-- The path to the second binary
```

A.8.2 Example: Verifying an optimization

One of the predefined comparative properties that `wp` can verify is the following: `wp` can look at a function that occurs in two programs, and it can figure out if those functions can produce different outputs.

This can be very useful if, say, you optimize a function. You can ask `wp` to compare the old unoptimized version with the new optimized version, and tell you if it's possible for the two functions to give you different outputs.

Suppose we are writing a C program, and we need to know if two integers have the same sign (positive or negative). Here is a very straightforward way to write a function to check (you can find this code in `tutorial/04/binary/main_1.c`:

```c
bool same_signs(int x, int y) {

    // When x is negative
    if(x < 0){

        // They have the same sign if x is negative too
        if (y < 0) { return true; }
        else { return false; }

    // When x is positive
    } else{

        // They have the same sign if y is positive too
        if (y >= 0) { return true; }
        else { return false; }

    }
}
```

First, we check if `x` is less than zero. If it is, we then check if `y` is also less than zero. If so, they have the same sign. Alternatively, if `x` is greater than zero, then we check if `y` is also greater than zero, and again, if so, then they have the same sign.

This code is fine, but it has a lot of `if` statements, so it branches a lot, and hence is not the fastest code on the planet. Suppose we want to optimize this, and we ask a clever engineer to write a new version. Suppose they come up with the following (this version can be found in `tutorial/04/binary/main_2.c`:

```
bool same_signs(int x, int y) {
    return !((x ^ y) < 0);
}
```

This looks promising. It has none of the branches, and it uses XOR (the caret ^) as a bit trick. To understand why this code checks if the two integers have the same same sign, recall that signed integers in C are typically represented with "two's complement" encoding. In this encoding, the top bit represents the sign—it is 0 for positive numbers and 1 for negative numbers. So, after a bit-wise XOR operation, the top bit will be 1 only if the two numbers have different signs. Therefore, the integer that results from the XOR will be negative only if the original two numbers have different signs.

We want to know: is this optimized version really the same as the earlier version. That is to say, does this optimized version really produce the same outputs as the first version?

We can ask `wp` to compare these two functions, and tell us if there is any possible way for the second one to produce a different output than the first one (when given the same input).

Remember that the output of a function is always placed in the `RAX` register, so what we really want to know is if there is any way for these two functions to put different values in `RAX`, given the same inputs. This is something we can ask `wp` to check for us.

This is our first example of using `wp` to compare two programs. You can ask `wp` to check if both versions of the `same_signs` function produce the same output in `RAX`. Here is the command:

```
$ bap wp \
      --func=same_signs \
      --compare-post-reg-values=RAX \
      04/binary/main_1 \
      04/binary/main_2
```

Note the parameter `-compare-post-reg-values=RAX` in this command. That tells `wp` to compare the value in `RAX` after the functions finish execution.

When `wp` finishes its check, it prints out the following:

```
Evaluating precondition.
Checking precondition with Z3.

No counterexample found.
```

What this means is that `wp` was not able to find a way to make these two function produce different outputs. And remember, `wp` does a logical check, so this is telling us that (up to our simplifying assumptions) it is logically impossible for these two functions to produce different outputs.

That's good news. This tells us that the optimized version of the function is indeed equivalent to the original, in the sense that it will always produce exactly the same outputs as the original.

A.9 Comparing Function Calls

Another predefined property that `wp` can check for when comparing two functions is this: it can check that both versions of the functions make the same function calls in the course of their execution.

Let's illustrate this with an example. Imagine the control loop for an unmanned, underwater vehicle (UUV). This code processes command signals and dispatches tasks to the appropriate handlers.

There are various signals it can receive:

- `SURFACE`, which tells the computer to surface the UUV.
- `NAV`, which tells the computer to alter the navigation course of the UUV.
- `DEPLOY`, which tells the computer to deploy a payload (e.g., fire a missile).
- `LOG`, which tells the computer to log the current status of the system.

These various signals are encoded in an `enum`:

```
/* Different types of signals */
typedef enum {SURFACE, NAV, LOG, DEPLOY} signal_t;
```

There are handlers for each one:

```
/* Stubbed handler for the SURFACE signal */
int surface() {
    int status_code = 1;
    // Handle surfacing...
    return status_code;
}

/* Stubbed handler for the NAV signal */
int alter_course() {
    int status_code = 2;
    // Handle navigation...
    return status_code;
}

/* Stubbed handler for the LOG signal */
int log_system_status() {
    int status_code = 3;
    // Log the system's status...
    return status_code;
}

/* Stubbed handler for the DEPLOY signal */
int deploy_payload() {
    int status_code = 4;
    // Deploy the payload...
    return status_code;
}
```

To process the signal, there is a switch statement:

```
/* Process a signal, and dispatch to an appropriate handler */
int process_signal(signal_t signal) {

    int status_code = 0;

    switch (signal) {
        case SURFACE:
            status_code = surface();
            break;

        case NAV:
            status_code = alter_course();

        case LOG:
            status_code = log_system_status();
            break;

        case DEPLOY:
            status_code = deploy_payload();
            break;
    }
    return status_code;
}
```

The switch statement here simply dispatches the task to the appropriate handler, depending on which signal it receives, and then it returns the resulting status code.

But notice that there is no `break` statement in the `NAV` case. The programmer wanted to log system status after every `NAV` signal, so they cleverly left off the `break` here. That causes execution to fall through from the `NAV` case to the `LOG` case. Hence, every time this program handles a `NAV` signal, it does that, but then it falls through to `LOG`, and so it *also* logs the system status.

The full code for this example can be found at `tutorial/05/binary/main_1.c`.

Suppose now that, many years later, we have decided that we do not need to have logging in this program anymore. We might apply an automated debloating tool to remove log-related functionality. For the purposes of this example, we'll imagine this tool has a bug that results in incorrectly transformed code. We'll then see how to use CBAT to catch this bug.

First, the patch removes the `LOG` from the enum, so now we just have `SURFACE`, `NAV`, and `DEPLOY`:

```
/* Different types of signals */
typedef enum {SURFACE, NAV, DEPLOY} signal_t;
```

It also removes the `log_status()` handler, so now we just have `surface()`, `alter_course()`, and `deploy_payload()`:

```
/* Stubbed handler for the SURFACE signal */
int surface() {
```

```
    int status_code = 1;
    // Handle surfacing...
    return status_code;
}

/* Stubbed handler for the NAV signal */
int alter_course() {
    int status_code = 2;
    // Handle navigation...
    return status_code;
}

/* Stubbed handler for the DEPLOY signal */
int deploy_payload() {
    int status_code = 4;
    // Deploy the payload...
    return status_code;
}
```

And the patch removes the `LOG` case from the switch statement, so now we only have the cases for `SURFACE`, `NAV`, and `DEPLOY`:

```
/* Process a signal, and dispatch to an appropriate handler */
int process_signal(signal_t signal) {

    int status_code = 0;

    switch (signal) {
        case SURFACE:
            status_code = surface();
            break;

        case NAV:
            status_code = alter_course();

        case DEPLOY:
            status_code = deploy_payload();
            break;
    }

    return status_code;

}
```

Here, stripping out that `LOG` case introduced a bug. Before the patch, every time a `NAV` signal was processed, the `NAV` case would fall through to the `LOG` case, and log the system's status. But now, in the patched version, there is no `LOG` case, so every time a `NAV` signal gets processed, it falls right on through to the `DEPLOY` case, and deploys the payload! A correct patch would have also inserted a **`break`** statement after the `NAV` case.

The full code for the patched version can be found at `tutorial/05/binary/main_2.c`.

We can ask wp to analyze both versions of the process_signal function, and check that they call the same functions. Then wp will find out if there is any way that the original and patched versions differ with respect to function calls.

We can ask wp to compare both binaries and find out if both versions of the process_signal functions differ with respect to the function calls they make. Here is the command:

```
$ bap wp \
      --func=process_signal \
      --compare-func-calls \
      05/binary/main_1 \
      05/binary/main_2
```

Notice that we added the flag --compare-func-calls. This tells wp to check the function calls in both versions of the process_signal function.

The output looks something like this:

```
Property falsified. Counterexample found.

Model:
    ZF  |->  0x0
    SF  |->  0x0
    RSP |->  0x000000003f800084
    RSI |->  0x0000000000000000
    RDX |->  0x0000000000000000
    RDI |->  0x0000000000000001
    RCX |->  0x0000000000000000
    RBP |->  0x0000000000000000
    RAX |->  0x0000000000000000
    R9  |->  0x0000000000000000
    R8  |->  0x0000000000000000
    PF  |->  0x0
    OF  |->  0x0
    CF  |->  0x0
    AF  |->  0x0
    mem_orig |-> [
        else |-> 0x00]
    mem_mod = mem_orig
```

This says Property falsified. Counterexample found, which means wp did indeed find a way to make the older and patched versions of these functions make different function calls.

Note the Model, which tells us that this violation occurs when RDI is set to 0x01. So if the value 0x01 is stored in RDI when these functions start, then the second version of the function will call different functions than the first version.

Given the code that we looked at above, we can see that this is right. Remember that RDI always holds the first argument to a function, which in the case of process_signals is the signal:

```
/* Process a signal, and dispatch to an appropriate handler */
int process_signal(signal_t signal) {   // RDI holds the signal
```

```
    ...
}
```

What is the value `0x01`? Well, a signal is an `enum`, which we can think of as an indexed array:

```
/* Different types of signals */
// items indexed 0, 1, 2, 3
typedef enum {SURFACE, NAV, LOG, DEPLOY} signal_t;\

// or items indexed 0, 1, 2
typedef enum {SURFACE, NAV, DEPLOY} signal_t;
```

Hence, index `0x00` refers to the first option (`SURFACE`), `0x01` refers to the second option (`NAV`), and so on.

With that in mind, we can make sense of what `wp` is telling us here. It's telling us that if we call `process_signal` with the `NAV` signal (i.e., with `RDI` holding the index `0x01`), then the functions will behave differently, in the sense that they will make different functions calls.

Which function calls do they differ on, exactly? We can find out by asking `wp` to print out its refuted goals. To do that, we add the parameter `-show=refuted-goals` when we run `wp`, like this:

```
$ bap wp \
    --func=process_signal \
    --compare-func-calls \
    --show=refuted-goals \
    05/binary/main_1 \
    05/binary/main_2
```

Then `wp` prints output that looks something like this:

```
Property falsified. Counterexample found.

Model:
    ZF  |->  0x0
    SF  |->  0x0
    RSP |->  0x000000003f800084
    RSI |->  0x0000000000000000
    RDX |->  0x0000000000000000
    RDI |->  0x0000000000000001
    RCX |->  0x0000000000000000
    RBP |->  0x0000000000000000
    RAX |->  0x0000000000000000
    R9  |->  0x0000000000000000
    R8  |->  0x0000000000000000
    PF  |->  0x0
    OF  |->  0x0
    CF  |->  0x0
    AF  |->  0x0
    mem_orig |-> [
```

```
            else |-> 0x00]
        mem_mod = mem_orig

    Refuted goals:
    deploy_payload not called in modified:
        Z3 Expression: false
```

Notice at the bottom:

```
Refuted goals:
deploy_payload not called in modified
```

What this says is `deploy_payload` is called in the modified version of the function and not in the original version, when the function is called with the argument `NAV` (i.e., when `RDI` is `0x01` at the start of the function).

(Why the double negation in "Refuted goals: deploy_payload not called in modified"? This is because `wp` proves what it proves by assuming that the same functions are not called, and then it tries to falsify that assumption.)

In practice, it is difficult for humans to catch these kinds of bugs, especially when the patches are applied automatically, and to many different portions of a program all at once. `wp` can be an important help here.

A.10 Custom Postcondition (One Binary)

So far we have been looking at predefined properties that are built in to `wp`. Let's turn now to defining some custom properties.

To define a custom property of a function in your program, you describe what you want the state of your program to look like after the function finishes execution. Then, `wp` will check if that description holds true of your program. If there is a way to to falsify this property, `wp` will find it and provide you with some example inputs that will make your function exhibit that behavior.

Let's do a simple example. Consider this toy C function (which can be found at `tutorial/06/binary/main.c`):

```c
int main() {
    // Return my lucky number.
    return 7;
}
```

As a simple example of a custom property, let us say that this function should always produce `7` for its output. More exactly, let us say that when this function terminates, the value in `RAX` will always be `7` (since `RAX` is where the output of a function is always placed).

For custom properties, `wp` accepts `SMTLIB` expressions. `SMTLIB` is a special lisp-like language that was designed explicitly for stating custom properties like this. To encode our property as an `SMTLIB` expression, we first need to know the name of the register, which of course we do know:

```
RAX
```

Then, we need to know the value that we want to say should be in it. The registers for `x86_64` programs hold 64-bit binary numbers, so we want to encode 7 as a 64-bit number. For `SMTLIB`, we simply make it a 64-bit hex number, like this:

```
#x0000000000000007
```

Next, we want to say that these are equal, which we write like this:

```
(= RAX #x0000000000000007)
```

Notice that this is lisp-like. We put the equal sign up front, then we list the two arguments after it, and then we wrap the whole thing in parentheses.

Finally, we want to assert that this holds, so we wrap the whole thing in an `assert`, like this:

```
(assert (= RAX #x0000000000000007))
```

That is our complete `SMTLIB` expression. It asserts that the value in `RAX` is the 64-bit number `7`.

Now we can tell `wp` to check that this holds for our function. To do that, we ask `wp` to analyze our `main` function just as we have done before, but we will specify our custom property with the following parameter:

```
--postcond='(assert (= RAX #x0000000000000007))'
```

This tells `wp` to check that the specified property holds after the function executes. Note that we enclose the `SMTLIB` expression in single quotes. This is just to force bash to treat it as a literal string and not perform any shell expansion.

Here is the full command to run:

```
$ bap wp \
      --func=main \
      --show=refuted-goals \
      --postcond='(assert (= RAX #x0000000000000007))' \
      06/binary/main
```

When `wp` runs this, it will try to falsify this property. That is, it will explore all logical possibilities (up to our simplifying assumptions), and find a way to make our `main` function put something other than `7` in `RAX`.

- If `wp` can find a way to falsify our property, it will return `property falsified. Counterexample found` and provide an example of how to make `main` produce a value other than `7`.

- If it cannot falsify our property, it will return `No countermodel found`, and that means the property holds. It means that `RAX` does indeed always contain the 64-bit number 7 at the end of our `main` function's execution.

When you run the above command, `wp` produces output like this:

```
Evaluating precondition.
Checking precondition with Z3.

No counterexample found.
```

So, in this case, `wp` could not find a way to falsify this property, and hence the property holds. And indeed, that makes sense, because as we can see from the code, our function always returns the number `7`.

Let's have `wp` check a different property, one that we know is false. For example, let's have `wp` check whether `RAX` always ends up with the value `3` in it. To express this property, we would write an `SMTLIB` expression just like the previous one, except we'll use the number `3` instead of `7`:

```
(assert (= RAX #x0000000000000003))
```

Now have `wp` check this:

```
$ bap wp \
    --func=main \
    --show=refuted-goals \
    --postcond='(assert (= RAX #x0000000000000003))' \
    06/binary/main
```

This time, `wp` outputs something that looks like this:

```
Property falsified. Counterexample found.

Model:
  RSP |->  0x000000003f800081
  RSI |->  0x0000000000000000
  RDX |->  0x0000000000000000
  RDI |->  0x0000000000000000
  RCX |->  0x0000000000000000
  RBP |->  0x0000000000000000
  RAX |->  0x0000000000000000
  R9  |->  0x0000000000000000
  R8  |->  0x0000000000000000
  mem_orig |-> [
    else |-> 0x00]
  mem_mod = mem_orig

Refuted goals:
(= RAX0 #x0000000000000003):
  Concrete values: = #x0000000000000007 #x0000000000000003
  Z3 Expression: = #x0000000000000007 #x0000000000000003
```

This time, wp comes back with `Property falsified. Counterexample found`, meaning that it could find a way to falsify our assertion, and it shows us how to do it. In the `model`, most of the values are zero, which makes sense. Since our `main` function always returns `7`, the registers can start with pretty much any values and `main` will not produce `3` in `RAX`.

A.11 Custom Pre and Postcondition (One Binary)

Consider the following C program (the source code can be seen at `tutorial/07/binary/main.c`):

```
int main(int argc, char **argv) {
    if (argc < 10) {
        return 7;
    } else {
        return 254;
    }
}
```

In this program, you can see that if the number of arguments (`argc`) is less than `10`, the `main` function returns `7`. Otherwise it returns `254`.

Suppose we want to assert that at the end of this function `RAX` will always contain `7`. We could have `wp` check that, just as we did before:

```
$ bap wp \
      --func=main \
      --show=refuted-goals \
      --postcond='(assert (= RAX #x0000000000000007))' \
      07/binary/main
```

When you run that, you'll see that `wp` produces output that looks something like this:

```
Property falsified. Counterexample found.

Model:
  ZF  |->  0x0
  SF  |->  0x0
  RSP |->  0x000000003f800081
  RSI |->  0x0000000000000000
  RDX |->  0x0000000000000000
  RDI |->  0x0000000000400000
  RCX |->  0x0000000000000000
  RBP |->  0x0000000000000000
  RAX |->  0x0000000000000000
  R9  |->  0x0000000000000000
  R8  |->  0x0000000000000000
  PF  |->  0x0
  OF  |->  0x0
  CF  |->  0x0
  AF  |->  0x0
  mem_orig |-> [
```

```
    else |-> 0x00]
  mem_mod = mem_orig

Refuted goals:
(= RAX0 #x0000000000000007):
  Concrete values: = #x00000000000000fe #x0000000000000007
  Z3 Expression: = #x00000000000000fe #x0000000000000007
```

Here, `wp` tells us that it can falsify our assertion, and it provides an example of how to do it. Notice the `RDI` register. If it is set to `0x400000`, then our `main` function will put something other than `7` in `RAX`. (Your CBAT installation might produce a different example.)

Of course, that makes sense, since we can see in the C code that if the number of arguments is `10` or more, it'll return `254`, and `0x400000` is certainly bigger than `10`.

Let's make our custom property a little more robust. Let's suppose that we want to assert that `main` will put `7` in `RAX`, given some particular input argument. For example, let's say that, if the input argument is `5`, then `RAX` will be `7`.

We can do that by asserting a precondition for the function. In this case, we want the precondition to be that the argument to our `main` function, which is stored in the `RDI` register, is `5`. Here is how we would express that in `SMTLIB`:

```
(assert (= RDI #x0000000000000005))
```

And now we can add this as a precondition to our check, using the `-precond` parameter:

```
$ bap wp \
      --func=main \
      --show=refuted-goals \
      --precond='(assert (= RDI #x0000000000000005))' \
      --postcond='(assert (= RAX #x0000000000000007))' \
      07/binary/main
```

Now `wp` will check that the postcondition holds, given that the precondition holds. In this case, it will check that, if `RDI` is `5` at the start of the function, then `RAX` will be `7` at the end of the function.

When you run this, `wp` will output something like this:

```
Evaluating precondition.
Checking precondition with Z3.

No counterexample found.
```

In other words, `wp` was not able to find a way to falsify our assertions, and hence our assertions hold.

A.12 Custom Postcondition (Two Binaries)

In the last two examples, we looked at specifying custom properties that `wp` can check for a single binary program. We can also specify custom properties that `wp` can check to compare two binary programs.

To illustrate, consider the `same_signs` function we looked at before. Here is the first version of the function (which can be seen at `tutorial/08/binary/main_1.c`):

```
bool same_signs(int x, int y) {

    // When x is negative
    if(x < 0){

        // They have the same sign if x is negative too
        if (y < 0) { return true; }
        else { return false; }

    // When x is positive
    } else{

        // They have the same sign if y is positive too
        if (y >= 0) { return true; }
        else { return false; }

    }
}
```

And here's the optimized version (which can be seen at `tutorial/08/binary/main_2.c`):

```
bool same_signs(int x, int y) {
    return !((x ^ y) < 0);
}
```

Before, we asked `wp` to check that these two functions produce the same output in `RAX`, using the `-compare-post-reg-values=RAX` parameter. But we can specify this same thing as a custom property.

What we want to say is that the two versions of `same_signs` always put the same value in `RAX`. Put another way, we want to say that at the end of the *original* function's execution, `RAX` is the same as it is at the end of the *modified* function's execution.

To explicitly refer to a register in the original function, we can append `_orig` to the register name, like this:

```
RAX_orig
```

Similarly, to explicitly refer to a register in the modified version of the function, we can append `_mod`, like this:

```
RAX_mod
```

Then, we can formulate an SMTLIB expression asserting that the two are equal:

```
(assert (= RAX_orig RAX_mod))
```

To have wp check this, we call wp in comparison mode, with our custom postcondition:

```
$ bap wp \
      --func=same_signs \
      --show=refuted-goals \
      --postcond='(assert (= RAX_orig RAX_mod))' \
      08/binary/main_1 \
      08/binary/main_2
```

When you run this, wp outputs the following:

```
Evaluating precondition.
Checking precondition with Z3.

No counterexample found.
```

In other words, wp could not find a way to falsify our assertion. Hence, the property we've asserted here holds. RAX in the original and the modified program will indeed always be equal.

A.13 Reference Guide

We introduced many command line options for CBAT. A complete reference for the command line interface, including many options not described in this introduction tutorial, can be found in Appendix B. This section also includes a guide to common SMTLIB expressions for use in your custom properties.

A.14 Tutorial Conclusion

As we noted at the outset, CBAT is a family of tools designed to verify the behavior of binary programs.

At the heart of the CBAT toolset is wp. As we have seen, wp is a tool that verifies whether functions behave in specified ways. wp can be used to both to verify the behavior of a function in a single program and to compare the behavior of functions from two programs.

Because of its ability to compare two programs, wp can be used to check patches and modifications that you might make to your binary programs. Patches are meant to change some aspects of a program, while preserving certain other behaviors. With wp, you can check whether the patched version of your program preserves the desired behavior that is present in the original version.

B CBAT Feature Reference

This section contains a brief reference to CBAT's command line options.

B.1 Viewing the Man Page

To view the man page:

```
bap wp --help
```

B.2 General Options

These options apply in most circumstances to adjust CBAT's behavior or output.

- `--func=<function-name>` — Determines which function to verify. WP verifies a single function, though calling it on the `main` function along with the `inline` option will analyze the whole program. If no function is specified or the function cannot be found in the binary/binaries, WP will exit with an error message.
- `--inline=<posix-regexp>` — Functions specified by the provided POSIX regular expression will inlined. When functions are not inlined, heuristic function summaries are used at function call sites. For example, if you want to inline the functions `foo` and `bar`, you can write `--inline=foo|bar`. To inline everything, use `--inline=.*` (not generally recommended).
- `--pointer-reg-list=<reg-list>` — This flag specifies a comma delimited list of input registers to be treated as pointers at the start of program execution. This means that these registers are restricted in value to point to memory known to be initialized at the start of the function. For example, `RSI,RDI` would specify that `RSI` and `RDI`'s values should be restricted to initialized memory at the start of execution.
- `--num-unroll=<num>` — Specifies the number of times to unroll each loop. WP will unroll each loop 5 times by default.
- `--gdb-output=<filename.gdb>` — When WP finds a countermodel, outputs a `gdb` script to file `filename.gdb`. From within `gdb`, run `source filename.gdb` to set a breakpoint at the function given by `--func` and fill the appropriate registers with the values found in the countermodel. In the case WP cannot falsify a property or returns UNKNOWN, no script will be outputted.
- `--bildb-output=<filename.yml>` — When WP finds a countermodel, outputs a BILDB initialization script to file `filename.yml`. This YAML file sets the registers and memory to the values found in the countermodel, allowing BILDB to follow the same execution trace. In the case the analysis cannot falsify the property or returns UNKNOWN, no script will be outputted.
- `--use-fun-input-regs` — At a function call site, uses all possible input registers as arguments to a function symbol generated for an output register. If this flag is not present, no registers will be used.

- --stack-base=<address> — Sets the location of the stack frame for the function under analysis. By default, WP assumes the stack frame for the current function is between `0x40000000` and `0x3F800080`.

- --stack-size=<size> — Sets the size of the stack. `size` should be denoted in bytes. By default, the size of the stack is `0x800000`, which is 8MB.

- --ext-solver-path=</bin/boolector> — Allows the usage of an external SMT solver. This option has only been tested with Boolector version 3.2.1. For other solvers, results may vary.

- `--show=[bir|refuted-goals|paths|precond-internal|precond-smtlib]` — A list of details to print out from the analysis. Multiple options can be specified as a comma-separated list. For example: `--show=bir,refuted-goals`. The options are:
  - `bir`: The code of the binary/binaries in BAP Intermediate Representation.
  - `refuted-goals`: In the case WP finds a countermodel, a list of goals refuted in the model that contains their tagged names, their concrete values, and their Z3 representation.
  - `paths`: The execution path of the binary that results in a refuted goal. The path contains information about the jumps taken, their addresses, and the values of the registers at each jump. This option automatically prints out the refuted goals.
  - `precond-smtlib`: The precondition printed out in Z3's SMT-LIB2 format.
  - `precond-internal`: The precondition printed out in WP's internal format for the `Constr.t` type.

- `--debug=[z3-solver-stats|z3-verbose|constraint-stats|eval-constraint-stats]` — A list of debugging statistics to display. Multiple statistics may be specified in a comma-separated list. For example: `--debug=z3-solver-stats,z3-verbose`. The options are:
  - `z3-solver-stats`: Information and statistics about Z3's solver. It includes information such as the maximum amount of memory used and the number of allocations.
  - `z3-verbose`: Z3's verbosity level. It outputs information such as the tactics the Z3 solver used.
  - `constraint-stats`: Statistics regarding the internal `Constr.t` data structure, including the number of goals, ITEs, clauses, and substitutions.
  - `eval-constraint-stats`: Statistics regarding the internal expression lists during evaluation of the `Constr.t` data type.

- --user-func-specs=<user-spec-list> — List of user-defined subroutine specifications. For each subroutine, it creates the weakest precondition given the name of the subroutine and its pre and post-conditions. Usage:

```
--user-func-specs="<sub name>,<precondition>,<postcondition>"
```

For example, "foo,(assert (= RAX RDI)),(assert (= RAX init_RDI))" means "for subroutine named foo, specify that its precondition is RAX = RDI and its postcondition is RAX = init_RDI". Multiple subroutine specifications are delimited with ';'s.

- --fun-specs=<spec-list> — List of built-in function summaries to be used at a function call site in order of precedence. A target function will be mapped to a function spec if it fulfills the spec's requirements. All function specs set the target function as called and update the stack pointer. The default specs set are verifier-assume, varifier-nondet, empty, and chaos-caller-saved. Note that if a function is set to be inlined, it will not use any of the following function specs. Available built-in specs:
  - verifier-error: Used for calls to __VERIFIER_error and __assert_fail. Looks for inputs that would cause execution to reach these functions.
  - verifier-assume: Used for calls to __VERIFIER_assume. Adds an assumption to the precondition based on the argument to the function call.
  - verifier-nondet: This option should be used for calls to nondeterministic functions such as __VERIFIER_nondet_*, calloc, and malloc. Chaoses the output to the function call representing an arbitrary pointer.
  - afl-maybe-log: Used for calls to __afl_maybe_log. Chaoses the registers RAX, RCX, and RDX.
  - arg-terms: Used when BAP's uplifter returns a nonempty list of input and output registers for the target function. Chaoses this list of output registers.
  - chaos-caller-saved: Used for the x86 architecture. Chaoses the caller-saved registers.
  - rax-out: Chaos RAX if it can be found on the left-hand side of an assignment in the target function.
  - chaos-rax: Chaos RAX regardless if it has been used on the left-hand side of an assignment in the target function.
  - empty: Used for empty subroutines. Performs no actions.

B.3 Single Program Analysis

To analyze a single program, the command you want to invoke has this basic form:

```
bap wp \                  <-- The command is wp
    --func=FUNC \         <-- The function to examine
    [options] \           <-- Any extra options
    /path/to/exe          <-- The path to a binary program
```

The following options select or adjust the property that will be checked for the function FUNC in the case of single-program analysis.

- --trip-asserts — Looks for inputs to the subroutine that would cause an __assert_fail or __VERIFIER_error to be reached.

- `--check-null-derefs` — Checks for inputs that would result in dereferencing a NULL address during a memory read or write. This flag can also be used in comparative analysis.
- `--precond=<smt-lib-string>` — Allows you to specify an assertion that WP will assume is true at the beginning of the function it is analyzing. Assertions are specified in the smt-lib2 format. If no precondition is specified, a trivial precondition of `true` will be used. This flag can also be used in comparative analysis.
- `--postcond=<smt-lib-string>` — Allows you to specify an assertion that WP will assume is true at the end of the function it is analyzing, using the smt-lib2 format. If no postcondition is specified, a trivial postcondition of `true` will be used. This flag can also be used in comparative analysis.
- `--loop-invariant=<s-expression>` — Usage:

  ```
  (((address <addr>) (invariant <smtlib>)) (...))
  ```

  Assumes the subroutine contains unnested while loops with one entry point and one exit each. Checks the loop invariant written in smt-lib2 format for the loop with its header at the given address. The address should be written in BAP's bitvector string format. Only supported for a single binary analysis.

B.4 Comparative Analysis

To analyze two programs,the command you want to invoke has this basic form:

```
bap wp \                        <-- The command is wp
    --func=FUNC \               <-- The function to examine in both programs
    [options] \                 <-- Any extra options
    /path/to/exe1 \             <-- The path to the first program
    /path/to/exe2               <-- The path to the second program
```

- `--check-null-derefs` — Checks that the modified binary has no additional paths with null dereferences in comparison with the original binary.
- `--check-invalid-derefs` — Checks that the modified binary has no additional paths that result in dereferences to invalid memory locations. That is, all memory dereferences are either on the stack or heap. The stack is defined as the memory region above the current stack pointer, and the heap is defined as the memory region 0x256 bytes below the lowest address of the stack.
- `--compare-func-calls` — Checks that function calls do not occur in the modified binary if they have not occurred in the original binary.
- `--compare-post-reg-values=<reg-list>` — Compares the values stored in the registers specified in `reg-list` at the end of the function's execution. For example, `RAX,RDI` compares the values of RAX and RDI at the end of execution. If unsure about which registers to compare, check the architecture's ABI. x86_64 architectures place their output in RAX and ARM architectures place their output in R0.

- --user-func-specs-orig=<user-spec-list> — List of user-defined subroutine specifications to be used only for the original binary in comparative analysis. For usage, see --user-func-specs.
- --user-func-specs-mod=<user-spec-list> — List of user-defined subroutine specifications to be used only for the modified binary in comparative analysis. For usage, see --user-func-specs.
- --mem-offset — Maps the symbols in the data and bss sections from their addresses in the original binary to their addresses in the modified binary. If this flag is not present, WP assumes that memory between both binaries start at the same offsets.
- --rewrite-addresses — This flag is only used in a comparative analysis. Rewrites the concrete addresses in the modified binary to the same address in the original binary if they point to the same symbol. This flag should not be used in conjunction with the --mem-offset flag.
- --precond=<smt-lib-string> — Allows you to specify an assertion that WP will assume is true at the beginning of the function it is analyzing, using the smt-lib2 format. For comparative predicates, one may refer to variables in the original and modified programs by appending the suffix _orig and _mod to variable names in the smt-lib expression. For example, --precond="(assert (= RDI_mod #x0000000000000003)) (assert (= RDI_orig #x0000000000000003))". If no precondition is specified, a trivial precondition of true will be used.
- --postcond=<smt-lib-string> — Allows you to specify an assertion that WP will assume is true at the end of the function it is analyzing, using the smt-lib2 format. Similar to --precond, one may create comparative postconditions on variables by appending _orig and _mod to variable names. If no postcondition is specified, a trivial postcondition of true will be used.
- --func-name-map=<regex-orig>,<regex-mod> — Maps the subroutine names from the original binary to their names in the modified binary based on the regex from the user. Usage:

  ```
  --func-name-map="<regex for original name>,<regex for modified name>"
  ```

  For example:

  ```
  --func-name-map="\(.*\),foo_\1"
  ```

  means that all subroutines in the original binary have foo_ prepended in the modified binary. Multiple patterns can be used to map function names and are delimited with ';'s (i.e. <reg1_orig>,<reg1_mod>;<reg2_orig>,<reg2_mod>"). By default, WP assumes subroutines have the same names between the two binaries.

B.5 SMTLIB Cheat Sheet

Custom properties use the smt-lib2 format. Here we provide a quick reference to commonly used features.

In the following, let E1, E2, and so on refer to SMTLIB expressions.

B.5.1 Assertion

- Assert: `(assert E1)`

B.5.2 Boolean Operations

- Conjunction: `(and E1 E2)`
- Disjunction: `(or E1 E2)`
- Implies: `(=> E1 E2)`
- Negation: `(not E1)`

B.5.3 Arithmetic Operations

- Equality: `(= E1 E2)`
- Inequality: `(not (= E1 E2))`
- Addition: `(+ E1 E2)`
- Subtraction: `(- E1 E2)`
- Multiplication: `(* E1 E2)`
- Less than: `(< E1 E2)`
- Less-than-or-equal-to: `(<= E1 E2)`
- Greater than: `(> E1 E2)`
- Greater-than-or-equal-to: `(>= E1 E2)`

B.5.4 Bitvector Operations

- Bitvector addition: `(bvadd E1 E2)`
- Bitvector subtraction: `(bvsub E1 E2)`
- Bitvector unsigned less-than: `(bvult E1 E2)`
- Bitvector unsigned less-than-or-equal-to: `(bvule E1 E2)`
- Bitvector unsigned greater-than: `(bvugt E1 E2)`
- Bitvector unsigned greater-than-or-equal-to: `(bvuge E1 E2)`
- Bitvector signed less-than: `(bvslt E1 E2)`
- Bitvector signed less-than-or-equal-to: `(bvsle E1 E2)`
- Bitvector signed greater-than: `(bvsgt E1 E2)`
- Bitvector signed greater-than-or-equal-to: `(bvsge E1 E2)`

C Using BAP: An Introduction and Reference

BAP is a complex system with many interaction points and a sophisticated mechanism for storing and retreiving program semantics. In this section, we include a reference to the use of BAP that we have developed over the course of the program.

This guide is intended as a "starter guide" to help navigate the BAP ecosystem and get developers started with common use cases. We have found that this guide is a handy internal reference, and it helps new developers come up to speed more quickly. It also serves as a useful guide to BAP's knowledge-base for semantics, a powerful feature available only in the more recent 2.0+ versions of BAP. The knowledge-base for semantics sets BAP apart from other binary analysis kits.

We have made this guide available on the public CBAT Github site. We hope that it can help other programmers get up to speed with BAP, potentially increasing wider adoption of BAP. Since this guide is open-source, other BAP users can make pull requests, so the community can collectively improve this "BAP book."

C.1 Preliminaries

C.1.1 Documentation and Help

Useful links

- https://binaryanalysisplatform.github.io/bap/api/master/index.html - The official BAP documentation.
- https://ocaml.janestreet.com/ocaml-core/latest/doc/ - Jane Street core documentation.
- https://github.com/BinaryAnalysisPlatform/bap-tutorial - The BAP tutorial

The command line tool

For help with the command line tool:

```
bap --help
```

List the installed plugins:

```
bap list plugins
```

View the help of any plugin:

```
bap --PLUGIN-help
```

where PLUGIN should be replaced by the name of one of the plugins, e.g., `bap -print-help`.

To run the bap command line tool in debug mode, set the environment variable:

```
BAP_DEBUG=1
```

Bap writes its logs to:

```
${XDG_STATE_HOME}/bap/log
```

E.g.:

```
~/.local/state/bap/log
```

More information about system directories that BAP utilizes can be found in the (Bap_main.Extension.Configurati
main/Bap_main/Extension/Configuration/index.html] module.

C.1.2 Running BAP with Docker

Instructions for installing BAP manually can be found on the official BAP website: https://github.com/BinaryAnaly

BAP releases a docker image containing the latest version of the master branch:

```
docker pull binaryanalysisplatform/bap:latest
```

Run the container:

```
docker run --rm -ti binaryanalysisplatform/bap:latest bash
```

Then confirm that the relevant executables are present:

```
bap --version
bapbuild -help
bapbundle -help
```

To exit:

```
exit
```

To mount your `${HOME}` directory in the container:

```
docker run --rm -ti -v ${HOME}:/external -w /external binaryanalysisplatform/bap:latest
```

Then your home directory will be available at `/external` inside the container:

```
pwd
ls
```

For example, if you have a file at `${HOME}/foo.txt` on your local computer, you can see it and edit it from inside your container:

```
cat /external/foo.txt
```

C.1.3 Using the CLI

The CLI help

See the main help/usage:

```
bap --help
```

BAP commands

The `bap` command is a parent command for various subcommands.

List all available subcommands:

```
bap list commands
```

To see the help/usage for any subcommand:

```
bap SUBCOMMAND --help
```

Where `SUBCOMMAND` is replaced by one of the subcommands, e.g.:

```
bap mc --help
```

Dissassemble and lift

To disassemble and lift a binary executable:

```
bap disassemble /path/to/exe
```

For instance:

```
bap disassemble /bin/true
```

If you run the above command, you will see no output, but BAP did in fact process `/bin/true`, and it stored what it learned in a cache so that the next time you look at `/bin/true` with BAP, it doesn't have to repeat the whole analysis all over again.

To clean the cache:

```
bap cache --clean
```

The default is `disassemble`

If you do not specify a subcommand, BAP will just use `disassemble`.

So, for instance, you can disassemble a binary executable like this:

```
bap disassemble /path/to/exe
```

Or you can just omit `disassemble` and type this instead:

```
bap /path/to/exe
```

Most examples (in BAP documentation, for example) omit `disassemble` in this manner.

The lifted IR

To see the lifted IR of a program that BAP has disassembled, add `-dump bir` to the `disassemble` command:

```
bap disassemble /bin/true --dump bir
```

As shortcut, use `-dbir` instead of `-dump bir`:

```
bap disassemble /bin/true -dbir
```

And as an even shorter shortcut, omit `disassemble`:

```
bap /bin/true -dbir
```

The CFG

To see the CFG that BAP generates for a binary executable:

```
bap /path/to/exe -dcfg
```

Which is shorthand for:

```
bap disassemble /path/to/exe --dump cfg
```

BAP outputs the CFG as a `.dot` file. You can save it into a file if you like:

```
bap /path/to/exe -dcfg > out.dot
```

Then cut and paste the contents of `out.dot` into a dot viewer such as dreampuf.github.io/GraphvizOnline.

The assembly of a program

To look at the assembly of a program that BAP has lifted:

```
bap /path/to/exe -dasm
```

Which is really just a shorthand for:

```
bap disassemble /path/to/exe --dump asm
```

Using `bap objdump`

When BAP uses the `disassemble` subcommand to lift a binary program, it builds a control flow graph. If you just want to look at each instruction one after another (a linear sweep), use the `objdump` subcommand:

```
bap objdump /path/to/exe --show-insn=asm
```

To show the ADT instead:

```
bap objdump /path/to/exe --show-insn=adt
```

To show the binary data itself:

```
bap objdump /path/to/exe --show-insn=bin
```

And pipe it through octal dump for display in a terminal:

```
bap objdump /path/to/exe --show-insn=bin | od -h
```

Using `bap mc`

To see how BAP disassembles a particular stream of bytes, use the `bap mc` command. For instance:

```
bap mc --show-insn=asm -- 48 83 ec 08
```

C.1.4 Using BAP in utop

Setup

Open `utop-full` (do not use `utop`):

```
utop-full
```

Then load `bap.top`:

```
#require "bap.top";;
```

It will print a warning about `Core.Time_ns.Ofday.pp`, but you can ignore this. `bap.top` will define a series of custom pretty printers instead, and it will also initialize BAP (it runs `Bap_main.init`).

Tell `utop` to use `topfind`:

```
#use "topfind";;
```

Load `core_kernel` and `bap`:

```
#require "core_kernel";;
#require "bap";;
```

Open `Core_kernel` and `Bap.Std`:

```
open Core_kernel;;
open Bap.Std;;
```

Now you're ready to start exploring BAP in utop.

Load a binary executable

Define a function that can load a binary executable:

```
let load_exe (filename : string) : project =
  let input = Project.Input.file ~loader:"llvm" ~filename in
  match Project.create input ~package:filename with
  | Ok proj -> proj
  | Error e -> failwith (Error.to_string_hum e)
```

Load an executable, e.g.:

```
let proj = load_exe "/bin/true";;
```

Now you can explore the project. For instance, extract the lifted program:

```
let prog = Project.program proj;;
```

And print it:

```
Format.printf "%a" Program.pp prog;;
```

Exiting utop

To quit `utop`:

```
#quit
```

C.1.5 Using BAP as a library

BAP can be used as a library. Before calling any code from the library, first call `Bap_main.init ()` to initialize the library, otherwise undefined behavior could (silently) occur.

Example

In a new folder somewhere, create a file called `main.ml`, with these declarations at the top:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
```

Call `Bap_main.init`:

```
let () = match Bap_main.init () with
  | Ok () -> ()
  | Error e ->
    failwith (Format.asprintf "%a" Bap_main.Extension.Error.pp e)
```

Add a function to load a binary program:

```
let load_exe (filename : string) : project =
  let input = Project.Input.file ~loader:"llvm" ~filename in
  match Project.create input ~package:filename with
  | Ok proj -> proj
  | Error e -> failwith (Error.to_string_hum e)
```

Finally, get the filepath to an executable from `argv`, load the program, and print the target architecture:

```
let () =
  let filepath =
    if Array.length Sys.argv <= 1 then
      failwith "Argument missing: specify a /path/to/exe"
    else
      Sys.argv.(1)
  in
  Format.printf "Loading: %s\n%!" filepath;
  let project = load_exe filepath in
  let target = Project.target project in
  Format.printf "Target architecture: %a\n%!" T.Target.pp target
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error e ->
    failwith (Format.asprintf "%a" Bap_main.Extension.Error.pp e)

let load_exe (filename : string) : project =
  let input = Project.Input.file ~loader:"llvm" ~filename in
  match Project.create input ~package:filename with
  | Ok proj -> proj
  | Error e -> failwith (Error.to_string_hum e)

let () =
  let filepath =
    if Array.length Sys.argv <= 1 then
      failwith "Argument missing: specify a /path/to/exe"
    else
      Sys.argv.(1)
  in
  Format.printf "Loading: %s\n%!" filepath;
  let project = load_exe filepath in
  let target = Project.target project in
  Format.printf "Target architecture: %a\n%!" T.Target.pp target
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE) -- /bin/true
```

Build the program:

```
make build
```

Run it on, say, `/bin/true`:

```
dune exec ./main.exe -- /bin/true
```

It will print the target architecture, e.g.:

```
Loading: /bin/true
Target architecture: bap:amd64
```

Clean up:

```
make clean
```

Documentation

The above example does not handle any caching. For a more sophisticated example, see the disassemble plugin. For more about using BAP as a library, see the documentation.

C.2 Extending BAP

C.2.1 Plugins

BAP can be extended in various ways with your own custom code.

A package that you create that extends BAP is called a *plugin*.

To see a list of available plugins for your local BAP installation:

```
bap list plugins
```

Plugins are built (compiled) with the `bapbuild` tool that ships with BAP:

```
bapbuild -help
```

Plugins are installed with the `bapbundle` tool that ships with BAP:

```
bapbundle -help
```

Note: to avoid undefined behavior, always tell `bapbuild` to use `ocamlfind` and the `findlib.dynload` library, e.g.:

```
bapbuild -use-ocamlfind -package findlibe.dynload ...
```

A hello world example

Create a file (in some folder) called `plugin00.ml` with these contents:

```
let () = print_endline "Hello, world!"
```

Create a `Makefile`:

```
NAME := plugin00
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
```

```
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle install $(PLUGIN)
```

To build and install it:

```
make
```

To accomplish this, the `Makefile` first builds the plugin:

```
bapbuild -use-ocamlfind -package findlib.dynload plugin00.plugin
```

Then it installs it:

```
bapbundle install plugin00.plugin
```

At this point, the plugin has been installed in your local BAP plugin ecosystem.

Note that plugin code is just this:

```
let () = print_endline "Hello, world!"
```

This code simply tells the system to print "Hello, world!" whenever this module is executed.

But when is it executed?

This module has been installed as a plugin, so it will be exeuted every time the `bap` command line tool is invoked.

To confirm that, try any BAP command, e.g.:

```
bap disassemble /bin/true
```

Notice that BAP prints "Hello, world!" This demonstrates that BAP has indeed loaded the plugin when it starts up.

Obviously, this plugin is useless. A more useful plugin will hook into one of BAP's extension points. But more on that later. This example is just for illustration.

To check that a plugin is installed, list all plugins:

```
bap list plugins
```

In this case, you should see that `plugin00` is on the list.

To uninstall and clean:

```
make uninstall clean
```

To carry this out, the `Makefile` first uninstalls the plugin:

```
bapbundle remove plugin00.plugin
```

Then it cleans the workspace:

```
bapbuild -clean
```

To confirm that a plugin is no longer installed, list all plugins:

```
bap list plugins
```

You should then see that `plugin00` is no longer on the list.

Underscores in plugin names

You can use underscores in the filenames. Rename `plugin00.ml` to `plugin_00.ml`, and in the `Makefile`, change `NAME := plugin00` to `NAME := plugin_00`. Then build and install again:

```
make
```

Look at the plugin that BAP has installed now:

```
bap list plugins
```

Notice that BAP lists it as `plugin-00`. When you build and install a plugin from a file `NAME.ml`, BAP will name the plugin `NAME`, but it will replace underscores with hyphens.

Clean and uninstall:

```
make uninstall clean
```

Custom plugin names and descriptions

A custom name and description can be added to a plugin, using `bapbundle update`. Change the `install` target in the `Makefile` to this:

```
install: build
        bapbundle update -name "my-plugin-00" $(PLUGIN)
        bapbundle update -desc "My hello-world plugin" $(PLUGIN)
        bapbundle install $(PLUGIN)
```

Build and install it:

```
make
```

Look at the name of the plugin:

```
bap list plugins
```

It is listed as `my-plugin-00`, and it has a description `My hello-world plugin`.

To remove a plugin with a custom name, you must refer to it by `FILENAME.plugin`, not `CUSTOM_NAME.plugin`. For instance, try this:

```
bapbundle remove my-plugin-00.plugin
```

Notice that `bapbundle` executes this without error, and it returns a zero exit code, as if to indicate "success":

```
echo $?
```

However, the plugin has not been removed. Confirm this by listing the plugins:

```
bap list plugins
```

To truly uninstall the plugin, you must uninstall `FILENAME.plugin`:

```
bapbundle remove plugin_00.plugin
```

Now you can see that the plugin has been removed:

```
bap list plugins
```

The `Makefile` calls the correct `bapbundle remove` command.

C.2.2 Organizing files (in plugins)

As far as `bapbuild` is concerned, there is one entry point into a plugin: the plugin `NAME.plugin` corresponds to the file `NAME.ml`.

Other files/modules can be organized into subfolders, or included as a library.

Subfolders

In a folder, create a `Makefile`:

```
NAME := plugin_01
PUBLIC_NAME := my-plugin-01
PUBLIC_DESC := My hello world plugin 01

SRC := $(NAME).ml
PLUGIN := $(NAME).plugin
FLAGS := -I lib


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(FLAGS) $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Create a file called `plugin_01.ml`:

```
let () = Utils.show "Hello world, again!"
```

And in a subfolder called `lib`, create a file called `utils.ml`:

```
let show msg = Format.printf "- %s\n%!" msg
```

Here we have a `plugin_01.ml` file, which invokes a function `show` from a `Utils` module. That module is defined in `utils.ml` in a `lib` folder.

To tell `bapbuild` where to find this file, we add `-I lib` to the `bapbuild` command.

To build and install:

```
make
```

Confirm that the plugin has been installed:

```
bap list plugins
```

Uninstall and clean:

```
make uninstall clean
```

A custom (toy) library

Create a folder called `toy_lib`, and add a file `events.ml`:

```
let report msg = Format.eprintf "[Event was logged] %s\n%!" msg
```

Add a `dune` file:

```
(library
  (name toy_lib)
  (public_name toy-lib)
  (libraries findlib.dynload))
```

Add a `toy-lib.opam` file:

```
opam-version: "2.0"
name: "toy-lib"
synopsis: "A toy library"
maintainer: "Somebody"
authors: "Somebody"
homepage: "http://somewhere.com"
bug-reports: "somebody@gmail.com"
build: [[make]]
```

And add a `Makefile`:

```
LIB := toy-lib


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

.PHONY: all
all: clean uninstall build install
```

```
#################################################
# CLEAN
#################################################

.PHONY: clean
clean:
	dune clean


#################################################
# BUILD
#################################################

build: $(LIB_SRC_FILES)
	dune build -p $(LIB) @install
	@echo "" # Force a newline in terminal output


#################################################
# INSTALL
#################################################

install: build
	dune install

uninstall: build
	dune uninstall
```

Build and install this toy library:

```
make
```

Confirm that `ocamlfind` sees it:

```
ocamlfind query toy-lib
```

If you would like to install it with `opam`:

```
opam install .
```

To try it out, open `utop`:

```
utop
```

Set up `topfind`:

```
#use "topfind";;
```

Import the library:

```
#require "toy-lib";;
```

Use it:

```
Toy_lib.Events.report "Hello, world!";;
```

Quit `utop`:

```
#quit
```

If you want to remove the library later, run:

```
make uninstall clean
```

If you installed it with opam, uninstall it from opam too:

```
opam remove .
```

Include the library in a BAP plugin

In a new folder, create a file `plugin_02.ml`:

```
let () = Toy_lib.Events.report "Hello world, yet again!"
```

Add a `Makefile`:

```
NAME := plugin_02
PUBLIC_NAME := my-plugin-02
PUBLIC_DESC := My hello world plugin 02

SRC := $(NAME).ml
PLUGIN := $(NAME).plugin
PKGS := -pkgs toy-lib

#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild $(PKGS) $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Note that we tell `bapbuild` to include our `toy-lib` package by adding a `-pkgs toy-lib` parameter.

Build and install the plugin (assuming `toy-lib` has been installed as was done above):

```
make
```

Confirm that the plugin is installed:

```
bap list plugins
```

Uninstall and clean:

```
make uninstall clean
```

NOTE: if you change the `toy-lib` library, you must *first* uninstall the BAP plugin, and *then* rebuild it. That will ensure that the new version of the `toy-lib` library will get re-packaged into the plugin.

C.2.3 BAP Extensions

Use `Bap_main.Extension.declare` to declare your own custom extensions to BAP.

In a new folder, create a file `extension_00.ml`. In it, create a function that will run your own extension code, e.g.:

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  print_endline "My extension runs";
  Ok ()
```

Next, declare the function as an extension of BAP:

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  print_endline "My extension runs";
  Ok ()

let () = Bap_main.Extension.declare run
```

This tells BAP that the `run` function is an extension that it should execute whenever it runs.

Create a `Makefile`:

```
PUBLIC_NAME := my-extension-00
PUBLIC_DESC := My demo extension 00

NAME := extension_00
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install
```

```
##################################################
# THE PLUGIN
##################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Confirm that your plugin is installed:

```
bap list plugins
```

Run `bap`, e.g.:

```
bap /bin/true
```

BAP will print `My extension runs`, which confirms that it executed the extension.

Uninstall and clean:

```
make uninstall clean
```

C.2.4 Extension errors

The signature of an extension function is this:

```
Bap_main.ctxt -> (unit, Bap_main.error) Stdlib.result
```

Note: * It takes one argument, namely a context that BAP passes it when it executes the extension. * It returns a status: either `unit` if all is okay, or a `Bap_main.error`.

The `Bap_main.error` is an alias for `Bap_main.Extension.Error.t`, which is an extensible type. You can define your own constructors for it.

In a new folder, create a file `extension_01.ml`, and add a `Fail` constructor:

```
type Bap_main.Extension.Error.t += Fail of string
```

Now we can invoke `Fail "some message"` to create a new `Bap_main.error`.

Next, add a custom error-printer function:

```
let print_error (e : Bap_main.Extension.Error.t) : string option =
  match e with
  | Fail s -> Some (Format.sprintf "We encountered an error: %s" s)
  | _ -> None
```

Add an extension function that returns a `Fail` error:

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Error (Fail "could not initialize extension")
```

Finally, register the custom printer, and declare the extension:

```
let () =
  Bap_main.Extension.Error.register_printer print_error;
  Bap_main.Extension.declare run
```

The entire `extension_01.ml` file now looks like this:

```
type Bap_main.Extension.Error.t += Fail of string

let print_error (e : Bap_main.Extension.Error.t) : string option =
  match e with
  | Fail s -> Some (Format.sprintf "We encountered an error: %s" s)
  | _ -> None

let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Error (Fail "could not initialize extension")

let () =
  Bap_main.Extension.Error.register_printer print_error;
  Bap_main.Extension.declare run
```

Create a `Makefile`:

```
PUBLIC_NAME := my-extension-01
PUBLIC_DESC := My demo extension 01

NAME := extension_01
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install
```

```
########################################################
# THE PLUGIN
########################################################

.PHONY: clean
clean:
        bapbuild -clean

uninstall:
        bapbundle remove $(PLUGIN)

build: $(SRC)
        bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
        bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
        bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
        bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Run BAP, e.g.:

```
bap /bin/true
```

BAP will print the custom error message, and exit.

Uninstall and clean:

```
make uninstall clean
```

C.2.5 Passes

A pass is an analysis of a program that you can trigger from the command line.

In a new folder, create a file `pass_00.ml`, with a function that can process a BAP project:

```
open Bap.Std

let pass (proj : Project.t) : unit =
  print_endline "Running my pass: hello, world!"
```

Next, define an extension, and have the extension register the pass:

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' pass;
  Ok ()
```

Note the tick-mark on `Project.register_pass'`. That registers a pass that returns `unit`.

Finally, declare the extension:

```
let () = Bap_main.Extension.declare run
```

The whole `pass_00.ml` file looks like this:

```
open Bap.Std

let pass (proj : Project.t) : unit =
  print_endline "Running my pass: hello, world!"

let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' pass;
  Ok ()

let () = Bap_main.Extension.declare run
```

Create a `Makefile`:

```
PUBLIC_NAME := my-pass-00
PUBLIC_DESC := My demo pass 00

NAME := pass_00
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Confirm that the plugin is installed:

```
bap list plugins
```

Now run the pass over some binary executable, e.g.:

```
bap /bin/true --my-pass-00
```

The pass is triggered by the `-my-pass-00` flag. If you omit the flag, BAP will not run your pass. For instance, no message will be printed out if you run this:

```
bap /bin/true
```

Clean up:

```
make uninstall
make clean
```

C.2.6 Multiple Passes

Passes that return `unit`

Suppose you have a pass:

```
let pass (proj : Project.t) : unit =
  print_endline "Running my pass: hello, world!"
```

Note that this pass returns `unit`. To register it, we declare an extension, and declare it with the `Project.register_pass'` function (note the tick mark):

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' pass;
  Ok ()
```

Passes that return updated projects

Suppose instead that you want to write a pass that alters the project somehow, and you want to return the altered version of the project so that other passes can use it. We can register that sort of pass with `Project.register_pass` (note the absense of a tick mark).

In a new folder, create a file `pass_01.ml`, with a function that takes a BAP project and returns a new, updated project:

```
open Bap.Std

let pass (proj : Project.t) : Project.t =
  print_endline "Running my pass: hello, world!"
  (* do something to update the project *)
  (* now return the project *)
  proj
```

Next, define an extension, and have the extension register the pass:

```
let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass pass;
  Ok ()
```

Note that we use `Project.register_pass` instead of `Project.register_pass'` to register this pass.

Finally, declare the extension:

```
let () = Bap_main.Extension.declare run
```

The whole `pass_01.ml` file looks like this:

```
open Bap.Std

let pass (proj : Project.t) : Project.t =
  print_endline "Running my pass: hello, world!"
  (* do something to update the project *)
  (* now return the project *)
  proj

let run (_ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass pass;
  Ok ()

let () = Bap_main.Extension.declare run
```

Create a `Makefile`:

```
PUBLIC_NAME := my-pass-01
PUBLIC_DESC := My demo pass 01

NAME := pass_01
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)
```

```
install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Confirm that the plugin is installed:

```
bap list plugins
```

Now run the pass over some binary executable, e.g.:

```
bap /bin/true --my-pass-01
```

When this pass runs, it returns an updated version of the project, which BAP an pass on to other passes.

Clean up:

```
make uninstall
make clean
```

Running multiple passes

You can build and install many passes, and tell BAP to run any of them together in a chain, one after the other.

For example, if you were to create another pass and give the plugin the public name `my-pass-02`, then after you install it, you can tell BAP to run `my-pass-00`, then `my-pass-01`, then `my-pass-02`, one after the other:

```
bap /bin/true --my-pass-00 --my-pass-01 --my-pass-02
```

That will first run `my-pass-00`, then `my-pass-01`, then `my-pass-02`. And, since `my-pass-01` returns an updated project, BAP will take the version of the project that `my-pass-01` produces, and feed it into `my-pass-02` as its input.

Note that if you switch the order of the arguments in the above command, BAP will execute the passes in the order you list them:

```
bap /bin/true --my-pass-01 --my-pass-00 --my-pass-02
```

Suppressing a pass

You can suppress a pass by prefixing `-no-` to it. For instance, to tell BAP *not* to run `my-pass-00`:

```
bap /bin/true --no-my-pass-00 --my-pass-01 --my-pass-02
```

C.2.7 Parameters

BAP can collect user-provided parameters from the command line or file, and you can retreive them from within an extension or pass. Parameters are declared using `Bap_main.Extension.Configuration`, and the types of parameters are defined in `Bap_main.Extension.Type`.

Declaring a parameter

In a new folder, create a file called `extension_02.ml` with the following:

```
open Core_kernel
open Bap.Std
```

For convenience, add the following aliases:

```
module Conf = Bap_main.Extension.Configuration
module Param_type = Bap_main.Extension.Type
```

Declare a parameter called `user`, which is just a string:

```
let user = Conf.parameter Param_type.string "user"
```

Now create an extension that uses `Conf.get` to retrieve the value of that parameter and print it:

```
let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  let user = Conf.get ctxt user in
  printf "Hello, '%s'\n%!" user;
  Ok ()
```

Finally, register the extension:

```
let () =
  Bap_main.Extension.declare run
```

The whole file looks like this:

```
open Core_kernel
open Bap.Std

module Conf = Bap_main.Extension.Configuration
module Param_type = Bap_main.Extension.Type

let user = Conf.parameter Param_type.string "user"

let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  let user = Conf.get ctxt user in
  printf "Hello, '%s'\n%!" user;
  Ok ()

let () =
  Bap_main.Extension.declare run
```

Create a `Makefile`:

```
PUBLIC_NAME := my-extension-02
PUBLIC_DESC := My demo extension 02

NAME := extension_02
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Run `bap`, e.g.:

```
bap /bin/true
```

It should print out:

```
Hello, ''
```

The user is empty. This is because we did not specify a value for the user parameter. The format for the command line argument is:

```
--<name-of-plugin>-<parameter-name>
```

In this case, the plugin is named `my-extension-02`, and the parameter name is `user`, so the command line argument is:

```
--my-extension-02-user
```

Run `bap` again, but this time, provide a value for that argument:

```
bap /bin/true --my-extension-02-user Alice
```

Now it prints out:

```
Hello, 'Alice'
```

Clean up:

```
make uninstall
make clean
```

Parameters for passes

Parameters can be retrieved from passes, if the context is sent to the pass. In a new folder, create a file called `pass_02.ml` with the following:

```
open Core_kernel
open Bap.Std

module Conf = Bap_main.Extension.Configuration
module Param_type = Bap_main.Extension.Type
```

Declare two parameters: `user` (a string), and `favorite-num` (an int):

```
let user = Conf.parameter Param_type.string "user"
let fav_num = Conf.parameter Param_type.int "favorite-num"
```

Create a pass that takes a BAP context and a project, and which retrieves the parameters from the context:

```
let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
  let user = Conf.get ctxt user in
  let fav_num = Conf.get ctxt fav_num in
  printf "Hello, '%s', your favorite number is '%d'\n%!" user fav_num
```

Now create an extension that registers the pass, curried with the context:

```
let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' (pass ctxt);
  Ok ()
```

Finally, declare the extension:

```
let () =
  Bap_main.Extension.declare run
```

The whole file looks like this:

```
open Core_kernel
open Bap.Std

module Conf = Bap_main.Extension.Configuration
module Param_type = Bap_main.Extension.Type

let user = Conf.parameter Param_type.string "user"
let fav_num = Conf.parameter Param_type.int "favorite-num"

let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
  let user = Conf.get ctxt user in
  let fav_num = Conf.get ctxt fav_num in
  printf "Hello, '%s', your favorite number is '%d'\n%!" user fav_num

let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' (pass ctxt);
  Ok ()

let () =
  Bap_main.Extension.declare run
```

Create a `Makefile`:

```
PUBLIC_NAME := my-pass-02
PUBLIC_DESC := My demo pass 02

NAME := pass_02
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)
```

```
install: build
        bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
        bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
        bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Run `bap`, and invoke the pass, e.g.:

```
bap /bin/true --my-pass-02
```

It should print out:

```
Hello, '', your favorite number is '0'
```

Note that `user` defaults to an empty string, and `favorite-num` defaults to `0`.

Run it again, but provide values for those parameters:

```
bap /bin/true --my-pass-02 --my-pass-02-user Alice --my-pass-02-favorite-num 7
```

It should print out:

```
Hello, 'Alice', your favorite number is '7'
```

Clean up:

```
make uninstall
make clean
```

Parameters via environment variables

You can specify parameters as environment variables. The format for the environment variable is:

```
BAP_<name-of-plugin>_<parameter-name>
```

For instance, take the `user` parameter for the `my-pass-02` plugin above. The environment variable for this would be:

```
BAP_MY_PASS_02_USER
```

If this environment variable is set, then BAP will use it when it runs the pass. For instance:

```
BAP_MY_PASS_02_USER=Bob bap /bin/true --my-pass-02
```

It should print out:

```
Hello, 'Bob', your favorite number is '0'
```

Parameters via configuration files

You can specify the values of parameters in configuration files. The format is:

```
<plugin-name>-<parameter-name>=<value>
```

For instance, if we take the `user` parameter for the `my-pass-02` plugin above, we could specify a value in a configuration file like this:

```
my-pass-02-user=Carol
```

BAP will look for configuration files in `$XDG_CONFIG_HOME/bap`. For instance, on Ubuntu, that would be `${HOME}/.config/bap`. Create that folder:

```
mkdir -p ~/.config/bap
```

Then create a file (it can be named anything) inside that folder, e.g.:

```
touch ~/.config/bap/conf
```

Inside that file, specify a value for the `user` and `favorite-num` parameters:

```
my-pass-02-user=Carol
my-pass-02-favorite-num=13
```

Now run the plugin again:

```
bap /bin/true --my-pass-02
```

It should print out:

```
Hello, 'Carol', your favorite number is '13'
```

Precedence

BAP evaluates parameters in the following order:

- Config file
- Environment variable
- Command line argument

C.2.8 Logging

In your plugins, you can send debug- and info-level messages to BAP's log stream, and they will be printed to the default BAP log, which is located at:

```
${XDG_STATE_HOME}/bap/log
```

E.g., on Ubuntu:

```
~/.local/state/bap/log
```

To see debug-level messages, you must run your plugin with the environment variable `BAP_DEBUG` set to `1`.

Example

In a new folder, create a file called `pass_03.ml` and instantiate the BAP log:

```
open Core_kernel
open Bap.Std

module L = Bap_main_event.Log.Create()
```

Add a pass that sends a message to the info channel and the debug channel:

```
let pass (proj : Project.t) : unit =
  L.info "My pass 03 is: %s" "running";
  L.debug "Debug: %s" "a debug message"
```

Create an extension that registers the pass:

```
let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' pass;
  Ok ()
```

And finally, declare the extension:

```
let () =
  Bap_main.Extension.declare run
```

So the whole file looks like this:

```
open Core_kernel
open Bap.Std

module L = Bap_main_event.Log.Create()

let pass (proj : Project.t) : unit =
  L.info "My pass 03 is: %s" "running";
  L.debug "Debug: %s" "a debug message"

let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
  Project.register_pass' pass;
  Ok ()

let () =
  Bap_main.Extension.declare run
```

Add a `Makefile`:

```
PUBLIC_NAME := my-pass-03
PUBLIC_DESC := My demo pass 03

NAME := pass_03
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
```

```
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Run the pass:

```
bap /bin/true --my-pass-03
```

Look at the last few lines of the BAP log:

```
tail -n3 ~/.local/state/bap/log
```

The last line should be:

```
my-pass-03.info> My pass 03 is: running
```

Notice that the debug message is not present. To see it, run `bap` while `BAP_DEBUG=1` is set:

```
BAP_DEBUG=1 bap /bin/true --my-pass-03
```

Tail the log again:

```
tail -n3 ~/.local/state/bap/log
```

The last two lines should be:

```
my-pass-03.info> My pass 03 is: running
my-pass-03.debug> Debug: a debug message
```

Clean up:

```
make uninstall clean
```

C.2.9 Custom commands

The BAP CLI knows about a number of subcommands. To see them all:

```
bap list commands
```

Each of the commands that it lists has its own further command line parameters, positional arguments, and options. E.g.:

```
bap mc --help
```

You can create your own custom command, and add it to BAP.

Example

In a new folder, create a file called `command_00.ml` with the following:

```
open Core_kernel
```

Add the following module aliases for convenience:

```
module Param_type = Bap_main.Extension.Type
module Cmd = Bap_main.Extension.Command
```

Next, create a `Cli` module, and specify a name and a doc string for the command:

```
module Cli = struct

  let name = "my-command-00"
  let doc = "A demo BAP command"

end
```

Next, create a `-job-title` parameter that takes a string:

```
  let job_title = Cmd.parameter Param_type.string "job-title"
    ~doc:"Your job title"
```

Create a positional argument:

```
  let first_name = Cmd.argument Param_type.string
    ~doc:"Your first name"
```

Now, specify the grammar of the command's arguments by listing them in order, separated by the `$` operator:

```
  let grammar = Cmd.(args $ job_title $ first_name)
```

Finally, create a callback that BAP can invoke when a user calls your command. The callback should take as arguments the CLI parameters, and a `Bap_main.ctxt`. It should return `Ok ()`, or a `Bap_main.error`. Here is an example:

```
  let callback (job_title : string) (first_name : string)
    (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) result =
    printf "First name: %s\n%!" first_name;
    printf "Job title: %s\n%!" job_title;
    Ok ()
```

Now, declare the command:

```
let () =
  Cmd.declare Cli.name Cli.grammar Cli.callback ~doc:Cli.doc
```

This tells BAP the name of the command, the grammar for the arguments, the callback it should invoke when a user calls the command, and the doc string.

The whole file looks like this:

```
open Core_kernel

module Param_type = Bap_main.Extension.Type
module Cmd = Bap_main.Extension.Command

module Cli = struct

  let name = "my-command-00"
  let doc = "A demo BAP command"

  let job_title = Cmd.parameter Param_type.string "job-title"
    ~doc:"Your job title"

  let first_name = Cmd.argument Param_type.string
    ~doc:"Your first name"

  let grammar = Cmd.(args $ job_title $ first_name)

  let callback (job_title : string) (first_name : string)
    (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) result =
    printf "First name: %s\n%!" first_name;
    printf "Job title: %s\n%!" job_title;
    Ok ()

end

let () =
  Cmd.declare Cli.name Cli.grammar Cli.callback ~doc:Cli.doc
```

Add a `Makefile`:

```
PUBLIC_NAME := my-command-00
PUBLIC_DESC := My demo command 00

NAME := command_00
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all
```

```
all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
        bapbuild -clean

uninstall:
        bapbundle remove $(PLUGIN)

build: $(SRC)
        bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
        bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
        bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
        bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Confirm that your command is installed:

```
bap list commands
```

You should see in the list:

```
  my-command-00            a demo BAP command
```

See the help:

```
bap my-command-00 --help
```

It will list the full help (which includes the parameters/arguments you specified, but also many other parameters that your command inherits from BAP).

Run the command:

```
bap my-command-00 "Jo" --job-title="Engineer"
```

It should print:

```
First name: Jo
Job title: Engineer
```

Clean up:

```
make uninstall clean
```

C.2.10 Custom command errors

A custom command returns `Ok ()` or a `Bap_main.error`. `Bap_main.error` is an extensible type, so you can add your own variants, and write your own error printer to handle your variants.

Example

In a new folder, create a file `command_01.ml`, with the following:

```
open Core_kernel

module Param_type = Bap_main.Extension.Type
module Cmd = Bap_main.Extension.Command
module Err = Bap_main.Extension.Error
```

Next, add a custom error constructor `Fail`:

```
type Err.t += Fail of string
```

Add a custom printer that returns `Some <string>` for `Fail <string>`:

```
let error_printer (e : Err.t) : string option =
  match e with
  | Fail s -> Some (sprintf "My custom command error: %s" s)
  | _ -> None
```

Next, add in your command CLI, e.g.:

```
module Cli = struct

  let name = "my-command-01"
  let doc = "Another demo BAP command"

  let job_title = Cmd.parameter Param_type.string "job-title"
    ~doc:"Your job title"

  let first_name = Cmd.argument Param_type.string
    ~doc:"Your first name"

  let grammar = Cmd.(args $ job_title $ first_name)

  let callback (job_title : string) (first_name : string)
    (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) result =
    match first_name with
    | "Jo" ->
      Error (Fail "Jo, you can't be here")
    | _ ->
      printf "First name: %s\n%!" first_name;
      printf "Job title: %s\n%!" job_title;
      Ok ()

end
```

Notice that if the first name is `Jo`, the callback returns a `Fail` error, otherwise it returns `Ok ()`.

Finally, register the custom error printer, and declare the command:

```
let () =
  Err.register_printer error_printer;
  Cmd.declare Cli.name Cli.grammar Cli.callback ~doc:Cli.doc
```

The whole file looks like this:

```
open Core_kernel

module Param_type = Bap_main.Extension.Type
module Cmd = Bap_main.Extension.Command
module Err = Bap_main.Extension.Error

type Err.t += Fail of string

let error_printer (e : Err.t) : string option =
  match e with
  | Fail s -> Some (sprintf "My custom command error: %s" s)
  | _ -> None

module Cli = struct

  let name = "my-command-01"
  let doc = "Another demo BAP command"

  let job_title = Cmd.parameter Param_type.string "job-title"
    ~doc:"Your job title"

  let first_name = Cmd.argument Param_type.string
    ~doc:"Your first name"

  let grammar = Cmd.(args $ job_title $ first_name)

  let callback (job_title : string) (first_name : string)
    (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) result =
    match first_name with
    | "Jo" ->
      Error (Fail "Jo, you can't be here")
    | _ ->
      printf "First name: %s\n%!" first_name;
      printf "Job title: %s\n%!" job_title;
      Ok ()

end

let () =
  Err.register_printer error_printer;
  Cmd.declare Cli.name Cli.grammar Cli.callback ~doc:Cli.doc
```

Add a `Makefile`:

```
PUBLIC_NAME := my-command-01
PUBLIC_DESC := My demo command 01

NAME := command_01
```

```
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


##################################################
# DEFAULT
##################################################

.DEFAULT_GOAL := all

all: clean uninstall install


##################################################
# THE PLUGIN
##################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install:

```
make
```

Confirm that your command got installed:

```
bap list commands
```

You should see it listed:

```
  my-command-01            another demo BAP command
```

Run the command:

```
bap my-command-01 "Tom" --job-title "Sales associate"
```

It should print out:

```
First name: Tom
Job title: Sales associate
```

Now run the command, but with the name "Jo":

```
bap my-command-01 "Jo" --job-title "Engineer"
```

That triggers the error:

```
My custom command error: Jo, you can't be here
```

Clean up:

```
make uninstall clean
```

C.3 The Knowledge Base

C.3.1 The Knowledge Base

The knowledge base (or "KB" for short) is a kind of database that your plugins can use to store information.

Objects

What do we actually store in the KB? We store "objects."

An object is an entity that has a series of slots that you can put information in. If you like, you can think of an object as a storage cabinet with a bunch of slots.

At first, the slots are empty, but your plugin can add information to (and read information from) the slots as needed.

Slots

Slots are "typed," in the sense that each slot can only hold data that comes from a specified domain of values.

Slots take information in an additive, non-destructive way. If you try to put some information into a slot that conflicts with information already there, BAP will reject it. Similarly, if you try to remove or overwrite information that's already there, BAP will prevent it.

In BAP's documentation, slots are often called "slots," but they are also called "properties" (of objects). Domains are always called "domains."

Snapshots

At any point during your plugin's lifetime, you can take a snapshot of an object. This is just a record of the information contained in each of the object's slots, at the time that the snapshot is taken.

In BAP's documentation, a snapshot is called a KB "value," but keep in mind that a KB-value is not a single piece of information. Rather, it is an array of values, taken from the slots when the snapshot was taken.

Classes

Before you create an object, you must first create a blueprint for it. In the blueprint, you specify which slots the object should have, and which domain of values each slot should hold.

Once you've created a blueprint, you can create many objects (instances) from the same blueprint. All instances of the blueprint will therefore have the same array of slots.

The blueprint is called a KB "class" in BAP's documentation. Hence, to say that each object is created from a blueprint is to say that each object is an instance of a class.

Documentation

For more about the KB and its API, see the documentation.

C.3.2 Creating KB classes

KB classes

Every KB class has the following type:

```
('k, 's) KB.cls
```

Note that: * 'k is a custom type that serves as a unique tag for the class. * 's is the type of a sorting index that can be used to index sub-classes.

A mono-sorted example

In a new folder somewhere, create a file called `main.ml`. For convenience, add an alias to the KB module:

```
module KB = Bap_knowledge.Knowledge
```

Then initialize BAP:

```
let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

For a toy example, let's define a class to represent cars.

First, let's encapsulate our definition inside a module:

```
module Car = struct

  (* We'll define the class here *)

end
```

Next, specify a package name (BAP uses this to namespace your classes, so choose a name that uniquely identifies your organizaton):

```
module Car = struct

  let package = "my.org"

end
```

Then specify a type to uniquely tag the class:

```
module Car = struct

  package = "my.org"
  type tag = Car

end
```

Next, we need to specify a type for a sorting index, which can be used to index sub-classes. For this example, let's assume that we don't need further sub-classes. So, we'll just use `unit` as the sort, since it has only one value (namely, `()`):

```
module Car = struct

  package = "my.org"
  type tag = Car
  type sort = unit

end
```

Now declare a class for cars:

```
module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = () (* The only value of unit, so the only possible index *)
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

end
```

We haven't attached any slots to this class yet, but this is a valid class nonethless, and it illustrates the basic procedure behind creating a class: specify a type to use to tag the class and a type to use as a sorting index, and then use `KB.Class.declare name index` to create the class.

At this point, there isn't much we can do with the class, so let's just print its name.

Add a function that uses `KB.Class.name` to extract the name of a class:

```
let get_name (cls : ('k, 's) KB.cls) : string =
  let kb_name = KB.Class.name cls in
  KB.Name.show kb_name
```

Then print the name:

```
let () =
  let name = get_name Car.cls in
  Format.printf "Car class name: %s\n%!" name
```

To summarize, the entire `main.ml` file looks like this:

```
module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

end
```

```
let get_name (cls : ('k, 's) KB.cls) : string =
  let kb_name = KB.Class.name cls in
  KB.Name.show kb_name

let () =
  let name = get_name Car.cls in
  Format.printf "Car class name: %s\n%!" name
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the name of the class:

```
Car class name: my.org:car
```

Clean up:

```
make clean
```

C.3.3 Multisorted KB classes

KB classes

Recall that every KB class has the following type:

```
('k, 's) KB.cls
```

Note that: * `'k` is a custom type that serves as a unique tag for the class. * `'s` is the type of a sorting index that can be used to index sub-classes.

The `'s` parameter can be used to distinguish different sub-classes.

Many-sorted example

In a new folder somewhere, create a file called `main.ml`. Add a `KB` alias and initialize BAP:

```
module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

For a many-sorted example, let's define a class to represent employees.

Fist, create a module to encapsulate the definition, and specify a package name and a tag for the class:

```
module Employee = struct

  package = "my.org"
  type tag = Employee

end
```

Now for a sorting index. Let's index our sub-classes by whether the employee is a member of the sales team, marketing team, or executive team:

```
module Employee = struct

  package = "my.org"
  type tag = Employee
  type sort = Sales | Marketing | Executive

end
```

Declare a class for employees on the sales team:

```
module Employee = struct

  package = "my.org"
  type tag = Employee
  type sort = Sales | Marketing | Executive

  let name = "sales-employee"
  let desc = "A class representing employees on the sales team"
```

```
  let index = Sales
  let sales_empl : (tag, sort) KB.cls =
    KB.Class.declare name index ~package ~desc

end
```

Declare another class, this time for employees on the executive team:

```
module Employee = struct

  package = "my.org"
  type tag = Employee
  type sort = Sales | Marketing | Executive

  let name = "sales-employee"
  let desc = "A class representing employees on the sales team"
  let index = Sales
  let sales_cls : (tag, sort) KB.cls =
    KB.Class.declare name index ~package ~desc

  let name = "executive-employee"
  let desc = "A class representing employees on the executive team"
  let index = Executive
  let executive_cls : (tag, sort) KB.cls =
    KB.Class.declare name index ~package ~desc

end
```

We haven't attached any slots to these classes, so let's just display the names of these classes:

```
let get_name (cls : ('k, 's) KB.cls) : string =
  let kb_name = KB.Class.name cls in
  KB.Name.show kb_name

let () =
  let sales_class_name = get_name Employee.sales_cls in
  let executive_class_name = get_name Employee.executive_cls in
  Format.printf "Sales class name: %s\n%!" sales_class_name;
  Format.printf "Executive class name: %s\n%!" executive_class_name
```

To summarize, the entire `main.ml` file looks like this:

```
module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Employee = struct

  package = "my.org"
  type tag = Employee
  type sort = Sales | Marketing | Executive
```

```
    let name = "sales-employee"
    let desc = "A class representing employees on the sales team"
    let index = Sales
    let sales_cls : (tag, sort) KB.cls =
      KB.Class.declare name index ~package ~desc

    let name = "executive-employee"
    let desc = "A class representing employees on the executive team"
    let index = Executive
    let executive_cls : (tag, sort) KB.cls =
      KB.Class.declare name index ~package ~desc

  end

  let get_name (cls : ('k, 's) KB.cls) : string =
    let kb_name = KB.Class.name cls in
    KB.Name.show kb_name

  let () =
    let sales_class_name = get_name Employee.sales_cls in
    let executive_class_name = get_name Employee.executive_cls in
    Format.printf "Sales class name: %s\n%!" sales_class_name;
    Format.printf "Executive class name: %s\n%!" executive_class_name
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)
```

```
run: build
	dune exec ./$(EXE)
```

Now run the program:

```
make
```

It will print out the names of the classes:

```
Sales class name: my.org:sales-employee
Executive class name: my.org:executive-employee
```

Clean up:

```
make clean
```

C.3.4 KB domains

Every KB slot has a domain, which is a specified range of values that it can hold.

Domains are special, because the values in a domain are ordered by their informativeness.

There is always a bottom element, which represents "we have no information about this." Then, all of the other values stand above that (as a flat array, or as an ascending chain, or as a lattice, and so on).

BAP allows us to add a value to a slot only if the value we are putting in is more informative than what was in the slot before.

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Define a flat domain that can hold strings, where the empty element is the empty string:

```
let string_domain : string KB.Domain.t =
  KB.Domain.flat "string-domain"
    ~inspect:(fun s -> Sexp.Atom s)
    ~equal:String.(=)
    ~empty:""
```

Note that we provided an `inspect` function, which converts the value into an S-expression. This lets BAP print values in the domain nicely.

We aren't doing anything with this domain yet, so for now let's just display its name:

```
let () =
  let name = KB.Domain.name string_domain in
  Format.printf "Domain name: %s\n%!" name
```

Create a domain that takes optional boolean values:

```
let optional_bool_domain : bool option KB.Domain.t =
  KB.Domain.optional "optional-bool-domain"
    ~inspect:(fun b -> Sexp.Atom (Bool.to_string b))
    ~equal:Bool.(=)
```

Print the domain's name:

```
let () =
  let name = KB.Domain.name optional_bool_domain in
  Format.printf "Domain name: %s\n%!" name
```

To summarize, here is the entire `main.ml` file:

```
open Core_kernel

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let string_domain : string KB.Domain.t =
  KB.Domain.flat "string-domain"
    ~inspect:(fun s -> Sexp.Atom s)
    ~equal:String.(=)
    ~empty:""

let () =
  let name = KB.Domain.name string_domain in
  Format.printf "Domain name: %s\n%!" name

let optional_bool_domain : bool option KB.Domain.t =
  KB.Domain.optional "optional-bool-domain"
    ~inspect:(fun b -> Sexp.Atom (Bool.to_string b))
    ~equal:Bool.(=)

let () =
  let name = KB.Domain.name optional_bool_domain in
  Format.printf "Domain name: %s\n%!" name
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
```

```
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONEY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the names of the two domains:

```
Domain name: string-domain
Domain name: optional-bool-domain
```

Clean up:

```
make clean
```

Documentation

For more details about the different kinds of domains, see the documentation.

C.3.5 KB Slots

Every KB slot has the following type:

```
('k, 'p) KB.slot
```

Note that: * `'k` is the tag that uniquely identifies a class, i.e., the `'k` in the class's type `('k, 's) Kb.cls`. * `'p` is the type of data that goes in the slot, e.g. `string`, `int option`, etc.

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a class to represent cars:

```
module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

end
```

Add a string domain:

```
module Car = struct

  ...

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
      ~equal:String.(=)
      ~empty:""

end
```

Declare a slot to hold the color:

```
module Car = struct

  ...

  let color : (tag, string) KB.slot =
    KB.Class.property cls "color" string_domain ~package

end
```

At this point, we have created a car class, whose objects will have one slot that can be filled with the name of a color (a string).

We aren't doing anything with this class yet, so let's just print out the name of the class.

```
let get_name (cls : ('k, 's) KB.cls) : string =
  let kb_name = KB.Class.name cls in
  KB.Name.show kb_name

let () =
  let name = get_name Car.cls in
  Format.printf "Car class name: %s\n%!" name
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
      ~equal:String.(=)
      ~empty:""

  let color : (tag, string) KB.slot =
    KB.Class.property cls "color" string_domain ~package

end

let get_name (cls : ('k, 's) KB.cls) : string =
  let kb_name = KB.Class.name cls in
  KB.Name.show kb_name

let () =
  let name = get_name Car.cls in
  Format.printf "Car class name: %s\n%!" name
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################
```

```
.DEFAULT_GOAL := all

all: clean run


####################################################
# THE EXE
####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the name of the class:

```
Car class name: my.org:car
```

Clean up:

```
make clean
```

C.3.6 Creating and updating KB objects

KB objects are created with `KB.Object.create`. Data can be put in their slots with `KB.provide`, and data can be retrieved from the slots with `KB.collect`.

Example

In a new folder somewhere, create a file called `main.ml` that has a class to represent cars:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit
```

```
    let name = "car"
    let desc = "A class representing cars"
    let index = ()
    let cls : (tag, sort) Kb.cls =
      KB.Class.declare name index ~package ~desc

    let string_domain : string KB.Domain.t =
      KB.Class.flat "string-domain"
        ~inspect:(fun s -> Sexp.Atom s)
        ~equal:String.(=)
        ~empty:""

    let color : (tag, string) KB.slot =
      KB.Class.property cls "color" string_domain ~package

  end
```

If we want to create and manipulate objects in the KB, we need to run a KB computation.

One simple way to do this is by using `Bap.Std.Toplevel.exec`:

```
  let () =
    Toplevel.exec
      begin

        (* Create and manipulate KB objects...*)

        KB.return ()
      end
```

First, create an instance of the car class:

```
  let () =
    Toplevel.exec
      begin
        let* car = KB.Object.create Car.cls in

        KB.return ()
      end
```

Get a string representation of the object and print it:

```
  let () =
    Toplevel.exec
      begin
        let* car = KB.Object.create Car.cls in
        let* repr = KB.Object.repr Car.cls in
        Format.printf "- Car: %s\n%!" repr;

        KB.return ()
      end
```

Put the color "red" in the `Car.color` slot:

```
let () =
  Toplevel.exec
    begin
      let* car = KB.Object.create Car.cls in
      let* repr = KB.Object.repr Car.cls in
      Format.printf "- Car: %s\n%!" repr;

      let* () = KB.provide Car.color car "red" in

      KB.return ()
    end
```

Collect the value in the slot and print it:

```
let () =
  Toplevel.exec
    begin
      let* car = KB.Object.create Car.cls in
      let* repr = KB.Object.repr Car.cls in
      Format.printf "- Car: %s\n%!" repr;

      let* () = KB.provide Car.color car "red" in
      let* color = KB.collect Car.color car in
      Format.printf "- Color: %s\n%!" color;

      KB.return ()
    end
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
```

```
        ~equal:String.(=)
        ~empty:""

    let color : (tag, string) KB.slot =
      KB.Class.property cls "color" string_domain ~package

  end

  let () =
    Toplevel.exec
      begin
        let* car = KB.Object.create Car.cls in
        let* repr = KB.Object.repr Car.cls in
        Format.printf "- Car: %s\n%!" repr;

        let* () = KB.provide Car.color car "red" in
        let* color = KB.collect Car.color car in
        Format.printf "- Color: %s\n%!" color;

        KB.return ()
      end
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out something that looks like this:

```
- Car: #<my.org:car <0x2>>
- Color: red
```

Clean up:

```
make clean
```

C.3.7 Promises

You can use `KB.provide` to fill a slot with data. If you want to defer providing that data until the slot is actually accessed, you can instead use `KB.promise` to register a function that will compute the value at call time.

Example

In a new folder somewhere, create a file called `main.ml` that has a class to represent cars:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
      ~equal:String.(=)
      ~empty:""

  let color : (tag, string) KB.slot =
    KB.Class.property cls "color" string_domain ~package

end
```

Define a function that takes an object of the car class, and provides a color:

```
let provide_color (_ : Car.tag KB.obj) : string KB.t =
  Kb.return "red"
```

Now, use `KB.promise` to register that function:

```
let () =
  KB.promise Car.color provide_color;
```

Now, create a car object, and retrieve its color:

```
let () =
  KB.promise Car.color provide_color;
  Toplevel.exec
    begin
      let* car = KB.Object.create Car.cls in
      let* color = KB.collect Car.color car in
      Format.printf "- Color: %s\n%!" color;
      KB.return ()
    end
```

When the color is collected, the `provide_color` function is triggered, which in turn returns the color.

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
      ~equal:String.(=)
      ~empty:""

  let color : (tag, string) KB.slot =
```

```
    KB.Class.property cls "color" string_domain ~package

end

let provide_color (_ : Car.tag KB.obj) : string KB.t =
  Kb.return "red"

let () =
  KB.promise Car.color provide_color;
  Toplevel.exec
    begin
      let* car = KB.Object.create Car.cls in
      let* color = KB.collect Car.color car in
      Format.printf "- Color: %s\n%!" color;
      KB.return ()
    end
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print the color:

```
- Color: red
```

Clean up:

```
make clean
```

C.3.8 Snapshots (KB values)

You can take a snapshot of an object. A snapshot is just a record of the values contained in the object's slots at the time the snapshot is taken.

In BAP's documentation, a snapshot is called a "value," but bear in mind that a KB-value is not just a single value. It is an array of values, taken from the slots of the object at the time the snapshot was taken.

Example

In a new folder somewhere, create a file called `main.ml` that has a class to represent cars:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
  let index = ()
  let cls : (tag, sort) Kb.cls =
    KB.Class.declare name index ~package ~desc

  let string_domain : string KB.Domain.t =
    KB.Class.flat "string-domain"
      ~inspect:(fun s -> Sexp.Atom s)
      ~equal:String.(=)
      ~empty:""

  let color : (tag, string) KB.slot =
    KB.Class.property cls "color" string_domain ~package

end
```

Add a function that creates a new car object and gives it a color:

```
let build_car : Car.tag KB.obj KB.t =
  let* car = KB.Object.create Car.cls in
  let* () = KB.provide Car.color car "red" in
  KB.return car
```

Next, use `KB.run` to execute the `build_car` procedure:

```
let () =
  let state = Toplevel.current () in
  let result = KB.run Car.cls build_car state in
```

If the KB computation succeeds, it returns a pair `(snapshot, new_state)`, where `snapshot` is a snapshot of the object at the end of the computation, and `new_state` is the updated KB state. If the computation fails, it returns a KB error. We can print both of them:

```
let () =
  let state = Toplevel.current () in
  let result = KB.run Car.cls build_car state in
  match result with
  | Ok (snapshot, _) -> Format.printf "- Snapshot: %a\n%!" KB.Value.pp snapshot
  | Error e -> Format.eprintf "KB problem: %a\n%!" KB.Conflict.pp e
```

We can pull out the data from the `color` slot, and print that too:

```
let () =
  let state = Toplevel.current () in
  let result = KB.run Car.cls build_car state in
  match result with
  | Ok (snapshot, _) ->
    begin
      Format.printf "- Snapshot: %a\n%!" KB.Value.pp snapshot;
      let color = KB.Value.get Car.color snapshot in
      Format.printf "- Color: %s\n%!"
    end
  | Error e -> Format.eprintf "KB problem: %a\n%!" KB.Conflict.pp e
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit
```

```
    let name = "car"
    let desc = "A class representing cars"
    let index = ()
    let cls : (tag, sort) Kb.cls =
      KB.Class.declare name index ~package ~desc

    let string_domain : string KB.Domain.t =
      KB.Class.flat "string-domain"
        ~inspect:(fun s -> Sexp.Atom s)
        ~equal:String.(=)
        ~empty:""

    let color : (tag, string) KB.slot =
      KB.Class.property cls "color" string_domain ~package

  end

  let build_car : Car.tag KB.obj KB.t =
    let* car = KB.Object.create Car.cls in
    let* () = KB.provide Car.color car "red" in
    KB.return car

  let () =
    let state = Bap.Std.Toplevel.current () in
    let result = KB.run Car.cls build_car state in
    match result with
    | Ok (snapshot, _) ->
      begin
        Format.printf "- Snapshot: %a\n%!" KB.Value.pp snapshot;
        let color = KB.Value.get Car.color snapshot in
        Format.printf "- Color: %s\n%!" color
    | Error e -> Format.eprintf "KB Problem: %a\n%!" KB.Conflict.pp e
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
```

```
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot and the color:

```
- Snapshot: ((my.org:color red))
- Color: red
```

Clean up:

```
make clean
```

Documentation

For more on snapshots (i.e., KB "values"), see the documentation.

C.3.9 Using Toplevel.eval

`Toplevel.eval` can be used to get the data held in a particular slot of a particular object.

Example

In a new folder somewhere, create a file called `main.ml` that has a class to represent cars:

```
open Core_kernel
open Bap.Std

module KB = Bap_knowledge.Knowledge
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

module Car = struct

  let package = "my.org"
  type tag = Car
  type sort = unit

  let name = "car"
  let desc = "A class representing cars"
```

```
    let index = ()
    let cls : (tag, sort) Kb.cls =
      KB.Class.declare name index ~package ~desc

    let string_domain : string KB.Domain.t =
      KB.Class.flat "string-domain"
        ~inspect:(fun s -> Sexp.Atom s)
        ~equal:String.(=)
        ~empty:""

    let color : (tag, string) KB.slot =
      KB.Class.property cls "color" string_domain ~package

  end
```

Add a function that creates a new car object and gives it a color:

```
  let provide_color : Car.tag KB.obj KB.t =
    let* car = KB.Object.create Car.cls in
    let* () = KB.provide Car.color car "red" in
    KB.return car
```

Next, use `Toplevel.eval` to extract the color:

```
  let () =
    let color = Toplevel.eval Car.color provide_color in
    Format.printf "- Color: %s\n%!" color
```

To summarize, the entire `main.ml` file looks like this:

```
  open Core_kernel
  open Bap.Std

  module KB = Bap_knowledge.Knowledge
  open KB.Let

  let () = match Bap_main.init () with
    | Ok () -> ()
    | Error _ -> failwith "Error initializing BAP"

  module Car = struct

    let package = "my.org"
    type tag = Car
    type sort = unit

    let name = "car"
    let desc = "A class representing cars"
    let index = ()
    let cls : (tag, sort) Kb.cls =
      KB.Class.declare name index ~package ~desc

    let string_domain : string KB.Domain.t =
      KB.Class.flat "string-domain"
```

```
          ~inspect:(fun s -> Sexp.Atom s)
          ~equal:String.(=)
          ~empty:""

      let color : (tag, string) KB.slot =
        KB.Class.property cls "color" string_domain ~package

  end

  let provide_color : Car.tag KB.obj KB.t =
    let* car = KB.Object.create Car.cls in
    let* () = KB.provide Car.color car "red" in
    KB.return car

  let () =
    let color = Toplevel.eval Car.color provide_color in
    Format.printf "- Color: %s\n%!" color
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print the color:

```
- Color: blue
```

Clean up:

```
make clean
```

Documentation

For more on Toplevel eval, see the documentation.

C.4 Compilation Units

C.4.1 Compilation units

The KB lets you create your own slots, objects, and classes, which you can use to store whatever information you like.

BAP also has many pre-defined slots, objects, and classes. They are arranged in a hierarchy. Some of the slots hold objects, which themselves have slots, which can in turn hold further objects, and so on.

At the top of the hierarchy is an object called a "label." A label basically represents a location in a binary program.

A label has various slots, e.g.:

- An address slot (to hold the address at that point in the program)
- A name slot (if there is a symbol name associated with this location)
- A memory slot (to hold the memory layout of the program at this point)

But a label also has slots that can hold further KB objects:

- A "unit" slot (to hold a compilation unit object)
- A "semantics" slot (to hold a semantics object)

Note that these slots hold KB *objects*, and so those objects themselves have further slots.

Here is a picture of a label object:

```
                      +--------------+    +----------------------+
                   +--| Address slot |--> | Can hold an address |
                   |  +--------------+    +----------------------+
                   |
                   |  +-----------+    +-----------------+
                   +--| Name slot |--> | Can hold a name |
                   |  +-----------+    +-----------------+
                   |
                   |  +-------------+    +------------------+
                   |--| Memory slot |--> | Can hold the mem |
  +-----------+    |  +-------------+    +------------------+
* | Label obj |--+
  +-----------+    |  +-----------------------+    +---------------------------+
                   |--| Compilation unit slot |--> | Can hold a Comp. unit obj |
                   |  +-----------------------+    +---------------------------+
```

```
                    |
                    |  +----------------+    +--------------------------+
                    +--| Semantics slot |--> | Can hold a Semantics obj |
                    |  +----------------+    +--------------------------+
                    |
                    +-- ... Various other slots (see documentation)
```

Here is a picture of a Compilation unit object:

```
                         +-------------------+    +-----------------+
                      +--| Target arch slot |--> | Holds Target.t |
                      |  +-------------------+    +-----------------+
  +----------------+  |
* | Comp. unit obj |--+
  +----------------+  |
                      |
                      +-- ... Various other slots (see documentation)
```

C.4.2 KB Labels

A "label" represents a location in a program. It represents the program that begins at that particular point in the executable.

In the KB, labels are objects that belong to the `Program` class. In other words, a label is the object BAP uses to represent a program.

Hence, these types are aliases:

```
Theory.Label.t
Theory.program KB.obj
```

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a procedure that creates a program object (a label):

```
let create_program : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  KB.return label
```

This will create an empty program, whose slots are empty and ready to be filled by further information. At this point, let's just use `KB.run` to execute this procedure and inspect the snapshot it returns:

```
let () =
  let state = Toplevel.current () in
  let result = KB.run T.Program.cls create_program state in
  match result with
  | Ok (snapshot, _) -> Format.printf "Program: %a\n%!" KB.Value.pp snapshot
  | Error e -> Format.eprintf "KB error: %a\n%!" KB.Conflict.pp e
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_program : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  KB.return label

let () =
  let state = Toplevel.current () in
  let result = KB.run T.Program.cls create_program state in
  match result with
  | Ok (snapshot, _) -> Format.printf "Program: %a\n%!" KB.Value.pp snapshot
  | Error e -> Format.eprintf "Error: %a\n%!" KB.Conflict.pp e
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
```

```
########################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot (just an empty program):

```
Program: ()
```

Clean up:

```
make clean
```

C.4.3 More about KB labels

A label has a variety of slots that can hold information. For instance, there are slots for:

- An address
- A name (e.g., a symbol name)
- Etc

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a procedure that creates a program object (a label) and then adds an address in the `addr` slot:

```
let create_program (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label
```

Add a function that prints an address:

```
let inspect_address (addr : Bitvec.t option) : unit =
  match addr with
  | Some addr -> Format.printf "Address: %a\n%!" Bitvec.pp addr
  | None -> Format.printf "Address: None\n%!"
```

Now create an address and a program:

```
let () =
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr in
```

Then use `KB.run` to evaluate the program, and inspect the address:

```
let () =
  ...
  let state = Toplevel.current () in
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) ->
    begin
      let addr = KB.Value.get T.Label.addr snapshot in
      inspect_address addr
    end
  | Error e -> Format.eprintf "Error: %a\n%!" KB.Conflict.pp e
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_program (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let inspect_address (addr : Bitvec.t option) : unit =
  match addr with
  | Some addr -> Format.printf "Address: %a\n%!" Bitvec.pp addr
  | None -> Format.printf "Address: None\n%!"

let () =
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr in

  let state = Toplevel.current () in
```

```
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) ->
    begin
      let addr = KB.Value.get T.Label.addr snapshot in
      inspect_address addr
    end
  | Error e -> Format.eprintf "Error: %a\n%!" KB.Conflict.pp e
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the the address:

```
Address: 0x10400
```

Clean up:

```
make clean
```

Documentation

For more information about other slots associated with labels, see the documentation.

C.4.4 Compilation units

Every program label has a slot to hold a compilation unit object.

A compilation unit object is a KB object, with its own set of slots. Those slots are:

```
* A target slot (to hold information about the target architecture)
* A source slot (for hold information about the source the program was compiled from)
* A compiler slot (for hold information about the compiler used to compile the program)
```

Note that the BAP documentation calls a compilation unit a "code unit" or just a "unit."

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a function that creates a program label with an address:

```
let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label
```

Add a function that creates a compilation unit with a target architecture:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Add a function that creates a program for a given address and target:

```
let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Note that the above function creates a compilation unit object (using `create_compilation_unit`), and then it inserts that object into the `T.Label.unit` slot. So here we have an object that lives in the slot of another object (it's a hiearchy).

Now use `KB.run` to evaluate the program and print the resulting snapshot:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) ->
    Format.printf "Program: %a\n%!" KB.Value.pp snapshot;
  | Error e -> Format.eprintf "Error: %a\n%!" KB.Conflict.pp e
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label

let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) ->
    Format.printf "Program: %a\n%!" KB.Value.pp snapshot;
  | Error e -> Format.eprintf "Error: %a\n%!" KB.Conflict.pp e
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the the snapshot:

```
Program: ((bap:arch armv7)
          (core:label-addr (0x10400))
          (core:label-unit (2))
          (core:encoding bap:llvm-armv7))
```

Clean up:

```
make clean
```

Documentation

For more information about other slots associated with units, see the documentation.

C.4.5 Extracting an object from inside another object

A compilation unit is a KB object (with its own set of slots), and it lives in the slot of a label, which is a parent object.

If we take a snapshot of the parent object (the label), and then we look in the compilation unit slot, we will find a compilation unit object.

If we want to inspect that object, we need to take a snapshot of it (so taking a snapshot of a parent object does not take a snapshot of children objects).

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add functions to create a label, a compilation unit, and a program:

```
let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Add a function that takes a `T.Unit.t option` and then uses `KB.run` to take a snapshot:

```
let inspect_comp_unit (comp_unit : T.Unit.t option) (state : KB.state) : unit =
  match comp_unit with
  | Some comp_unit' ->
    begin
      let result = KB.run T.Unit.cls (KB.return comp_unit') state in
      match result with
      | Ok (snapshot, _) ->
        let target = KB.Value.get T.Unit.target snapshot in
```

```
          Format.printf "- Target: %a\n%!" T.Target.pp target
        | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
      end
    | None -> Format.printf "No compilation unit\n%!"
```

Add a function that takes a program label and then uses `KB.run` to take a snapshot:

```
let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, state') ->
    begin
      let comp_unit = KB.Value.get T.Label.unit snapshot in
      inspect_comp_unit comp_unit state'
    end
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
```

Note that the above retrieves the compilation unit object from the `T.Label.unit` slot in the snapshot, and then it applies `inspect_comp_unit` to it.

Finally, create a program, and then inspect it:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  inspect_program program state
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
```

```
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in

  KB.return label

let inspect_comp_unit (comp_unit : T.Unit.t option) (state : KB.state) : unit =
  match comp_unit with
  | Some comp_unit' ->
    begin
      let result = KB.run T.Unit.cls (KB.return comp_unit') state in
      match result with
      | Ok (snapshot, _) ->
        let target = KB.Value.get T.Unit.target snapshot in
        Format.printf "- Target: %a\n%!" T.Target.pp target
      | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
    end
  | None -> Format.printf "No compilation unit\n%!"

let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, state') ->
    begin
      let comp_unit = KB.Value.get T.Label.unit snapshot in
      inspect_comp_unit comp_unit state'
    end
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e

let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  inspect_program program state
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all
```

```
all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
        dune clean

build:
        dune build ./$(EXE)

run: build
        dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the target:

```
- Target: bap:armv7+le
```

Clean up:

```
make clean
```

C.4.6 Targets

A compilation unit object has a slot that can hold a target architecture. Once you retrieve a target, you can retrieve various features:

- The endianness
- The word size (in bits)
- The size of addresses
- The size of instructions
- The size of instruction alignment
- Which register is the stack pointer
- A list of registers
- Etc

Retrieving targets

To find out which targets your local copy of BAP supports, use `bap list`:

```
bap list targets
```

That will provide a list of target names. To programmatically retrieve a target by its name, use `Theory.Target.read`, e.g.:

```
let target = Theory.Target.read "bap:armv7+le"
```

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add functions to create a label, a compilation unit, and a program:

```
let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Add a function that takes a `T.Target.t` and then prints some of its features:

```
let inspect_target (target : T.Target.t) : unit =
  let endianness = T.Target.endianness target in
  let wordsize = T.Target.bits target in
  let memory_address_size = T.Target.data_addr_size target in
  let pc_size = T.Target.code_addr_size target in
  let insn_alignment_size = T.Target.code_alignment target in
  let mem_var = T.Target.data target in
  let sp = T.Target.reg target T.Role.Register.stack_pointer in
  let sp_to_string sp =
    match sp with
    | Some reg -> T.Var.name reg
    | None -> "unknown"
  in
  let regs = T.Target.regs target in
  let regs_to_string regs =
    let regs_list = List.map (Set.to_list regs) ~f:T.Var.name in
    String.concat regs_list ~sep:", "
  in
  Format.printf "- Endianness: %a\n%!" T.Endianness.pp endianness;
```

```
  Format.printf "- Wordsize: %d\n%!" wordsize;
  Format.printf "- Memory address size: %d\n%!" memory_address_size;
  Format.printf "- Instruction size (program counter size): %d\n%!" pc_size;
  Format.printf "- Instruction alignment: %d\n%!" insn_alignment_size;
  Format.printf "- Memory variable (its name): %s\n%!" (T.Var.name mem_var);
  Format.printf "- SP: %s\n%!" (sp_to_string sp);
  Format.printf "- Registers: %s\n%!" (regs_to_string regs)
```

Add a function that takes a `T.Unit.t option` and then uses `KB.run` to take a snapshot and inspect the target:

```
let inspect_comp_unit (comp_unit : T.Unit.t option) (state : KB.state) : unit =
  match comp_unit with
  | Some comp_unit' ->
    begin
      let result = KB.run T.Unit.cls (KB.return comp_unit') state in
      match result with
      | Ok (snapshot, _) ->
        let target = KB.Value.get T.Unit.target snapshot in
        Format.printf "- Target: %a\n%!" T.Target.pp target;
        inspect_target target
      | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
    end
  | None -> Format.printf "No compilation unit\n%!"
```

Add a function that takes a program label and then uses `KB.run` to take a snapshot:

```
let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, state') ->
    begin
      let comp_unit = KB.Value.get T.Label.unit snapshot in
      inspect_comp_unit comp_unit state'
    end
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
```

Finally, create a program, and then inspect it:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  inspect_program program state
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB
```

```
open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_label (addr : Bitvec.t) : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let* () = KB.provide T.Label.addr label (Some addr) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = create_label addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label

let inspect_target (target : T.Target.t) : unit =
  let endianness = T.Target.endianness target in
  let wordsize = T.Target.bits target in
  let memory_address_size = T.Target.data_addr_size target in
  let pc_size = T.Target.code_addr_size target in
  let insn_alignment_size = T.Target.code_alignment target in
  let mem_var = T.Target.data target in
  let sp = T.Target.reg target T.Role.Register.stack_pointer in
  let sp_to_string sp =
    match sp with
    | Some reg -> T.Var.name reg
    | None -> "unknown"
  in
  let regs = T.Target.regs target in
  let regs_to_string regs =
    let regs_list = List.map (Set.to_list regs) ~f:T.Var.name in
    String.concat regs_list ~sep:", "
  in
  Format.printf "- Endianness: %a\n%!" T.Endianness.pp endianness;
  Format.printf "- Wordsize: %d\n%!" wordsize;
  Format.printf "- Memory address size: %d\n%!" memory_address_size;
  Format.printf "- Instruction size (program counter size): %d\n%!" pc_size;
  Format.printf "- Instruction alignment: %d\n%!" insn_alignment_size;
  Format.printf "- Memory variable (its name): %s\n%!" (T.Var.name mem_var);
  Format.printf "- SP: %s\n%!" (sp_to_string sp);
  Format.printf "- Registers: %s\n%!" (regs_to_string regs)

let inspect_comp_unit (comp_unit : T.Unit.t option) (state : KB.state) : unit =
  match comp_unit with
  | Some comp_unit' ->
    begin
      let result = KB.run T.Unit.cls (KB.return comp_unit') state in
```

```
        match result with
        | Ok (snapshot, _) ->
          let target = KB.Value.get T.Unit.target snapshot in
          Format.printf "- Target: %a\n%!" T.Target.pp target;
          inspect_target target
        | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
      end
    | None -> Format.printf "No compilation unit\n%!"

let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, state') ->
    begin
      let comp_unit = KB.Value.get T.Label.unit snapshot in
      inspect_comp_unit comp_unit state'
    end
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e

let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let program = create_program addr target in

  let state = Toplevel.current () in
  inspect_program program state
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean
```

```
build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out:

```
- Target: bap:armv7+le
- Endianness: core:le
- Wordsize: 32
- Memory address size: 32
- Instruction size (program counter size): 32
- Instruction alignment: 32
- Memory variable (its name): mem
- SP: SP
- Registers: CF, LR, NF, QF, R0, R1, R10, R11, R12, R2, R3, R4, R5, R6, R7, R8, R9, SP
```

Clean up:

```
make clean
```

Other ways to get the target

If you have a label, there is a shortcut to get the target:

```
let* target = Theory.Label.target label in
...
```

From a pass, you can use `Project.target`:

```
let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
  let target = Project.target proj in
  ...
```

Documentation

To see more options, see the documentation.

C.4.7 Memory

A program label can have a chunk of memory associated with it. This is stored in the label's memory slot.

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a function to get the endianness:

```
let get_endianness (target : T.Target.t) : endian =
  let endianness = T.Target.endianness target in
  if T.Endianness.(equal endianness le)
    then LittleEndian
    else BigEndian
```

Add a function to create a blank (zeroed-out) chunk of memory:

```
let create_mem (addr : Word.t) (size : int) (target : T.Target.t) : Memory.t =
  let endianness = get_endianness target in
  let addr_size = T.Target.data_addr_size target in
  let num_bytes = size * (addr_size / 8) in
  let bytes = Bytes.init num_bytes ~f:(fun _ -> '\x00') in
  let data = Bigstring.of_bytes bytes in
  let result = Memory.create endianness addr data in
  match result with
  | Ok mem -> mem
  | _ -> failwith "Failed to create memory"
```

Add a function to create a label, with a chunk of memory:

```
let create_label (addr : Word.t) (target : T.Target.t) (mem_size : int)
    : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let addr_bv = Word.to_bitvec addr in
  let* () = KB.provide T.Label.addr label (Some addr_bv) in
  let mem = create_mem addr mem_size target in
  let* () = KB.provide Memory.slot label (Some mem) in
  KB.return label
```

Add a function to create a compilation unit:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Add a functon to create a program:

```
let create_program (addr : Word.t) (target : T.Target.t) (mem_size : int)
    : T.Label.t KB.t =
  let* label = create_label addr target mem_size in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Add a function to take a snapshot and inspect the result:

```
let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) -> Format.printf "Program: %a\n%!" KB.Value.pp snapshot
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e
```

Finally, create the program and trigger the snapshot:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Word.of_int 0x10400 ~width:(T.Target.bits target) in
  let mem_size = 16 in
  let program = create_program addr target mem_size in

  let state = Toplevel.current () in
  inspect_program program state
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let get_endianness (target : T.Target.t) : endian =
  let endianness = T.Target.endianness target in
  if T.Endianness.(equal endianness le)
    then LittleEndian
    else BigEndian

let create_mem (addr : Word.t) (size : int) (target : T.Target.t) : Memory.t =
  let endianness = get_endianness target in
  let addr_size = T.Target.data_addr_size target in
  let num_bytes = size * (addr_size / 8) in
  let bytes = Bytes.init num_bytes ~f:(fun _ -> '\x00') in
  let data = Bigstring.of_bytes bytes in
  let result = Memory.create endianness addr data in
  match result with
```

```
  | Ok mem -> mem
  | _ -> failwith "Failed to create memory"

let create_label (addr : Word.t) (target : T.Target.t) (mem_size : int)
    : T.Label.t KB.t =
  let* label = KB.Object.create T.Program.cls in
  let addr_bv = Word.to_bitvec addr in
  let* () = KB.provide T.Label.addr label (Some addr_bv) in
  let mem = create_mem addr mem_size target in
  let* () = KB.provide Memory.slot label (Some mem) in
  KB.return label

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Word.t) (target : T.Target.t) (mem_size : int)
    : T.Label.t KB.t =
  let* label = create_label addr target mem_size in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label

let inspect_program (program : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls program state in
  match result with
  | Ok (snapshot, _) -> Format.printf "Program: %a\n%!" KB.Value.pp snapshot
  | Error e -> Format.printf "Error: %a\n%!" KB.Conflict.pp e

let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Word.of_int 0x10400 ~width:(T.Target.bits target) in
  let mem_size = 16 in
  let program = create_program addr target mem_size in

  let state = Toplevel.current () in
  inspect_program program state
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################
```

```
.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
        dune clean

build:
        dune build ./$(EXE)

run: build
        dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out:

```
Program: ((bap:mem
           ("00010400  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
             |................|
             00010410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
             |................|
             00010420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
             |................|
             00010430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
             |................|"))
          (bap:arch armv7)
          (core:semantics
           ((core:insn-code
             ("00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
               00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
               00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
               00"))))
          (core:label-addr (0x10400))
          (core:label-unit (2))
          (core:encoding bap:llvm-armv7))
```

Clean up:

```
make clean
```

C.4.8 Retrieving a label

When BAP disassembles a binary program, it fills the slots for each label as best as it can.

You can take a snapshot of any program label, if you know its address.

A toy executable

First, create a toy executable program that we can use for the example.

In a new folder somewhere, create a sub-folder called `resources`. Inside of the `resources` folder, create an assembly file `main.asm` with the following contents:

```
    global main:function (main.end - main)

; -------------------------------------------
    section .text
; -------------------------------------------

main:
    mov rdi, 3
    mov rax, 60
    syscall
.end:
```

This program will simply exit with an exit code of 3.

In the `resources` sub-folder, add a `Makefile`:

```
SRC := main.asm
OBJ := main.o
EXE := main.elf


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all
all: clean build


#####################################################
# BUILD
#####################################################

$(EXE): $(SRC)
        nasm -w+all -f elf64 -o $(OBJ) $(SRC)
        ld -e main -o $(EXE) $(OBJ)
        rm -rf $(OBJ)

build: $(EXE)


#####################################################
# CLEAN
#####################################################

.PHONY: clean
clean:
        rm -rf $(OBJ) $(EXE)
```

Build and compile:

```
make -C resources
```

Run the compiled program:

```
./resources/main.elf
```

Confirm that it returns an exit code of 3:

```
echo ${?}
```

Use `objdump` to view the program:

```
objdump -Ds resources/main.elf
```

Identify the address of the `main` function. For me, it's `0x400080`.

A BAP pass

The next task is to create a BAP pass that can inspect a label.

In the folder that is parent to the `resources` sub-folder, create a file called `kb_pass_01.ml` with these contents at the top:

```
open Core_kernel
open Bap.Std
```

Create a sub-module:

```
module Analysis = struct

  module T = Bap_core_theory.Theory
  module KB = Bap_core_theory.KB
  open KB.Let

  (* We'll add code to inspect the label here ... *)

end
```

Add a function that retrieves the label for a given address:

```
module Analysis = struct

  ...

  let get_label (addr : Bitvec.t) : T.Label.t KB.t =
    let* label = T.Label.for_addr addr in
    KB.return label

end
```

And add a function that takes a snapshot of a label and prints the result:

```
module Analysis = struct

  ...

  let explore (addr : Bitvec.t) : unit =
    let label = get_label addr in
    let state = Toplevel.current () in
    let result = KB.run T.Program.cls label state in
    match result with
    | Ok (snapshot, _) ->
      begin
        Format.printf "Program: %a\n%!" KB.Value.pp snapshot
      end
    | Error e -> Format.printf "KB error: %a\n%!" KB.Conflict.pp e

end
```

Create another sub-module:

```
module Setup = struct

  module Conf = Bap_main.Extension.Configuration
  module Param_type = Bap_main.Extension.Type

  (* We'll set up the BAP pass here... *)

end
```

Define a command line parameter that takes an address (as a string):

```
module Setup = struct

  ...

  let addr = Conf.parameter Param_type.string "addr"

end
```

Add a pass which runs `Analysis.explore` on the provided address (provided the address is not empty):

```
module Setup = struct

  ...

  let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
    let addr = Conf.get ctxt addr in
    let () =
      if String.is_empty addr
        then failwith "No address specified"
        else ()
    in
    let word = Bitvec.of_string addr in
    Analysis.explore word

end
```

Add a function that registers the pass:

```
module Setup = struct

  ...

  let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
    Project.register_pass' (pass ctxt);
    Ok ()

end
```

Finally, declare `run` as an extension:

```
let () = Bap_main.Extension.declare Setup.run
```

To summarize, the whole `kb_pass_01.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module Analysis = struct

  module T = Bap_core_theory.Theory
  module KB = Bap_core_theory.KB
  open KB.Let

  let get_label (addr : Bitvec.t) : T.Label.t KB.t =
    let* label = T.Label.for_addr addr in
    KB.return label

  let explore (addr : Bitvec.t) : unit =
    let label = get_label addr in
    let state = Toplevel.current () in
    let result = KB.run T.Program.cls label state in
    match result with
    | Ok (snapshot, _) ->
      begin
        Format.printf "Program: %a\n%!" KB.Value.pp snapshot
      end
    | Error e -> Format.printf "KB error: %a\n%!" KB.Conflict.pp e

end

module Setup = struct

  module Conf = Bap_main.Extension.Configuration
  module Param_type = Bap_main.Extension.Type

  let addr = Conf.parameter Param_type.string "addr"

  let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
    let addr = Conf.get ctxt addr in
    let () =
```

```
        if String.is_empty addr
          then failwith "No address specified"
          else ()
      in
      let word = Bitvec.of_string addr in
      Analysis.explore word

    let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
      Project.register_pass' (pass ctxt);
      Ok ()

end

let () = Bap_main.Extension.declare Setup.run
```

Add a `Makefile`:

```
PUBLIC_NAME := my-kb-pass-01
PUBLIC_DESC := My demo KB pass 01

NAME := kb_pass_01
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Confirm that the plugin (which is named `my-kb-pass-01`) is installed:

```
bap list plugins
```

Now run the pass over the toy executable, providing the address of the `main` function as the `addr` argument:

```
bap resources/main.elf --my-kb-pass-01 --my-kb-pass-01-addr 0x400080
```

It should print out a fair amount of information about the program at that location:

```
Program: ((bap:lisp-args
           ((((lisp-symbol (EDI)) (bap:exp EDI))
             ((bap:static-value (0x3)) (bap:exp 3)))))
          (bap:lisp-name (llvm-x86_64:MOV32ri))
          (bap:insn ((MOV32ri EDI 0x3)))
          (bap:mem ("400080: bf 03 00 00 00"))
          ...
```

Try to run it with an address that does not exist in the binary, e.g.:

```
bap resources/main.elf --my-kb-pass-01 --my-kb-pass-01-addr 0x555555
```

It will print out nothing but an address:

```
Program: ((core:label-addr (0x555555)))
```

This is because there is no program information about this program label, since the label does not exist in the binary.

Clean up:

```
make uninstall
make clean
```

Clean up the toy executable in the `resources` sub-folder too if you like:

```
make clean -C resources
```

C.5 Semantics

C.5.1 Semantics

The "semantics" of a program at a label is, roughly speaking, what the program "does" to the state of the machine when it executes. BAP represents this as the following type:

```
'a Theory.effect
```

In other words, the semantics of a program at a particulal label is the "effect" the program will produce in a machine's state.

Some examples:

- The semantics of assigning a variable is to set a value for an identifier in the machine's state.

- The semantics of calculating a sum and putting the result in a register involves assigning a value to a register (a variable), but it might also involve updating a zero flag (another variable), an overflow flag (yet another variable), and so on.
- The semantics of writing to memory is to put a value in the memory at a particular address.
- The semantics of jumping to some other label in the program is to update the value of the instruction pointer to the targeted label.

To encode the semantics of programs, BAP provides what it calls a "core theory." A core theory is a very general kind of assembly language. It has variables, integers, floating point numbers, jumps, if-then-elses, and the like.

When you want to stipulate what a program does at a particular label, you write it down as a core theory program, and you put that core theory program in the label's "semantics" slot (`Theory.Semantics.slot`). That core theory program then represents what the program "does" at that particular label.

Toy example

For instance, suppose we have a simple binary program, and we are looking at a particular label that moves the number 3 into a register R2 (in pseudo-assembly):

```
mov R2, 0x03
```

What is the semantics of the program at this label? The semantics is what it "does," and what it "does" is assign the integer `3` to the variable `R2`.

To encode that, we create a core theory program that assigns 3 to r2, which looks something like this (in pseudo-core theory code):

```
(set (var R2) (int 0x03))
```

Then, we store this little core theory program in the "semantics" slot of this program label.

This little core theory program then "represents" what the program "does" at that particular label.

At any point in a later analysis, if we want to know what the program does at that particular label, we can just look in the label's "semantics" slot to see.

Another toy example

Suppose we have a binary program that is meant to run on a machine with 8-bit words, a zero flag, and an overflow flag. Suppose we are looking at a particular label that adds 255 and 1, and then stores the result in the register R2 (in pseudo assembly):

```
mov R2, 0xff + 0x01
```

What is the semantics of the program at this label?

The semantics of this program is what it "does" to the state of the machine when it executes, and that involves three things.

- `R2` (a variable) is set to the sum of `0xff` and `0x01`, which is `0x00` (because the machine has 8 bit words, so it cannot hold anything bigger than `0xff`).

- Since the result of adding `0xff` and `0x01` is zero, the zero flag `ZF` (another variable) is set to `0x01`.
- Third, since adding `0xff` and `0x01` is an overflow, the overflow flag `OF` (another variable) is set to `0x01`.

To encode this, we could create a core theory program that does all three of these things. It might look something like this (in pseudo-core theory code):

```
(
  (set (var R2) (add (int 0xff) (int 0x01)))
  (set (var ZF) (int 0x01))
  (set (var OF) (int 0x01))
)
```

We can then store this little core theory program in the "semantics" slot of this particular program label, so that at any later time, we can look up what the program at this particular label "does."

Documentation

For more details, see the documentation.

C.5.2 Providing Semantics

The simplest core theory program is a block. Blocks contain two parts:

- A data part
- A control part

The data part contains variable assignments. In other words, this part of the block only has data effects, i.e., it updates the values of variables.

The control part contains instructions that have control effects, i.e., instructions which change the control flow of the program, e.g., gotos and jumps.

The simplest block is an empty block (it has an empty data part, and an empty control part). This is basically just a fallthrough.

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Start a function to create the semantics for a label:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  (* We'll build an empty block here *)
```

Get an instance of the core theory module:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
```

Define a nop as a type of data effect:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
```

Create a data part that's just a nop:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
```

Create an empty control part (a fallthrough):

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
```

Finally, create a core theory block from the specified data and control parts:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl
```

Next, add a function that creates a compilation unit:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Create a function that creates a program label and adds a compilation unit and semantics to it:

```
let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  let* semantics = create_semantics label in
  let* () = KB.provide T.Semantics.slot label semantics in
  KB.return label
```

Create a function that takes a snapshot of a program label and prints the result:

```
let inspect_program (label : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls label state in
  match result with
  | Ok (snapshot, _) -> Format.printf "%a\n%!" KB.Value.pp snapshot
  | Error e -> Format.printf "%a\n%!" KB.Conflict.pp e
```

Finally, trigger the whole thing:

```
let () =
  let state = Toplevel.current () in
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  inspect_program label state
```

To summarize, the entire `main.ml` file looks like this:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  let* semantics = create_semantics label in
  let* () = KB.provide T.Semantics.slot label semantics in
  KB.return label

let inspect_program (label : T.Label.t KB.t) (state : KB.state) : unit =
  let result = KB.run T.Program.cls label state in
  match result with
  | Ok (snapshot, _) -> Format.printf "%a\n%!" KB.Value.pp snapshot
  | Error e -> Format.printf "%a\n%!" KB.Conflict.pp e
```

```
let () =
  let state = Toplevel.current () in
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  inspect_program label state
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot:

```
((bap:arch armv7)
 (core:semantics
  ((bap:ir-graph "00000009:
                 ")
   (bap:insn-dests (()))
   (bap:bil "{
              label(%00000009)
             }")))
 (core:label-addr (0x10400))
 (core:label-unit (2))
 (core:encoding bap:llvm-armv7))
```

Clean up:

```
make clean
```

Documentation

For more details about core theory programs, see the documentation.

C.5.3 Promising semantics

You can use `KB.promise` to promise values for slots. Whenever you use `KB.collect` to retrieve the data in a slot, BAP will trigger all the promises, in order to compute the best information at that time.

However, if you use `KB.provide` to fill a slot, then BAP will consider that to be the definitive value for that slot. If you use `KB.collect` after that, BAP will *not* trigger any of the promises.

Various parts of BAP's inner plumbing relies on promises associated with the semantics slot. Whenever you want to provide semantics, it is therefore best to use `KB.promise`. If you use `KB.provide` to put semantics into `Theory.Semantics.slot`, you may prevent other promises from firing.

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a function that generates semantics (an empty block):

```
let provide_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl
```

Register this function as a promise for `T.Semantics.slot`:

```
let () = KB.promise T.Semantics.slot provide_semantics
```

Next, add a function that creates a compilation unit:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Add a function that creates a program label and adds the compilation unit to it:

```
let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Notice that no semantics have been provided (instead, they have been promised).

Now, create a program, and use `Toplevel.eval` to evaluate its semantics:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

When `Toplevel.eval` runs, it will collect the semantics for the given label, and that will trigger the promise registered above.

To summarize, the entire `main.ml` file looks like this:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let provide_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let nop = T.Effect.Sort.data "NOP" in
  let data = KB.return (T.Effect.empty nop) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl

let () = KB.promise T.Semantics.slot provide_semantics

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot:

```
(bap:ir-graph "00000009:
                ")
 (bap:insn-dests (()))
 (bap:bir (%00000009))
 (bap:bil "{
             label(%00000009)
           }"))
```

Clean up:

```
make clean
```

C.5.4 Variable assignments

As an example of a very simple program, consider one that moves, say, the integer 3 into a register R2 (in pseudo-assembly):

```
mov R2, 0x03
```

What is the semantics of this program? The semantics is what it "does" when executed, and in this case, it simply assigns the value `0x03` to a variable `R2`. In pseudo-core theory:

```
(set (var R2) (int 0x03))
```

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a function that creates a binary word of a certain size:

```
let create_word (i : int) (bits : int) : Bitvec.t =
  let m = Bitvec.modulus bits in
  Bitvec.(int i mod m)
```

Start a function to create the semantics for a label:

```
let create_semantics (label : T.Label.t) : 'a T.effect KB.t =
  (* We'll build the core-theory program here *)
```

Get an instance of the core theory module:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
```

Define a bitvector type with the target's word size, and create a binary word with that width to represent `0x03`:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
```

Create a core theory integer with that value:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
  let value = CT.int width word in
```

Create a core theory variable `R2`:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
  let value = CT.int width word in
  let var = CT.var width "R2" in
```

Now assign the value to the variable:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
  let value = CT.int width word in
  let var = CT.var width "R2" in
  let data = CT.set var value in
```

Add a fallthrough:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
  let value = CT.int width word in
  let var = CT.var width "R2" in
  let data = CT.set var value in
  let ctrl = KB.return (T.Effect.empty T.EFfect.Sort.fall) in
```

Finally, create a core theory block:

```
let create_semantics (label : T.Label.t) (bits : int) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits
  let value = CT.int width word in
  let var = CT.var width "R2" in
  let data = CT.set var value in
  let ctrl = KB.return (T.Effect.empty T.EFfect.Sort.fall) in
  CT.blk label data ctrl
```

Register the function as a promise:

```
let () = KB.promise T.Semantics.slot provide_semantics
```

Next, add a function that creates a compilation unit:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Add a function that creates a program label and adds a compilation unit to it:

```
let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Now, generate the program, and use `Toplevel.eval` to retrieve the semantics and print the result:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

To summarize, the entire `main.ml` file looks like this:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_word (i : int) (bits : int) : Bitvec.t =
  let m = Bitvec.modulus bits in
  Bitvec.(int i mod m)

let provide_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in
  let word = create_word 0x03 bits in
  let value = CT.int width word in
  let var = T.Var.define width "R2" in
  let data = CT.set var value in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
```

```
  CT.blk label data ctrl

let () = KB.promise T.Semantics.slot provide_semantics

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label

let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot:

```
((bap:ir-graph "00000009:
                 0000000c: R2 := 3")
 (bap:insn-dests (()))
 (bap:bir (%00000009))
 (bap:bil "{
             label(%00000009)
             R2 := 3
           }"))
```

Clean up:

```
make clean
```

C.5.5 Multiple variable assignments

Sometimes a simple assignment involves setting flags too. For instance, suppose on a 32-bit architecture machine that uses status flags, we have a program that adds 0xffffffff and 0x01, and then assigns the result to the register R2:

```
mov R2, 0xffffffff + 0x01
```

What is the semantics of this program? It adds `0xffffffff` and `0x01` and assigns the result to `R2`, but it also sets the zero flag `ZF` and the overflow flag `OF`. In pseudo-core theory code:

```
(
  (set (var R2) (add (int 0xffffffff) (int 0x01)))
  (set (var ZF) (int 0x01))
  (set (var OF) (int 0x01))
)
```

Example

In a new folder somewhere, create a file called `main.ml` with the following:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Create a function that creates a binary word of a certain size:

```
let create_word (i : int) (bits : int) : Bitvec.t =
  let m = Bitvec.modulus bits in
  Bitvec.(int i mod m)
```

Create a function that adds `0xffffffff` and `0x01`, assigns the result to `R2`, sets `ZF` and `OF` to `0x01`, and then sequences those assignments into a block that ends with a fallthrough:

```
let provide_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in

  let one_bv = create_word 0x01 bits in
  let max_int_bv = create_word 0xffffffff bits in

  let one = CT.int width one_bv in
  let max_int = CT.int width max_int_bv in

  let r2 = T.Var.define width "R2" in
  let z_flag = T.Var.define width "ZF" in
  let o_flag = T.Var.define width "OF" in

  let r2_assignment = CT.set r2 (CT.add max_int one) in
  let zf_assignment = CT.set z_flag one in
  let of_assignment = CT.set o_flag one in

  let data = CT.seq r2_assignment (CT.seq zf_assignment of_assignment) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl
```

Register the function as a promise:

```
let () = KB.promise T.Semantics.slot provide_semantics
```

Add a function that creates a compilation unit:

```
let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit
```

Add a function that creates a program label and adds a compilation unit to it:

```
let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

Finally, generate the program and use `Toplevel.eval` to retrieve the semantics and print the result:

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

To summarize, the entire `main.ml` file looks like this:

```
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let create_word (i : int) (bits : int) : Bitvec.t =
  let m = Bitvec.modulus bits in
  Bitvec.(int i mod m)

let provide_semantics (label : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let* target = T.Label.target label in
  let bits = T.Target.bits target in
  let width = T.Bitv.define bits in

  let one_bv = create_word 0x01 bits in
  let max_int_bv = create_word 0xffffffff bits in

  let one = CT.int width one_bv in
  let max_int = CT.int width max_int_bv in

  let r2 = T.Var.define width "R2" in
  let z_flag = T.Var.define width "ZF" in
  let o_flag = T.Var.define width "OF" in

  let r2_assignment = CT.set r2 (CT.add max_int one) in
  let zf_assignment = CT.set z_flag one in
  let of_assignment = CT.set o_flag one in

  let data = CT.seq r2_assignment (CT.seq zf_assignment of_assignment) in
  let ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall) in
  CT.blk label data ctrl

let () = KB.promise T.Semantics.slot provide_semantics

let create_compilation_unit (target : T.Target.t) : T.Unit.t KB.t =
  let* comp_unit = KB.Object.create T.Unit.cls in
  let* () = KB.provide T.Unit.target comp_unit target in
  KB.return comp_unit

let create_program (addr : Bitvec.t) (target : T.Target.t) : T.Label.t KB.t =
  let* label = T.Label.for_addr addr in
  let* comp_unit = create_compilation_unit target in
  let* () = KB.provide T.Label.unit label (Some comp_unit) in
  KB.return label
```

```
let () =
  let target = T.Target.read "bap:armv7+le" in
  let addr = Bitvec.(int 0x10400 mod m32) in
  let label = create_program addr target in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
# THE EXE
#####################################################

.PHONY: clean
clean:
	dune clean

build:
	dune build ./$(EXE)

run: build
	dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot:

```
((bap:ir-graph "00000009:
                0000000c: R2 := 0
                0000000f: ZF := 1
                00000012: OF := 1")
 (bap:insn-dests (()))
 (bap:bir (%00000009))
 (bap:bil "{
             label(%00000009)
```

```
      R2 := 0
      ZF := 1
      OF := 1
    }"))
```

Clean up:

```
make clean
```

C.5.6 "Compiling" to core theory programs

Core theory programs can be used to provide semantics for a custom assembly-like language.

In this example, we will:

- Write a program in a toy assembly-like language.
- Read it in and construct a simple AST.
- Walk the AST and create core theory semantics for the program.

Example

In a new folder somewhere, create a file `program.lisp` with the following contents:

```
(block foo
  (data (
    (set r0 (int 0x3))
    (set r1 (reg r0))))
  (control
    (goto bar)))

(block bar
  (data (
    (set r2 (add (int 0x7) (reg r3)))
    (set r0 (reg r2))))
  (control
    (goto foo)))
```

This program has two blocks (labeled `foo` and `bar`). Each block has a `data` section (with variable assignments), and a `control` section (with `goto`s).

Create a file `ast.ml` with the following types:

```
open Core_kernel

type label = string
type reg = string
type num = int

type expr = Var of reg | Num of num | Add of expr * expr
type assignment = Assign of reg * expr

type data = assignment list
type control = Goto of label | Fallthrough

type block = Block of label * data * control
type t = block list
```

Add some accessor functions:

```
let lhs_of_assignment a = match a with Assign (reg, _) -> reg
let rhs_of_assignment a = match a with Assign (_, expr) -> expr

let label_of_block b = match b with Block (label, _, _) -> label
let data_of_block b = match b with Block (_, data, _) -> data
let control_of_block b = match b with Block (_, _, control) -> control
```

Add a convenience function to generate error messages:

```
let err (msg : string) (sexp : Sexp.t) : string =
  let sexp_str = Sexp.to_string sexp in
  Format.sprintf "Parse error: %s: %s" msg sexp_str
```

Add a function to convert a string to an integer:

```
let parse_number (num : string) : num =
  try int_of_string num
  with _ -> failwith (err "Invalid number" (Sexp.Atom num))
```

Add a function to recursively parse an expression (variables, integers, and additions):

```
let rec parse_expr (sexp : Sexp.t) : expr =
  match sexp with
  | Sexp.List [ Atom "reg"; Atom reg ] -> Var reg
  | Sexp.List [ Atom "int"; Atom n ] -> Num (parse_number n)
  | Sexp.List [ Atom "add"; e1 ; e2 ] ->
    Add (parse_expr e1, parse_expr e2)
  | _ -> failwith (err "Invalid expr" sexp)
```

Add a function to parse the data assignments:

```
let parse_data (sexps : Sexp.t list) : data =
  List.fold sexps ~init:[] ~f:(fun assigns sexp -> match sexp with
    | Sexp.List [ Atom "set"; Atom reg; e ] ->
      List.append assigns [Assign (reg, parse_expr e)]
    | _ -> failwith (err "Invalid data" sexp))
```

Add a function to parse the control section:

```
let parse_ctrl (sexp : Sexp.t) : control =
  match sexp with
  | Sexp.List [ Atom "goto"; Atom label ] -> Goto label
  | Sexp.List [ Atom "fallthrough" ] -> Fallthrough
  | _ -> failwith (err "Invalid control" sexp)
```

Add a function to start parsing:

```
let parse (sexps : Sexp.t list) : t =
  List.fold sexps ~init:[] ~f:(fun blks sexp -> match sexp with
    | Sexp.List [ Atom "block"; Atom label;
        List [ Atom "data"; List data ];
        List [ Atom "control"; ctrl ]; ] ->
      List.append blks [Block (label, parse_data data, parse_ctrl ctrl)]
    | _ -> failwith (err "Parse error" sexp))
```

Add a function to read a program from a file:

```
et from_file (filepath : string) : t =
  let sexps = Sexp.load_sexps filepath in
  parse sexps
```

To summarize, the entire `ast.ml` file looks like this:

```
open Core_kernel

type label = string
type reg = string
type num = int

type expr = Var of reg | Num of num | Add of expr * expr
type assignment = Assign of reg * expr

type data = assignment list
type control = Goto of label | Fallthrough

type block = Block of label * data * control
type t = block list

let lhs_of_assignment a = match a with Assign (reg, _) -> reg
let rhs_of_assignment a = match a with Assign (_, expr) -> expr

let label_of_block b = match b with Block (label, _, _) -> label
let data_of_block b = match b with Block (_, data, _) -> data
let control_of_block b = match b with Block (_, _, control) -> control

let err (msg : string) (sexp : Sexp.t) : string =
  let sexp_str = Sexp.to_string sexp in
  Format.sprintf "Parse error: %s: %s" msg sexp_str

let parse_number (num : string) : num =
  try int_of_string num
  with _ -> failwith (err "Invalid number" (Sexp.Atom num))

let rec parse_expr (sexp : Sexp.t) : expr =
  match sexp with
  | Sexp.List [ Atom "reg"; Atom reg ] -> Var reg
  | Sexp.List [ Atom "int"; Atom n ] -> Num (parse_number n)
  | Sexp.List [ Atom "add"; e1 ; e2 ] ->
    Add (parse_expr e1, parse_expr e2)
  | _ -> failwith (err "Invalid expr" sexp)

let parse_data (sexps : Sexp.t list) : data =
  List.fold sexps ~init:[] ~f:(fun assigns sexp -> match sexp with
    | Sexp.List [ Atom "set"; Atom reg; e ] ->
      List.append assigns [Assign (reg, parse_expr e)]
    | _ -> failwith (err "Invalid data" sexp))

let parse_ctrl (sexp : Sexp.t) : control =
  match sexp with
```

```
    | Sexp.List [ Atom "goto"; Atom label ] -> Goto label
    | Sexp.List [ Atom "fallthrough" ] -> Fallthrough
    | _ -> failwith (err "Invalid control" sexp)

let parse (sexps : Sexp.t list) : t =
  List.fold sexps ~init:[] ~f:(fun blks sexp -> match sexp with
    | Sexp.List [ Atom "block"; Atom label;
        List [ Atom "data"; List data ];
        List [ Atom "control"; ctrl ]; ] ->
      List.append blks [Block (label, parse_data data, parse_ctrl ctrl)]
    | _ -> failwith (err "Parse error" sexp))

let from_file (filepath : string) : t =
  let sexps = Sexp.load_sexps filepath in
  parse sexps
```

Next, create a file `compiler.mli` with the following interface:

```
module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

module Make(CT : T.Core) : sig
  val semantics_of : Ast.t -> unit T.effect KB.t
end
```

Create a file called `compiler.ml` with the following:

```
open Core_kernel

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let
```

Add a module parameterized by the `Core` theory module:

```
module Make(CT : T.Core) = struct

  (* We'll "compile" the semantics here... *)

end
```

Add the following declarations:

```
module Make(CT : T.Core) = struct

  let m = Bitvec.modulus 32
  let width = T.Bitv.define 32
  let nop = T.Effect.Sort.data "NOP"
  let no_data = KB.return (T.Effect.empty nop)
  let no_ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall)
  let empty_blk = CT.blk T.Label.null no_data no_ctrl

end
```

Add a function to generate semantics for expressions:

```
module Make(CT : T.Core) = struct

  ...

  let rec compile_expr (expr : Ast.expr) : 'b T.Bitv.t T.value KB.t =
    match expr with
    | Ast.Var reg -> CT.var (T.Var.define width reg)
    | Ast.Num n -> CT.int width Bitvec.(int n mod m)
    | Ast.Add (e1, e2) -> CT.add (compile_expr e1) (compile_expr e2)

end
```

Add a function to generate semantics for assignments:

```
module Make(CT : T.Core) = struct

  ...

  let compile_assignment (assign : Ast.assignment) : 'a T.effect KB.t =
    let reg = Ast.lhs_of_assignment assign in
    let var = T.Var.define width reg in
    let expr = Ast.rhs_of_assignment assign in
    CT.set var (compile_expr expr)

end
```

Add a function to fold the assignments into a single data section, and a function to generate semantics for the control section:

```
module Make(CT : T.Core) = struct

  ...

  let compile_data (data : Ast.data) : 'a T.effect KB.t =
    List.fold data ~init:no_data ~f:(fun acc assignment ->
      CT.seq (compile_assignment assignment) acc)

  let compile_ctrl (ctrl : Ast.control) : 'a T.effect KB.t =
    match ctrl with
    | Goto dest ->
      let* label = T.Label.for_name dest in
      CT.goto label
    | Fallthrough -> no_ctrl

end
```

Add a function to generate the semantics for blocks:

```
module Make(CT : T.Core) = struct

  ...
```

```
  let compile_blk (blk : Ast.block) : 'a T.effect KB.t =
    let blk_name = Ast.label_of_block blk in
    let* label = T.Label.for_name blk_name in
    let assignments = Ast.data_of_block blk in
    let* data = compile_data assignments in
    let control = Ast.control_of_block blk in
    let* ctrl = compile_ctrl control in
    CT.blk label (KB.return data) (KB.return ctrl)

end
```

Finally, add a function to get the semantics from an AST:

```
module Make(CT : T.Core) = struct

  ...

  let semantics_of (ast : Ast.t) : 'a T.effect KB.t =
    List.fold ast ~init:empty_blk ~f:(fun acc blk ->
      CT.seq (compile_blk blk) acc)

end
```

To summarize the "compiler," the entire `compiler.ml` file looks like this:

```
open Core_kernel

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

module Make(CT : T.Core) = struct

  let m = Bitvec.modulus 32
  let width = T.Bitv.define 32
  let nop = T.Effect.Sort.data "NOP"
  let no_data = KB.return (T.Effect.empty nop)
  let no_ctrl = KB.return (T.Effect.empty T.Effect.Sort.fall)
  let empty_blk = CT.blk T.Label.null no_data no_ctrl

  let rec compile_expr (expr : Ast.expr) : 'b T.Bitv.t T.value KB.t =
    match expr with
    | Ast.Var reg -> CT.var (T.Var.define width reg)
    | Ast.Num n -> CT.int width Bitvec.(int n mod m)
    | Ast.Add (e1, e2) -> CT.add (compile_expr e1) (compile_expr e2)

  let compile_assignment (assign : Ast.assignment) : 'a T.effect KB.t =
    let reg = Ast.lhs_of_assignment assign in
    let var = T.Var.define width reg in
    let expr = Ast.rhs_of_assignment assign in
    CT.set var (compile_expr expr)

  let compile_data (data : Ast.data) : 'a T.effect KB.t =
```

```
      List.fold data ~init:no_data ~f:(fun acc assignment ->
        CT.seq (compile_assignment assignment) acc)

    let compile_ctrl (ctrl : Ast.control) : 'a T.effect KB.t =
      match ctrl with
      | Goto dest ->
        let* label = T.Label.for_name dest in
        CT.goto label
      | Fallthrough -> no_ctrl

    let compile_blk (blk : Ast.block) : 'a T.effect KB.t =
      let blk_name = Ast.label_of_block blk in
      let* label = T.Label.for_name blk_name in
      let assignments = Ast.data_of_block blk in
      let* data = compile_data assignments in
      let control = Ast.control_of_block blk in
      let* ctrl = compile_ctrl control in
      CT.blk label (KB.return data) (KB.return ctrl)

    let semantics_of (ast : Ast.t) : 'a T.effect KB.t =
      List.fold ast ~init:empty_blk ~f:(fun acc blk ->
        CT.seq (compile_blk blk) acc)

end
```

Create a file `main.ml` with the following contents:

```
pen Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"
```

Add a function that uses the compiler to generate semantics:

```
let provide_semantics (ast : Ast.t) (_ : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let module Cmplr = Compiler.Make(CT) in
  Cmplr.semantics_of ast
```

Get the AST and register the promise:

```
let () =
  let ast = Ast.from_file "program.lisp" in
  KB.promise T.Semantics.slot (provide_semantics ast)
```

Finally, generate the program, retrieve its semantics, and print the result:

```
let () =
  let label = KB.Object.create T.Program.cls in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

To summarize, the entire `main.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let () = match Bap_main.init () with
  | Ok () -> ()
  | Error _ -> failwith "Error initializing BAP"

let provide_semantics (ast : Ast.t) (_ : T.Label.t) : 'a T.effect KB.t =
  let* (module CT) = T.current in
  let module Cmplr = Compiler.Make(CT) in
  Cmplr.semantics_of ast

let () =
  let ast = Ast.from_file "program.lisp" in
  KB.promise T.Semantics.slot (provide_semantics ast)

let () =
  let label = KB.Object.create T.Program.cls in
  let program = Toplevel.eval T.Semantics.slot label in
  Format.printf "%a\n%!" KB.Value.pp program
```

Add a `dune` file:

```
(executable
  (name main)
  (libraries findlib.dynload bap))
```

Add a `Makefile`:

```
EXE := main.exe


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean run


#####################################################
```

```
# THE EXE
#####################################################

.PHONY: clean
clean:
        dune clean

build:
        dune build ./$(EXE)

run: build
        dune exec ./$(EXE)
```

Run the program:

```
make
```

It will print out the snapshot:

```
((bap:ir-graph
  "0000000a:
   0000000e: r0 := r2
   00000012: r2 := 7 + r3
   00000015: goto @foo
   0000001f: goto @foo
   00000013:
   00000019: r1 := r0
   0000001c: r0 := 3
   0000001e: goto @bar")
 (bap:insn-dests ((10 19)))
 (bap:bir (@bar @foo))
 (bap:bil
  "{
     label(%0000000a)
     r0 := r2
     r2 := 7 + r3
     call(foo)
     label(%00000013)
     r1 := r0
     r0 := 3
     call(bar)
   }"))
```

Clean up:

```
make clean
```

C.5.7 Custom Theories

BAP's core theory language is actually a *signature*. You can add your own semantics by implementing the signature and registering it with BAP.

In your implementation, you write the methods required by the signature, but you have them return your own custom semantics.

If you register your theory with BAP, then every time that BAP disassembles a binary program, it will build your theory alongside any other registered theories. Later, when you inspect the disassembled program, you can retrieve the semantics your theory provided.

To implement your own theory, start with a module that is an instance of the `Theory.Core` signature:

```
module My_theory : Theory.Core = struct

   (* Implement the signature's methods here... *)

end
```

To make sure that all of the methods of the signature are covered, include the empty theory, which returns empty semantics for all of the methods.

```
module My_theory : Theory.Core = struct

   include Theory.Empty

   (* Implement the signature's methods here... *)

end
```

Then, implement any methods you want to provide semantics for. There are two kinds of semantics you need to provide: you need to provide the semantics of expressions (which the documentation calls "values" or "pure values"), and the semantics of statements (which the documentation calls "effects").

The general procedure is first to do this:

- Create a custom slot to hold custom semantics for expressions.
- Create another custom slot to hold custom semantics for statements.

The methods in the signature build up the semantics of each program compositionally, so they start with the smaller pieces, and then combine them into bigger and bigger pieces. So, when you implement a method in your theory, the general procedure is this:

- Fetch the semantics of the smaller pieces out of the appropriate slots
- Combine those pieces into a bigger piece of semantics
- Stash that bigger piece of semantics in the appropriate slot
- Return the semantics

The following example create a theory which provides as the "semantics" an S-expression-like string of the program.

A toy executable

First, create a toy executable program that we can use for the example.

In a new folder somewhere, create a sub-folder called `resources`. Inside of the `resources` folder, create an assembly file `main.asm` with the following contents:

```
    global main:function (main.end - main)
```

```
; -----------------------------------------------
    section .text
; -----------------------------------------------

main:
    mov rdi, 7
    add rdi, 3
    mov rax, 60
    syscall
.end:
```

In the `resources` sub-folder, add a `Makefile`:

```
SRC := main.asm
OBJ := main.o
EXE := main.elf


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all
all: clean build


#####################################################
# BUILD
#####################################################

$(EXE): $(SRC)
        nasm -w+all -f elf64 -o $(OBJ) $(SRC)
        ld -e main -o $(EXE) $(OBJ)
        rm -rf $(OBJ)

build: $(EXE)


#####################################################
# CLEAN
#####################################################

.PHONY: clean
clean:
        rm -rf $(OBJ) $(EXE)
```

Build and compile:

```
make -C resources
```

Run the compiled program:

```
./resources/main.elf
```

Confirm that it returns an exit code of 10:

```
echo ${?}
```

Use `objdump` to view the program:

```
objdump -Ds resources/main.elf
```

Identify the address of the `main` function. For me, it's `0x400080`.

The custom theory

The next task is to create a custom theory.

In the folder that is parent to the `resources` sub-folder, create a file called `custom.ml`, which the the following contents:

```
module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let
```

Add a name and a package:

```
let name = "my-theory"
let package = "my.org"
```

Add a slot to store the S-expression-like string representation that we'll build for expressions, and a slot to store the S-expression-like string representation that we'll build for statements:

```
let expr_slot : (T.Value.cls, string) KB.slot =
  KB.Class.property T.Value.cls "expr-slot" KB.Domain.string
    ~package

let stmnt_slot : (T.Semantics.cls, string) KB.slot =
  KB.Class.property T.Semantics.cls "stmnt-slot" KB.Domain.string
    ~package
```

Add a module that implements `T.Core`, and include the empty theory:

```
module Theory : T.Core = struct

  include T.Empty

  (* We'll implement a few methods here... *)

end
```

For convenience, define an empty effect, and an empty expression (an empty value):

```
module Theory : T.Core = struct

  ...

  let empty = T.Effect.empty T.Effect.Sort.bot
  let null_of s = T.Value.empty s

end
```

Next, implement the `int` method of the `T.Core` signature. When disassembling, BAP will call this function whenever it encounters a literal integer in an expression.

To represent an integer, we will simply generate a string version of the binary word, and we will stash that in the expressions slot so that it can serve as the "meaning" of the expression.

```
module Theory : T.Core = struct

  ...

  let int (sort : 's T.Bitv.t T.Value.sort) (bv : Bitvec.t)
      : 's T.Bitv.t T.Value.t KB.t =
    let semantics = Format.asprintf "%a" Bitvec.pp bv in
    let snapshot = KB.Value.put expr_slot (null_of sort) semantics in
    KB.return snapshot

end
```

Next, implement the `var` method of the `T.Core` signature. When disassembling, BAP will call this function whenever it encounters a variable in an expression.

To represent a variable, we will simply use the string name of the variable, and we will stash that in the expressions slot so that it can serve as the "meaning" of the expression.

```
module Theory : T.Core = struct

  ...

  let var (var : 'a T.Var.t) : 'a T.Value.t KB.t =
    let name = T.Var.name var in
    let sort = T.Var.sort var in
    let snapshot = KB.Value.put expr_slot (null_of sort) name in
    KB.return snapshot

end
```

Next, implement the `set` method of the `T.Core` signature. When disassembling, BAP will call this function whenever it encounters a variable assignment (i.e. setting a variable to the value of an expression).

To represent a variable assignment statement as a string, we will generate a string of the form `(set VAR (EXPR))`, where we will replace `VAR` with the name of the variable, and we will replace `EXPR` with whatever S-expression-like string we have already generated for it. We will stash the resulting string in the statements slot so that it can serve as the "meaning" of the whole variable assignment statement:

```
module Theory : T.Core = struct

  ...

  let set (var : 'a T.Var.t) (expr : 'a T.Value.t KB.t)
      : T.data T.Effect.t KB.t =
    let* e = expr in
```

```
    let lhs = T.Var.name var in
    let rhs = KB.Value.get expr_slot e in
    let semantics = Format.sprintf "set %s (%s)" lhs rhs in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

end
```

Next, implement the `blk` method of the `T.Core` signature. When disasembling, BAP will call this function whenever it encounters a block.

To represent a block, we will generate a string `(DATA) (CTRL)`, and we will replace `DATA` with whatever S-expression-like string we have already generated for the data part of the block, and we will replace `CTRL` with whatever S-expression-like string we have already generated for the control part of the block.

```
module Theory : T.Core = struct

  ...

  let blk (label : T.Label.t) (data : T.data T.Effect.t KB.t)
      (ctrl : T.ctrl T.Effect.t KB.t) : unit T.Effect.t KB.t =
    let* d = data in
    let* c = ctrl in
    let sem1 = KB.Value.get stmnt_slot d in
    let sem2 = KB.Value.get stmnt_slot c in
    let semantics = Format.sprintf "(%s) (%s)" sem1 sem2 in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

end
```

Finally, implement the `seq` method of the `T.Core` signature. When disassmebling, BAP will call this function whenever it encounters a sequence of two statements.

To represent a sequence of two statements, we will generate a string `(STMNT1 STMNT2)`, where we replace `STMNT1` and `STMNT2` with the strings we have already generated for the two statements:

```
module Theory : T.Core = struct

  ...

  let seq (prog1 : 'a T.Effect.t KB.t) (prog2 : 'a T.Effect.t KB.t)
      : 'a T.Effect.t KB.t =
    let* p1 = prog1 in
    let* p2 = prog2 in
    let sem1 = KB.Value.get stmnt_slot p1 in
    let sem2 = KB.Value.get stmnt_slot p2 in
    let semantics = Format.sprintf "(%s %s)" sem1 sem2 in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

end
```

There are further methods we could implement, but this is enough for an example. Any methods we did not implement are implemented by the `T.Empty` theory (which will simply return empty snapshots for anything we did not implement).

To summarize, the entire `custom.ml` file looks like this:

```
module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

let name = "my-theory"
let package = "my.org"

let expr_slot : (T.Value.cls, string) KB.slot =
  KB.Class.property T.Value.cls "expr-slot" KB.Domain.string
    ~package

let stmnt_slot : (T.Semantics.cls, string) KB.slot =
  KB.Class.property T.Semantics.cls "stmnt-slot" KB.Domain.string
    ~package

module Theory : T.Core = struct

  include T.Empty

  let empty = T.Effect.empty T.Effect.Sort.bot
  let null_of s = T.Value.empty s

  let int (sort : 's T.Bitv.t T.Value.sort) (bv : Bitvec.t)
      : 's T.Bitv.t T.Value.t KB.t =
    let semantics = Format.asprintf "%a" Bitvec.pp bv in
    let snapshot = KB.Value.put expr_slot (null_of sort) semantics in
    KB.return snapshot

  let var (var : 'a T.Var.t) : 'a T.Value.t KB.t =
    let name = T.Var.name var in
    let sort = T.Var.sort var in
    let snapshot = KB.Value.put expr_slot (null_of sort) name in
    KB.return snapshot

  let set (var : 'a T.Var.t) (expr : 'a T.Value.t KB.t)
      : T.data T.Effect.t KB.t =
    let* e = expr in
    let lhs = T.Var.name var in
    let rhs = KB.Value.get expr_slot e in
    let semantics = Format.sprintf "set %s (%s)" lhs rhs in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

  let blk (label : T.Label.t) (data : T.data T.Effect.t KB.t)
      (ctrl : T.ctrl T.Effect.t KB.t) : unit T.Effect.t KB.t =
    let* d = data in
    let* c = ctrl in
```

```
    let sem1 = KB.Value.get stmnt_slot d in
    let sem2 = KB.Value.get stmnt_slot c in
    let semantics = Format.sprintf "(%s) (%s)" sem1 sem2 in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

  let seq (prog1 : 'a T.Effect.t KB.t) (prog2 : 'a T.Effect.t KB.t)
      : 'a T.Effect.t KB.t =
    let* p1 = prog1 in
    let* p2 = prog2 in
    let sem1 = KB.Value.get stmnt_slot p1 in
    let sem2 = KB.Value.get stmnt_slot p2 in
    let semantics = Format.sprintf "(%s %s)" sem1 sem2 in
    let snapshot = KB.Value.put stmnt_slot empty semantics in
    KB.return snapshot

end
```

A BAP pass

The next task is to register our theory, then create a BAP pass that can inspect a label, so that we can see the semantics our theory has provided for the program.

In the folder that is parent to the `resources` sub-folder, create a file called `kb_pass_02.ml` with these contents at the top:

```
open Core_kernel
open Bap.Std
```

Create a sub-module which contains a BAP pass that can explore the label of a particular address:

```
module Setup = struct

  module Conf = Bap_main.Extension.Configuration
  module Param_type = Bap_main.Extension.Type

  let addr = Conf.parameter Param_type.string "addr"

  let explore (addr : Bitvec.t) : unit =
    let label = T.Label.for_addr addr in
    let semantics = Toplevel.eval T.Semantics.slot label in
    Format.printf "%a\n%!" KB.Value.pp semantics

  let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
    let addr = Conf.get ctxt addr in
    let () =
      if String.is_empty addr
        then failwith "No address specified"
        else ()
    in
    let word = Bitvec.of_string addr in
    explore word

end
```

Next, create a function that registers our theory and the pass:

```
module Setup = struct

  ...

  let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
    let theory = KB.return (module Custom.Theory : T.Core) in
    T.declare theory ~package:Custom.package ~name:Custom.name;
    Project.register_pass' (pass ctxt);
    Ok ()

end
```

Finally, register the extension:

```
let () = Bap_main.Extension.declare Setup.run
```

To summarize, the whole `kb_pass_02.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

module Setup = struct

  module Conf = Bap_main.Extension.Configuration
  module Param_type = Bap_main.Extension.Type

  let addr = Conf.parameter Param_type.string "addr"

  let explore (addr : Bitvec.t) : unit =
    let label = T.Label.for_addr addr in
    let semantics = Toplevel.eval T.Semantics.slot label in
    Format.printf "%a\n%!" KB.Value.pp semantics

  let pass (ctxt : Bap_main.ctxt) (proj : Project.t) : unit =
    let addr = Conf.get ctxt addr in
    let () =
      if String.is_empty addr
        then failwith "No address specified"
        else ()
    in
    let word = Bitvec.of_string addr in
    explore word

  let run (ctxt : Bap_main.ctxt) : (unit, Bap_main.error) Stdlib.result =
    let theory = KB.return (module Custom.Theory : T.Core) in
    T.declare theory ~package:Custom.package ~name:Custom.name;
    Project.register_pass' (pass ctxt);
    Ok ()

end

let () = Bap_main.Extension.declare Setup.run
```

Add a `Makefile`:

```
PUBLIC_NAME := my-kb-pass-02
PUBLIC_DESC := My demo KB pass 02

NAME := kb_pass_02
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Confirm that the plugin (which is named `my-kb-pass-02`) is installed:

```
bap list plugins
```

Now run the pass over the toy executable, providing the address of the `main` function as the `addr` argument:

```
bap resources/main.elf --my-kb-pass-02 --my-kb-pass-02-addr 0x400080
```

It should print out a fair amount of information about the program at that location, something like this:

```
((bap:ir-graph "0000001f:
                00000020: RDI := 7")
 (bap:insn-dests (()))
 (bap:insn-ops ((EDI 7)))
 (bap:insn-asm "movl $0x7, %edi")
 ...
```

If you do not see `((my.org:stmnt-slot ...` in the output, clean the cache:

```
bap cache --clean
```

Then try again:

```
bap resources/main.elf --my-kb-pass-02 --my-kb-pass-02-addr 0x400080
```

This time, you should see something about `my.org:stmnt-slot` in the output:

```
((my.org:stmnt-slot "((set RDI (0x7)) () () ())")
 (bap:ir-graph "0000001f:
                00000020: RDI := 7")
 (bap:insn-dests (()))
 (bap:insn-ops ((EDI 7)))
 (bap:insn-asm "movl $0x7, %edi")
 ...
```

Notice the "semantics" that our custom theory has provided:

```
((set RDI (0x7)) () () ())
```

We can see that our theory has generated an S-expression-like string that expresses the "meaning" of the program at this particular program label. (The empty parantheses are there because we did not implement all of the methods in the `Theory.Core` signature.)

Clean up:

```
make uninstall
make clean
```

Clean up the toy executable in the `resources` sub-folder too if you like:

```
make clean -C resources
```

Other examples

For a fuller example, see the BIL plugin semantics. It constructs BIL expressions and puts them in Exp.slot, and it constructs BIL statement lists and puts them in the Bil.slot.

Documentation

For the full `Theory.Core` signature and all of the methods you can implement, see the documentation.

C.6 KB Analyses

C.6.1 About KB Analyses

In BAP terminology, a "pass" is an analysis that explores a disassembled project/program. A "KB analysis" is a similar idea, but a KB analysis explores the knowledge base.

You can register your KB analyses with BAP, and specify a command line interface to them. BAP has a REPL that lets you run your command from the REPL, but you can also trigger the command in batch mode, or even from a script.

The REPL

To start a bare REPL (i.e., with no knowledge base loaded up), type the following and hit `<enter>`:

```
bap analyze
```

You will see a REPL prompt:

```
bap>
```

To list the registered commands, type `commands` and hit `<enter>`:

```
bap> commands
```

BAP will then print the list of available commands, e.g.:

```
bap:subroutines                          prints all subroutines
bap:units                                prints all units
bap:subroutine                           prints a subroutine
bap:instructions                         prints all instructions
bap:instruction                          prints the instruction semantics
```

The commands are listed on the left, and a description of each one is on the right.

To see the help for any command, type `help <command>`. For instance:

```
bap> help bap:units
bap> help bap:subroutines
```

Try to run a command. For instance, type `bap:units` and hit `<enter>`:

```
bap> bap:units
```

It prints nothing, and returns the REPL prompt:

```
bap>
```

This is because we started the REPL without any knowledge base loaded up.

To exit the REPL, type `quit` and hit `<enter>`:

```
bap> quit
```

Loading a knowledge base

The REPL can load up a knowledge base from a file. First, dump a project to a file. For instance, to dump `/bin/true` to a file called `test.proj`:

```
bap /path/to/bin --project test.proj --update
```

Next, start the analysis REPL with that project:

```
bap analyze --project test.proj
```

That starts the REPL, with the knowledge base for the project loaded into memory.

Now that we have a knowledge base loaded up, run the `bap:units` command:

```
bap> bap:units
```

It lists the sole compilation unit of the program:

```
file:/bin/true                          bap:amd64
```

The name of the code unit is on the left, and the name of the target architecture is on the right.

Now use the `bap:subroutines` command to list the subroutines associated with this code unit:

```
bap> bap:subroutines file:/bin/true
```

The output is a big list of all of the subroutines in the project:

```
/bin/true:__ctype_b_loc\:external: __ctype_b_loc:external
/bin/true:iswprint\:external: iswprint:external
/bin/true:mbsinit\:external: mbsinit:external
/bin/true:__fprintf_chk\:external: __fprintf_chk:external
/bin/true:fwrite\:external: fwrite:external
/bin/true:exit\:external: exit:external
...
```

Exit the REPL:

```
bap> quit
```

Documentation

To see the source code for the built-in analysis commands, see here. For more about KB analyses in general, see the documentation.

C.6.2 A simple analysis plugin

Here is an example of how to build and register a custom KB analysis.

Example

In a new folder somewhere, create a folder called `project_analysis_01.ml` with the following code at the top of the file:

```
open Bap.Std

module KB = Bap_core_theory.KB
module A = Project.Analysis
```

Provide a name, package, and description for a command:

```
let name = "hello-world"
let package = "my.org"
let desc = "prints 'hello world'"
```

Add a "grammar" of command line arguments that the command accepts:

```
let grammar = A.(args empty)
```

This grammar says that the command takes only an empty argument.

Now that we have specified a name for the command a grammar, we need to implement it. To do that, we need to create a function that takes an empty argument (unit), and returns `unit KB.t`. We'll just have it print "hello world":

```
let do_hello_world () : unit KB.t =
  print_endline "Hello world";
  KB.return ()
```

Note: the arguments that this function accepts need to correspond exactly to the grammar specified. In this case, the grammar `A.(args empty)` says the command just takes an empty argument, and so the function here also accepts an empty argument (a thunk).

Now that we have specifed a name and a grammar for the command, and written a function to implement the command, let's register the command:

```
let () =
  A.register name grammar do_hello_world
    ~desc
    ~package
```

To summarize, the entire `project_analysis_01.ml` file looks like this:

```
open Bap.Std

module KB = Bap_core_theory.KB
module A = Project.Analysis

let name = "hello-world"
let package = "my.org"
let desc = "prints 'hello world'"
let grammar = A.(args empty)

let do_hello_world () : unit KB.t =
  print_endline "Hello world";
  KB.return ()

let () =
  A.register name grammar do_hello_world
    ~desc
    ~package
```

Add a `Makefile`:

```
PUBLIC_NAME := my-project-analysis-01
PUBLIC_DESC := My project analysis 01

NAME := project_analysis_01
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################

.DEFAULT_GOAL := all

all: clean uninstall install


#####################################################
# THE PLUGIN
#####################################################

.PHONY: clean
clean:
        bapbuild -clean

uninstall:
        bapbundle remove $(PLUGIN)

build: $(SRC)
        bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
        bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
        bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
        bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Start the analysis REPL:

```
bap analyze
```

List the installed commands:

```
bap> commands
```

You should see your `my.org:hello-world` command in the list:

```
bap:subroutines                         prints all subroutines
my.org:hello-world                      prints 'hello world'
bap:units                               prints all units
bap:subroutine                          prints a subroutine
bap:instructions                        prints all instructions
bap:instruction                         prints the instruction semantics
```

Try running it:

```
bap> my.org:hello-world
```

It prints the expected result:

```
Hello world
```

Quit the REPL:

```
bap> quit
```

Clean up:

```
make uninstall clean
```

C.6.3 A more complex analysis plugin

Here is an example of a slightly more complex KB analysis plugin.

Example

In a new folder somewhere, create a folder called `project_analysis_02.ml` with the following code at the top of the file:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

module A = Project.Analysis
```

Provide a name, package, description, and grammar for a command:

```
let name = "encoding"
let package = "my.org"
let desc = "prints the encoding of a given address"
let grammar = A.(args @@ program $ flag "show-name")
```

Note that the grammar says that the command takes a `program` argument (i.e., a program label), and a flag `show-name`.

Create a function that takes a program label, and a boolean flag. Have it retrieve the encoding of the given label, and if the flag is specified, have it also print the name of the program label (if it has a name).

```
let show_encoding (label : T.Label.t) (show_name : bool) : unit KB.t =
  let* encoding = KB.collect T.Label.encoding label in
  Format.printf "Encoding: %s\n%!" (T.Language.to_string encoding);
  if show_name then
    let* name = KB.collect T.Label.name label in
    let repr = Option.value name ~default:"none" in
    Format.printf "Name: %s\n%!" repr;
    KB.return ()
  else
    KB.return ()
```

Finally, register the command:

```
let () =
  A.register name grammar show_encoding
    ~desc
    ~package
```

To summarize, the entire `project_analysis_02.ml` file looks like this:

```
open Core_kernel
open Bap.Std

module T = Bap_core_theory.Theory
module KB = Bap_core_theory.KB

open KB.Let

module A = Project.Analysis

let name = "encoding"
let package = "my.org"
let desc = "prints the encoding of a given address"
let grammar = A.(args @@ program $ flag "show-name")

let show_encoding (label : T.Label.t) (show_name : bool) : unit KB.t =
  let* encoding = KB.collect T.Label.encoding label in
  Format.printf "Encoding: %s\n%!" (T.Language.to_string encoding);
  if show_name then
    let* name = KB.collect T.Label.name label in
    let repr = Option.value name ~default:"none" in
    Format.printf "Name: %s\n%!" repr;
    KB.return ()
  else
    KB.return ()

let () =
  A.register name grammar show_encoding
    ~desc
    ~package
```

Add a `Makefile`:

```
PUBLIC_NAME := my-project-analysis-02
PUBLIC_DESC := My project analysis 02

NAME := project_analysis_02
SRC := $(NAME).ml
PLUGIN := $(NAME).plugin


#####################################################
# DEFAULT
#####################################################
```

```
.DEFAULT_GOAL := all

all: clean uninstall install


##################################################
# THE PLUGIN
##################################################

.PHONY: clean
clean:
	bapbuild -clean

uninstall:
	bapbundle remove $(PLUGIN)

build: $(SRC)
	bapbuild -use-ocamlfind -package findlib.dynload $(PLUGIN)

install: build
	bapbundle update -name $(PUBLIC_NAME) $(PLUGIN)
	bapbundle update -desc "$(PUBLIC_DESC)" $(PLUGIN)
	bapbundle install $(PLUGIN)
```

Build and install the plugin:

```
make
```

Save a project, e.g.:

```
bap /bin/true --project test.proj --update
```

Start the analysis REPL for that project:

```
bap analyze --project test.proj
```

List the installed commands:

```
bap> commands
```

You should see your `my.org:encoding` command in the list:

```
my.org:encoding                           prints the encoding of a given address
bap:subroutines                           prints all subroutines
bap:units                                 prints all units
bap:subroutine                            prints a subroutine
bap:instructions                          prints all instructions
bap:instruction                           prints the instruction semantics
```

List the compilation units:

```
bap> bap:units
```

It should print something like this:

```
file:/bin/true                          bap:amd64
```

List the instructions for the compilation unit:

```
bap> bap:instructions /bin/true
```

It should list all of the instruction labels:

```
...
/bin/true:0x17da
/bin/true:__libc_start_main
/bin/true:0x17d4
/bin/true:0x17cd
/bin/true:0x17c6
/bin/true:0x17bf
/bin/true:0x17be
...
```

Run your `encoding` command on, say, the `__libc_start_main` label:

```
bap> my.org:encoding /bin/true:__libc_start_main
```

It should print out the encoding, e.g.:

```
Encoding: bap:llvm-x86_64
```

Note that it did not print out the name of the label. This is because we ran the command without adding the `:show-name` flag.

Run it with the `:show-name` flag:

```
bap> my.org:encoding /bin/true:__libc_start_main :show-name
```

Now it prints the name of the label in addition to the encoding:

```
Encoding: bap:llvm-x86_64
Name: __libc_start_main
```

Try running it on a label that doesn't have a name, e.g.:

```
bap> my.org:encoding /bin/true:0x1d47 :show-name
```

This prints the encoding of the label, and provides no name as expected:

```
Encoding: bap:llvm-x86_64
Name: none
```

Exit the REPL:

```
bap> quit
```

Clean up:

```
make uninstall clean
```

D References

References

[1] D. Newcomb, "The Next Big OS War is in Your Dashboard," *Wired*, Dec 2012.

[2] S. McConnell, *Code Complete*. Microsoft Press, 2004.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 283–294, ACM, 2011.

[4] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), pp. 345–355, ACM, 2013.

[5] S. Joshi, S. Lahiri, and A. Lal, "Underspecified Harnesses and Interleaved Bugs," in *Principles of Programming Languages (POPL) 2012*, ACM SIGPLAN, January 2012.

[6] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Verified integrity properties for safe approximate program transformations," in *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[7] G. C. Necula, "Translation validation for an optimizing compiler," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, (New York, NY, USA), Association for Computing Machinery, 2000.

[8] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, (Berlin, Heidelberg), pp. 712–717, Springer-Verlag, 2012.

[9] DARPA, "Cyber grand challenge homepage." Available: https://www.cybergrandchallenge.com, 2016.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, (Berlin, Heidelberg), pp. 463–469, Springer-Verlag, 2011.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, May 2016.

[12] National Security Agency, "Ghidra." https://www.nsa.gov/resources/everyone/ghidra, 2019.

[13] Y. Shoshitaishvili, "angr's new year resolutions." Available: http://angr.io/blog/new_years_resolutions_2017/, Jan 2017.

[14] Draper, "CBAT: A Comparative Binary Analysis Tool." https://github.com/draperlaboratory/cbat_tools, Jan 2021.

[15] C. Casinghino, J. Paasch, C. Roux, J. Altidor, M. Dixon, and D. Jamner, "Using binary analysis frameworks: The case for BAP and angr," in *NASA Formal Methods '19*, 2019.

[16] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, (Piscataway, NJ, USA), pp. 353–364, IEEE Press, 2017.

[17] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[18] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pp. 845–860, USENIX Association, 2014.

[19] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 583–600, USENIX Association, 2016.

[20] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 611–620, ACM, 2009.

[21] P. P. F. Chan and C. Collberg, "A method to evaluate cfg comparison algorithms," in *2014 14th International Conference on Quality Software*, pp. 95–104, Oct 2014.

[22] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[23] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International Conference on Compiler Construction (CC)*, (Berlin, Heidelberg), pp. 5–23, Springer Berlin Heidelberg, 2004.

[24] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 23:1–23:84, Aug. 2010.

[25] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.

[26] R. Sen and Y. N. Srikant, “Executable analysis using abstract interpretation with circular linear progressions,” in *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE ’07, (Washington, DC, USA), pp. 39–48, IEEE Computer Society, 2007.

[27] L. Källberg, “Circular linear progressions in SWEET,” tech. rep., Mälardalen University, Embedded Systems, 2014.

[28] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, “Signedness-agnostic program analysis: Precise integer bounds for low-level code,” in *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, pp. 115–130, 2012.

[29] J. Lee, T. Avgerinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” in *Network and Distributed Systems Security Syposium (NDSS)*, Internet Society, 01 2011.

[30] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, Aug. 1975.

[31] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS’08/ETAPS’08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[32] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0,” *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014.

[33] P. Godefroid, “Micro execution,” in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 539–549, ACM, 2014.

[34] S. Dinesh, N. Burow, D. Xu, and M. Payer, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1497–1511, 2020.

[35] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.

[36] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[37] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of dram disturbance errors,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 361–372, 2014.

[38] C. Hawblitzel, S. K. Lahiri, K. Pawar, H. Hashmi, S. Gokbulut, L. Fernando, D. Detlefs, and S. Wadsworth, "Will you still compile me tomorrow? static cross-version compiler validation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, (New York, NY, USA), p. 191201, Association for Computing Machinery, 2013.

[39] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pp. 311–325, 2007.

[40] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[41] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, p. 772–781, IEEE Press, 2013.

[42] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, p. 448–458, IEEE Press, 2015.

[43] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 691–701, Association for Computing Machinery, 2016.

[44] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 593–604, Association for Computing Machinery, 2017.

[45] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, (New York, NY, USA), p. 532–543, Association for Computing Machinery, 2015.

[46] X.-B. D. Le, F. Thung, D. Lo, and C. L. Goues, "Overfitting in semantics-based automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), p. 163, Association for Computing Machinery, 2018.

[47] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 831–841, Association for Computing Machinery, 2017.

[48] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on*

Software Testing and Analysis, ISSTA 2017, (New York, NY, USA), p. 226–236, Association for Computing Machinery, 2017.

[49] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, “Identifying patch correctness in test-based program repair,” in *Proceedings of the 40th International Conference on Software Engineering*, ICSE ’18, (New York, NY, USA), p. 789–799, Association for Computing Machinery, 2018.

[50] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in *Proceedings of the 39th International Conference on Software Engineering*, ICSE ’17, p. 416–426, IEEE Press, 2017.

[51] Z. Huang, D. Lie, G. Tan, and T. Jaeger, “Using safety properties to generate vulnerability patches,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 539–554, 2019.

[52] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond tests: Program vulnerability repair via crash constraint extraction,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, feb 2021.

[53] X. Leroy, “Formal verification of a realistic compiler,” *Commun. ACM*, vol. 52, p. 107–115, jul 2009.

[54] A. Pnueli, M. Siegel, and E. Singerman, “Translation validation,” in *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS ’98, (Berlin, Heidelberg), p. 151–166, Springer-Verlag, 1998.

[55] G. C. Necula, “Translation validation for an optimizing compiler,” *SIGPLAN Not.*, vol. 35, p. 83–94, may 2000.

[56] N. Benton, “Simple relational correctness proofs for static analyses and program transformations,” *SIGPLAN Not.*, vol. 39, p. 14–25, jan 2004.

[57] J.-B. Tristan, P. Govereau, and G. Morrisett, “Evaluating value-graph translation validation for llvm,” *SIGPLAN Not.*, vol. 46, p. 295–305, jun 2011.

[58] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization,” in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’09, (New York, NY, USA), p. 264–276, Association for Computing Machinery, 2009.

[59] S. Gupta, A. Rose, and S. Bansal, “Counterexample-guided correlation algorithm for translation validation,” *Proc. ACM Program. Lang.*, vol. 4, nov 2020.

[60] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, “Alive2: Bounded translation validation for llvm,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 65–79, Association for Computing Machinery, 2021.

[61] M. Patrignani, A. Ahmed, and D. Clarke, “Formal approaches to secure compilation: A survey of fully abstract compilation and related work,” *ACM Comput. Surv.*, vol. 51, feb 2019.

[62] S. Lahiri, K. McMillan, R. Sharma, and C. Hawblitzel, “Differential assertian checking,” in *ESEC/FSE 2013: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 45–355, 2013.

[63] B. Beckert and M. Ulbrich, “Trends in relational program verification,” in *Principled Software Development*, pp. 41–58, Springer-Verlag, 2018.

[64] G. Barthe, J. Crespo, and C. Kunz, “Relational verification using product programs,” in *FM 2011: Formal Methods*, pp. 200–214.

[65] A. Zaks and A. Pnueli, “CoVaC: Compiler validation by program analysis of the cross-product,” in *FM 2008: Formal Methods*, pp. 35–51, 2008.

[66] V. D’Silva, M. Payer, and D. Song, “The correctness-security gap in compiler optimization,” in *2015 IEEE Security and Privacy Workshops*, pp. 73–87, 2015.

[67] C. Deng and K. S. Namjoshi, “Securing a compiler transformation,” *Form. Methods Syst. Des.*, vol. 53, p. 166–188, oct 2018.

[68] J. Guttag, J. Horning, and J. Wing, “Some notes on putting formal specifications to productive use,” *Science of Computer Programming*, vol. 2, pp. 53–68, 1982.

[69] D. Berry and J. Wing, “Specifying and prototyping: Some thoughts on why they are successful,” in *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, vol. 2 of *Lecture Notes in Computer Science 186*, pp. 117–128, Springer-Verlag, 1985.

[70] R. Nakajima, M. Honda, and H. Nakahara, “Hierarchical program specification and verification — a many-sorted logical approach,” *Acta Informatica*, vol. 14, pp. 135–155, 1980.

[71] J. Guttag and J. Horning, “Formal specification as a design tool,” in *POPL ’80: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 251–261, 1980.

[72] J. Wing, “A two-tiered approach to specifying programs,” Tech. Rep. Technical Report 299, MIT Laboratory for Computer Science, 1983.

[73] J. Wing, “Helping specifiers evaluate their specifications,” in *Proceedings of AFCET Second International Conference on Software Engineering, Nice, France, June 4-6, 1984*, pp. 179–189, 1985.

[74] “Using a graphical design tool for formal specification,” in *Proceedings 13th Annual Workshop of the Psychology of Programming Interest Group*, 2001.

[75] J. Goguen and R. Burstall, "Putting theories together to make specifications," in *IJCAI'77: Proceedings of the 5th International Joint Conference on Artificial Intelligence*, vol. 2, p. 1045–1058, 1977.

[76] J. Goguen and R. Burstall, "Institutions: Abstract model theory for specification and programming," *Journal of the ACM*, vol. 39, no. 1, p. 95–146, 1992.

[77] M. Abadi and L. Lamport, "Composing specifications," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 1, p. 73–132, 1993.

[78] J. Goguen, "Data, schema, ontology and logic integration," *Logic Journal of the IGPL*, vol. 13, no. 6, pp. 685–715, 2005.

[79] D. Sannella and A. Tarlecki, "Horizontal composability revisited," in *Algebra, Meaning, and Computation*, p. 296–316, 2006.

[80] D. Smith, "Composition by colimit and formal software development," in *Algebra, Meaning, and Computation*, p. 317–332, 2006.

[81] R. Burstall and J. Goguen, "The semantics of CLEAR, a specification language," in *Abstract Software Specifications* (D. Bjorner, ed.), vol. 86 of *LNCS*, p. 292–332, 1980.

[82] J. Goguen and J. Tardo, "An introduction to OBJ: A language for writing and testing software specifications," in *Specification of Reliable Software* (M. Zelkowitz, ed.), p. 170–189, 1979.

[83] M. Bidoit and P. Mosses, "Casl user manual," in *CASL User Manual* (M. Bidoit and P. Mosses, eds.), vol. 2900 of *LNCS*, p. 193–201, 2004.

[84] S. Clérici, R. Jiménez, and F. Orejas, "Semantic constructions in the specification language glider," in *Recent Trends in Data Type Specification, ADT 1992, COMPASS 1992*, p. 144–157, 1992.

[85] J. Goguen and R. Burstall, "Cat, a system for the structured elaboration of correct programs from structured specifications," Tech. Rep. Technical Report CSL-118, Computer Science Laboratory, SRI International, 1980.

[86] C. Jones, *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[87] L. Blane and A. Goldberg, "DTRE – a semi-automatic transformation system," in *Constructing Programs from Specifications* (B. Moller, ed.), pp. 165–204, North Holland, 1991.

[88] D. Smith, "KIDS – a semi-automatic program development system," *IEEE Transactions on Software Engineering, Special Issue on Formal Methods of Software Engineering*, vol. 16, no. 9, pp. 1024–1043, 1990.

[89] Y. Srinivas and R. Jullig, "Specware: Formal support for composing software," in *Mathematics of Program Construction, Third International Conference, MPC '95* (B. Moller, ed.), pp. 399–422, Springer, 1995.

[90] M. Barnett, M. Fähndrich, K. Leino, P. Müller, W. Schulte, and H. Venter, “Specification and verification: The spec# experience,” *Communications of the ACM*, vol. 54, no. 6, p. 81–91, 2011.

[91] D. Steinhofel, *Abstract Execution: Automatically Proving Infinitely Many Programs*. PhD thesis, Technische Universitat Darmstadt, 2020.

[92] S. Lerner, T. Millstein, and C. Chambers, “Automatically proving the correctness of compiler optimizations,” *SIGPLAN Not.*, vol. 38, p. 220–231, may 2003.

[93] S. Lerner, T. Millstein, E. Rice, and C. Chambers, “Automated soundness proofs for dataflow analyses and transformations via local rules,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’05, (New York, NY, USA), p. 364–377, Association for Computing Machinery, 2005.

[94] J. Nimmer and M. Ernst, “Automatic generation of program specifications,” *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 229–239, 2002.

[95] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, “Synthesis of interface specifications for java classes,” *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 98–109, 2005.

[96] M. Ramanathan, A. Grama, and S. Jagannathan, “Static specification inference using predicate mining,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 123–134, 2007.

[97] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, “Static specification mining using automata-based abstractions,” *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 651–666, 2008.

[98] B. Livshits, A. Nori, S. Rajamani, and A. Banerjee, “Merlin: Specification inference for explicit information flow problems,” *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 75–86, 2009.

[99] G. Reger, “An overview of specification inference,” tech. rep., University of Manchester, 2011.

[100] K. Hoder and N. Bjorner, “Generalized property directed reachability,” in *SAT 2012: Theory and Applications of Satisfiability Testing*, p. 157–171, 2012.

[101] O. Bastani, S. Anand, and A. Aiken, “Specification inference using context-free language reachability,” *ACM SIGPLAN Notices*, vol. 50, no. 1, p. 553–566, 2015.

[102] Y. Li, A. Turrini, M. Vardi, and L. Zhang, “Synthesizing good-enough strategies for LTL_f specifications,” in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21)*, pp. 4144–4151, 2021.

[103] E. Kamburjan and N. Wasser, “The right kind of non-determinism: Using concurrency to verify c programs with underspecified semantics,” *Electronic Proceedings in Theoretical Computer Science*, vol. 365, pp. 1–16, 2022.

[104] K. Ali and O. Lhoták, “AVERROES: Whole-program analysis without the whole program,” in *ECOOP 2013: ECOOP 2013 – Object-Oriented Programming*, pp. 378–400, 2013.

[105] H. Zhu, T. Dillig, and I. Dillig, “Automated inference of library specifications for source-sink property verification,” in *APLAS 2013: Programming Languages and Systems*, p. 290–306, 2013.

[106] C. Calcagno, D. Distefano, P. O’Hearn, and H. Yang, “Compositional shape analysis by means of bi-abduction,” *Journal of the ACM*, vol. 58, no. 6, pp. 1–66, 2011.

[107] I. Dillig, T. Dillig, and A. Aiken, “Automated error diagnosis using abductive inference,” *ACM SIGPLAN Notices*, vol. 47, no. 6, p. 181–192, 2012.

[108] M. Seghir and D. Kroening, “Counterexample-guided precondition inference,” in *ESOP 2013: Programming Languages and Systems*, pp. 451–471, 2013.

[109] A. Das, S. Lahiri, A. Lal, and Y. Li, “Angelic verification: Precise verification modulo unknowns,” in *Computer Aided Verification: CAV 2015*, pp. 324–342, 2015.

[110] A. Albarghouthi, I. Dillig, and A. Gurfinkel, “Maximal specification synthesis,” in *POPL ’16: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 789–801, 2016.

[111] S. Prabhu, G. Fedyukovich, K. Madhukar, and D. D’Souza, “Specification synthesis with constrained horn clauses,” in *PLDI ’21: Proceedings of the 2021 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021.

[112] B. Alpern and F. Schneider, “Defining liveness,” *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.

[113] B. Alpern and F. Schneider, “Recognizing safety and liveness,” *Distributed Computing*, vol. 2, pp. 117–126, 1987.

[114] M. Clarkson and F. Schneider, “Hyperproperties,” *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[115] G. Barthe, P. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” in *Proceedings, 17th IEEE Computer Security Foundations Workshop, 30 June 2004*, pp. 100–114, 2004.

[116] S. Agrawal and B. Bonakdarpour, “Runtime verification of k-safety hyperproperties in HyperLTL,” in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pp. 239–252, 2016.

[117] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, “Property directed self-composition,” in *Computer Aided Verification: CAV 2019* (I. Dillig and S. Tasiran, eds.), pp. 161–179, 2019.

[118] H. Unno, T. Terauchi, and E. Koskinen, “Constraint-based relational verification,” in *Computer Aided Verification: CAV 2021*, p. 742–766, 2021.

[119] L. Lamport and F. Schneider, "Verifying hyperproperties in TLA," in *Proceedings of the 34th IEEE Computer Security Foundations Symposium, CSF 2021*, pp. 1–16, 2021.

[120] Y. Srinivas, "A sheaf-theoretic approach to pattern matching and related problems," *Theoretical Computer Science*, vol. 112, pp. 53–97, 1993.

[121] R. Short, A. Hylton, R. Cardona, R. Green, G. Bainbridge, M. Moy, and J. Cleveland, "Towards sheaf theoretic analyses for delay tolerant networking," in *2021 IEEE Aerospace Conference (50100)*, pp. 1–9, 2021.

[122] A. Mazurkiewicz, "Basic notions of trace theory," in *REX 1988: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pp. 285–363, 1988.

[123] M. Smyth, "Topology," in *Handbook of Logic in Computer Science* (S. Abramsky, D. Gabbay, and T. Maibaum, eds.), vol. 1, pp. 641–761, Clarendon Press, 1992.

[124] M. Kwiatkowska, "On topological characterization of behavioural properties," in *Topology and Category Theory in Computer Science* (G. Reed, A. Roscoe, and R. Wachter, eds.), pp. 153–177, Clarendon Press, 1991.