CLOSURE Toolchain User Manual for C++ Language

Prototype Release Version (v1.0)

Peraton Labs

May 8, 2024

Contents

1 CLOSURE Toolchain Overview

1.1 What is CLOSURE?

DARPA's Guaranteed Architecture for Physical Systems (GAPS) is a research program that addresses software and hardware for compartmentalized applications where multiple parties with strong physical isolation of their computational environment, have specific constraints on data sharing (possibly with redaction requirements) with other parties, and any data exchange between the parties is mediated through a guard that enforces the security requirements.

Peraton Labs' Cross-domain Language extensions for Optimal SecUre Refactoring and Execution (CLOSURE) project is building a toolchain to support the development, refactoring, and correct-by-construction partitioning of applications and configuration of the guards. Using the CLOSURE approach and toolchain, developers will express security intent through annotations applied to the program, which drive the program analysis, partitioning, and code auto-generation required by a GAPS application.

Problem: The machinery required to verifiable and securely establish communication between cross-domain systems (CDS) without jeopardizing data spillage is too complex to implement for many software platforms where such communication would otherwise be desired. To regulate data exchanges between domains, network architects rely on several risk mitigation strategies including human fusion of data, diodes, and hypervisors which are insufficient for future commercial and government needs as they are high overhead, customized to specific setups, prone to misconfiguration, and vulnerable to software/hardware security flaws. To streamline the design, development, and deployment of provably secure CDSs, new hardware and software co-design tools are needed to more effectively build cross-domain support directly into applications and associated hardware early in the development lifecycle.

Solution: Peraton Labs is developing CLOSURE (Cross-domain Language-extensions for Optimal SecUre Refactoring and Execution) to address the challenges associated with building cross-domain applications in software. CLOSURE extends existing programming languages by enabling developers the ability to express security intent through overlay annotations and security policies such that an application can be compiled to separable binaries for concurrent execution on physically isolated platforms.

The CLOSURE compiler toolchain interprets annotation directives and performs program analysis of the annotated program and produces a correct-by-construction partition if feasible. CLOSURE automatically generates and inserts serialization, marshaling,

and remote-procedure call code for cross-domain interactions between the program partitions.

1.2 The C++ Challenge

As stated in [1], "writing a tool capable of assuring a property that falls out of the scope of the standard type checking performed by a compiler is a task comparable to writing the compiler itself." This is especially true with respect to C++ where an extensive list of object-oriented features makes it difficult to conduct the required program analysis to track the control and data dependencies incurred by multiple inheritance, polymorphism, templates, exceptions, overrides, etc. Unlike C and Java for which extensive research and tools for formal dependency modeling exists [2] [3], the set of such tools for C++ are sparse. While a large subset of the C language has been formalized [4] [5], formalisms for analysis of C++ programs [1] [6] handle a limited subset of the C++ language. For example, cpp2v [7] does not support templates, virtual inheritance, floating point numbers, and many other critical features. Other approaches to formalize C++ for analysis exhibit other limitations that fall short of their applicability to real-world C++ programs. The alternative approach of defining type extensions and rewriting the program based on those extensions so that the type checker can check for properties is not pragmatic due to the learning curve, lack of tooling support for non-standard extensions, and cost of rewriting legacy programs in the new dialect. Much of the CLOSURE analyses occur at the LLVM-IR level, and that generated by C++ is far more difficult to understand without proper tools (names are mangled, classes and objects are refactored, etc.). Further complicating the effort is the lack of reflection in C++ (afforded to us in Java) for explicitly describing the types of fields and structure of opaque objects as they are marshalled and serialized for streaming communications across guard devices.

In the final months of GAPS Phase 3, we turned our focus to that of a preliminary CLOSURE C++ toolchain proof-of-concept design. In the following sections, we document our findings, initial approach, and early implementation. Our proof-of-concept toolchain builds around a modern C++ frontend AST, though we stick to programs of C++98 complexity, incrementally adding support for modern features as they become of interest. Several of the toolchain steps are conducted manually with plans to automate them in future research.

Our approach follows the same high-level workflow that has proven successful for us in C and Java:

- Build a program model representation that consists of information extracted from the AST and points-to analysis
- Extend application of CLOSURE Language Extensions (CLE) to Object-Oriented constructs (building upon our work for a subset of Java) to cover a subset of C++ adequate to meet the transition partner code
- Extend the formal constraint model for C++ constructs

- Code generation to manage cross-domain object instantiation, destruction, and methods

1.3 Architecture

The CLOSURE architecture for C++ follows the same workflow that has proven successful for C and Java with language-specific differences in annotation, modeling, constraint, and code-generation steps as illustrated in Figure 1:

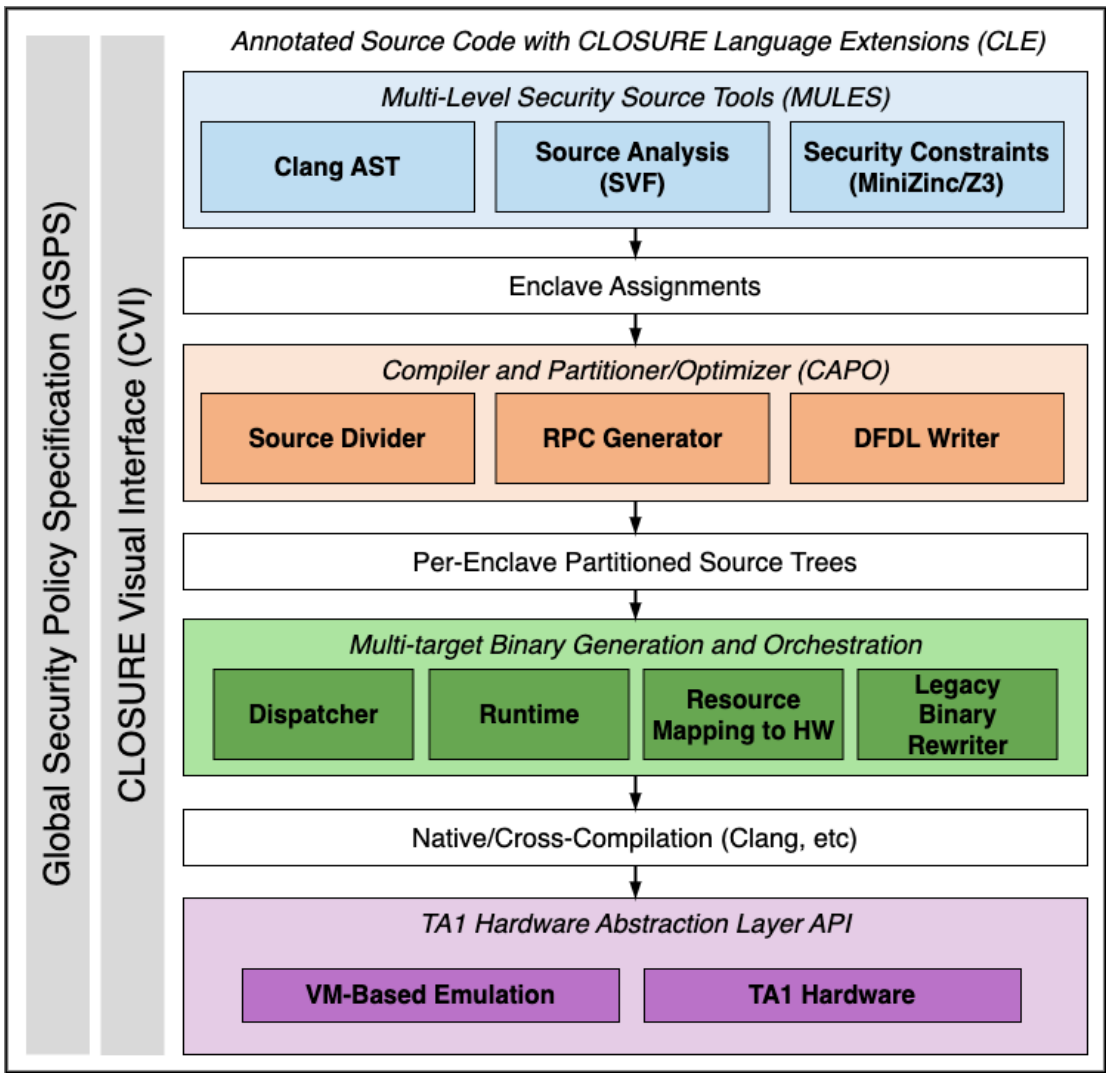

Figure 1: Figure: CLOSURE Architecture

The key sub-modules of the C++ toolchain include:

- **CVI** (CLOSURE Visual Interface): The editor built on top of VSCode [8]. Provides the IDE and co-design tools for cross-domain software development.
- **MULES** (Multi-Level Security Source Tools): The tools used to analyze the annotated source code to produce enclave assignments. The Clang API, along with SVF for pointer analysis, is used to generate the program dependency model. Constraint solvers (MiniZinc and Z3) take as input the program dependency model and security policy constraints to produce enclave assignments.
- **CAPO** (Conflict Analyzer Partition Optimizer): The constraint-based conflict analysis tools used to determine if a partitioning is feasible. Additional tools in CAPO auto-generate the additional logic needed to make a program cross-domain enabled (i.e., data type marshalling/serialization, RPCs for cross-domain data exchange, interfacing to the device drivers of cross-domain guards, DFDL [9] and rule generation, among others). CAPO also includes a post-partitioning verifier which checks that the partitioned program including auto-generated code is functionally equivalent to complies with developer security annotations.
- **MBIG** (Multi-Target Binary Generation): Currently, for C++, MBIG is limited to x86 but could be extended to match the features of the C toolchain.
- **HAL** (Hardware-Abstraction-layer): Abstracts hardware APIs of different cross-domain hardware devices. Currently, for C++, HAL has been tested in the minimal, localhost-mode using a single machine, though could easily extend to the other hardware supported in C toolchain.

Notable differences between the C++ proof-of-concept toolchain and its C/Java counterparts include:

- Annotation of fields, methods (including constructors/destructors), and class definitions
- Leveraging of the Clang AST to facilitate annotation and program analysis
- A translation layer between LLVM registers and C++ declarations
- Object-oriented constraints
- RPC/Serialization code conducive to C++ semantics

Further details of the workflow are presented in following section.

1.4 Workflow

Like the CLOSURE architecture, the CLOSURE workflow for C++ mimics the workflow that has proven successful for C and Java with language-specific differences in tooling. The Clang API is used, along with SVF for pointer analysis, to interpret annotations and generate the program dependency model. Along with MiniZinc, Z3 has been added as a second constraint solver and is useful for diagnostics. The workflow is illustrated in Figure 2:

The workflow is defined for a single source cpp follows:

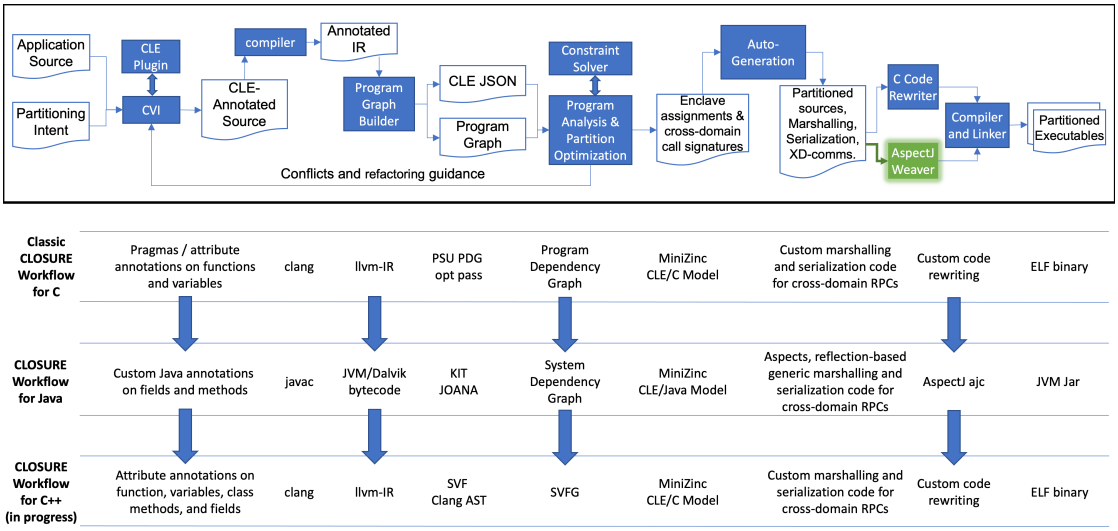

Figure 2: Figure: CLOSURE Workflow

1. We first use our clang-plugin on the source file, which outputs the program graph and CLE-json. The output graph is a pair of `.csv` files `nodes.csv` and `edges.csv` that contains the nodes and edges of the graph.

2. We compile the source file to LLVM IR using `clang` and use our SVF-derived tool called `DumpPTG` to extract points-to relations between LLVM registers for each generated LLVM function from the C++ source code. The output is a Points-To Graph, a pair of csv files `svf_nodes.csv` and `svf_edges.csv` where the nodes are memory locations corresponding to LLVM registers (e.g., `%4` or `@a`) and the edges are points-to relations.

3. We then use our translation layer, `extract-declares` to translate each of the points-to relations outputted the previous step back to declarations at the program graph/C++ source level. By default, each LLVM register which corresponds to a declaration is annotated with a special debug call `llvm.dbg.declare` relating the two, so that information is used to create a map between certain LLVM registers and their corresponding C++ declarations
4. Once `extract-declares` program is run, we can use another small python program `points_to_edges.py` to translate the points-to edges in the Points-To Graph to edges between declarations in the program graph. These edges are added as new `PointsTo` edges into `edges.csv`
5. A python program `solver/main.py` takes in as input the modified program with points to edges and the cle-json and outputs a constraint satisfaction problem for Minizinc or optionally Z3.
6. The output of the previous step is used as input for an SMT solver (Z3/Minizinc), which either gives us a valid assignment or an unsat core, the smallest set of

unsatisfiable constraints. An unsat-core helps the user understand which rules were violated in our generated constraint model and potentially how to correct the program annotations

As of this prototype release, CLOSURE's C++ toolchain supports up to the program division step for a limited subset of C++ language constructs, with automated generation of RPCs, DFDLs, and HAL configuration left for future research.

1.5 Document Roadmap

In the rest of this document, we first present a quick start guide followed by a detailed usage of the toolchain components. For each component, we describe what it does, provide some insight into how it works, discuss inputs and outputs and provide invocation syntax for usage. We conclude with a discussion of the limitations of our current toolchain and a roadmap for future work. We provide samples of significant input and output files in the appendices and provide a list of bibliographic references at the end.

2 Installation and Quick Start For CLOSURE C++ Toolchain

The quickest way to get started with the C++ toolchain is using the provided Dockerfile to build an image after cloning the repository.

```
git clone https://github.com/gaps-closure/cpp-closure
cd cpp-closure
docker build -t gapsclosure/cpp-closure:develop .
```

Afterwards, one can use the Remote - Containers extension with VSCode to work within the CLE environment for C++. Alternatively, a docker image can be run via the command line:

```
docker run -it gapsclosure/cpp-closure:develop
```

Within the docker container, there will be a number of utilities, libraries and python scripts under `/opt/closure`. The provided binaries should be part of the user's `$PATH` and should be accessible from the command line.

3 Detailed Usage and Reference Manual

3.1 Annotations

3.1.1 Annotating the C++ source

The CLE annotations in a C++ program look different from that of C programs. Every declaration can be annotated using `__attribute__((cle_annotate("LABEL_NAME")))` where `"LABEL_NAME"` is a reference to a CLE definition. A CLE definition is defined using the `#pragma cle def <CLE_JSON>`, exactly the same as it is in C.

When annotating a declaration, usually the `__attribute__((cle_annotate("ORANGE")))` can be positioned in front of the declaration, or on the line above, like so:

```
__attribute__((cle_annotate("ORANGE")))
Foo foo = Foo();
```

The only exception to this is `class` annotations, where the attribute statement must be placed after the `class` keyword, like so:

```
class __attribute__((cle_annotate("PURPLE_SHAREABLE"))) Bar
{
...
};
```

Below are sample CLE definitions for `ORANGE` and `PURPLE_SHAREABLE` for reference:

```
#pragma cle def ORANGE {"level":"orange"}

#pragma cle def PURPLE_SHAREABLE {"level":"purple",\
  "cdf": [\
    {"remotelevel":"orange", \
     "direction": "egress", \
     "guarddirective": { "operation": "allow"}}\
  ] }
```

3.1.2 Annotation flow

An annotation will have its CLE label appear in the exported program graph from the clang-plugin. For example, the following annotation `"ORANGE"` on the construction of `Foo()`

```
__attribute__((cle_annotate("ORANGE")))
Foo foo = Foo();
```

will appear in the `nodes.csv` file, like so:

```
1,Decl.Var,"foo",ORANGE,4,,4,,Foo.cpp,245,301
...
```

Additionally, the clang-plugin will output the CLE definitions in a `collated.json` file. For example, our definition for `ORANGE` will appear in the json in the following manner:

```
[{"cle-label": "PURPLE","cle-json":{"level":"purple"}},
...
]
```

The annotation labels and definitions are eventually used to create the constraint solver instance, either in Minizinc or Z3.

3.2 Modeling a C++ Program as a Program Graph

The following is table of node and edges types, a description of their purpose and where that information came from (either the Clang AST or SVF).

3.2.1 Node Types

Node Name	Node Description
Decl.Var	All variable declarations.
Decl.Function	A static global variable inside a function
Decl.Record	A class, struct or union definition.
Decl.Field	A record field.
Decl.Method	A record method. Includes operator overloads
Decl.Param	Formal parameter of a function/method. Contains information about parameter index
Decl.Constructor	A constructor for a class/struct
Decl.Destructor	A destructor for a class/struct
Stmt.Decl	A statement which also has a declaration
Stmt.Call	An clang statement that represents a call to a function, method, constructor or destructor.
Stmt.Compound	An clang statement that only has other statements as children.
Stmt.Ref	An clang statement which holds a reference to a decl
Stmt.Field	An clang statement which is a field accessor
Stmt.This	An clang statement which refers to ‘this’
Stmt.Return	A return within a given function
Stmt.Other	An clang statement that does not transfer control to another function.

3.2.2 Edge Types

Edge Name	Edge Description
Struct.Field	Connects a record to its field
Struct.Method	Connects a record to its method
Struct.Constructor	Connects a record to its constructor
Struct.Destructor	Connects a record to its destructor
Struct.Inherit	Connects two records by inheritance relation. Type of inheritance given by a property
Struct.Param	Connects a function-like object to its parameters
Struct.Child	When a statement has a child not described by the above. Can be pruned out of the graph
Control.Entry	Connects a function-like object to it's body
Control.FunctionInvocation	A direct call invocation to a C-style function
Control.MethodInvocation	A method call
Control.ConstructorInvocation	A constructor call
Control.DestructorInvocation	A call to a destructor
Data.DefUse	A def-use relation between statements
Data.ArgPass	An argument pass to a call instruction. Destination has a paramater index
Data.Return	Connects a call to its resulting value at the call site
Data.Object	Connects a method call/field to the object being called upon (e.g. f as in f.foo())
Data.FieldAccess	A field access
Data.InstanceOf	Connects a 'this' instance to a class definition
Data.Decl	Connects a decl statment to a decl
Data.PointsTo	A points-to relation

Edge Name	Source Type	Destination Type
Struct.Field	Decl.Record	Decl.Field
Struct.Method	Decl.Record	Decl.Method
Struct.Constructor	Decl.Record	Decl.Constructor
Struct.Destructor	Decl.Record	Decl.Destructor
Struct.Inherit	Decl.Record	Decl.Record
Struct.Param	Decl	Decl.Param
Struct.Child	Stmt	Stmt
Control.Entry	Decl	Stmt
Control.FunctionInvocation	Stmt.Call	Decl.Function
Control.MethodInvocation	Stmt.Call	Decl.Method
Control.ConstructorInvocation	Stmt.Call	Decl.Constructor
Control.DestructorInvocation	Stmt.Call	Decl.Destructor

Edge Name	Source Type	Destination Type
Data.DefUse	Stmt.Decl	Stmt.Ref
Data.ArgPass	Stmt.Call	Stmt
Data.Return	Stmt.Call	Stmt
Data.Object	Stmt.Call/Stmt.Field	Stmt
Data.FieldAccess	Stmt	Decl.Field
Data.InstanceOf	Stmt.This	Decl.Record
Data.Decl	Stmt.Decl	Decl
Data.PointsTo (from SVF+AST)	Decl	Decl

3.3 Conflict analyzer based on MiniZinc Constraint Solver

See the C documentation [10] for description, purpose, and usage of the conflict analyzer. For the purposes of this documentation, we focus on the C++ CLOSURE constraint model in the text below.

3.3.1 Detailed MiniZinc Constraint Model

In this section, we present an informal statement of constraints to be enforced by our conflict analyzer. We then present the main constraints coded in MiniZinc used by our model to achieve these constraints. More information about MiniZinc including its usage and syntax can be found in [11].

In the model below, the `nodeEnclave` decision variable stores the enclave assignment for each node, the `taint` decision variable stores the label assignment for each node, and the `xdedge` decision variable stores whether a given edge is in the enclave cut (i.e., the source and destination nodes of the edge are in different enclaves). Several other auxiliary decision variables are used in the constraint model to express the constraints or for efficient compilation.

3.3.1.1 Structural Constraints on Node Enclave/Level Every node must be assigned to a valid enclave. Functions and classes must be assigned to the same enclave as their containing node. The level of the label (taint) of every node must match the level of the enclave the node is assigned to.

```
constraint :: "NodeHasEnclave"            forall (n in NodeIdx) (nodeEnclave[n]
!= nullEnclave);
constraint :: "NodeEnclaveIsFunEnclave"   forall (n in NodeIdx)
((hasFunction[n] != 0) -> (nodeEnclave[n] == nodeEnclave[hasFunction[n]]));
constraint :: "NodeEnclaveIsClassEnclave" forall (n in NodeIdx) ((hasClass[n]
!= 0) -> (nodeEnclave[n] == nodeEnclave[hasClass[n]]));
constraint :: "NodeLevelAtEnclaveLevel"   forall (n in NodeIdx)
(hasLabelLevel[taint[n]] == hasEnclaveLevel[nodeEnclave[n]]);
```

3.3.1.2 Classifying Annotations

Only functions, methods, and constructors can be assigned a function annotation. Function annotations can only be made by the user.

```
constraint :: "FnAnnotationForFnOnly"
    forall (n in NodeIdx)          (isFunctionAnnotation[taint[n]] ->
isFunction(n));
constraint :: "FnAnnotationByUserOnly"
    forall (n in FunOrMethodOrCon) (isFunctionAnnotation[taint[n]] ->
userAnnotatedNode[n]);
constraint :: "annotationOnFunctionIsFunAnnotation"
    forall (n in FunOrMethodOrCon) (userAnnotatedNode[n] ->
isFunctionAnnotation[taint[n]]);
```

Annotations on classes and their fields are node annotations, not function annotations.

```
constraint :: "annotationOnClassIsNodeAnnotation"
    forall (n in Decl_Record) (userAnnotatedNode[n] -> not
(isFunctionAnnotation[taint[n]]));
constraint :: "annotationOnFieldIsNodeAnnotation"
    forall (n in Decl_Field)  (userAnnotatedNode[n] -> not
(isFunctionAnnotation[taint[n]]));
```

3.3.1.3 Structural Taint Relationships Between Nodes

All nodes in an un-annotated function must have the taint of the function. All nodes in an un-annotated class must have the taint of the class. All nodes in an un-annotated class must be un-annotated and have the taint of the class.

```
constraint :: "UnannotatedFunContentTaintMatch"
    forall (n in NodeIdx where hasFunction[n] != 0) ((not
userAnnotatedNode[hasFunction[n]]) -> (taint[n] == ftaint(n)));

constraint :: "UnannotatedClassTaintsMatch"
    forall (n in NodeIdx where hasClass[n] != 0)
        ((not userAnnotatedNode[hasClass[n]]) -> (taint[n] ==
taint[hasClass[n]]));

constraint :: "noAnnotatedDataForUnannotatedClass"
    forall (n in NodeIdx where hasClass[n] != 0)
        ((not userAnnotatedNode[hasClass[n]]) -> (taint[n] ==
taint[hasClass[n]]));

constraint :: "noAnnotatedDataForUnannotatedClass"
    forall (n in NodeIdx where hasClass[n] != 0)
        ((not userAnnotatedNode[hasClass[n]]) -> (not userAnnotatedNode[n]));
```

Un-annotated constructors, destructors, and methods must have the taint of the class.

```
constraint :: "unannotatedConstructorGetsClassTaint"
    forall (n in Decl_Constructor)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));

constraint :: "unannotatedDestructorGetsClassTaint"
    forall (n in Decl_Destructor)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));

constraint :: "unannotatedMethodGetsClassTaint"
    forall (n in Decl_Method)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));
```

Classes connected by an inheritance relationship share the same taint.

```
constraint :: "inheritTaint"
    forall (e in Record_Inherit) (esTaint(e) == edTaint(e));
```

All nodes in an annotated function must have a taint in the ARCtaints.

```
constraint :: "AnnotatedFunContentCoercible"
    forall (n in NodeIdx where (hasFunction[n] != 0) /\ (not isFunction(n)))
        (userAnnotatedNode[hasFunction[n]] -> isInArctaint(ftaint(n), taint[n],
hasLabelLevel[taint[n]]));
```

Annotated constructors (which are only valid in annotated classes) must have the class taint as their sole rettaint in every CDF.

```
constraint :: "annotatedConstructorReturnsClassTaint"
    forall (n in Decl_Constructor)
        (userAnnotatedNode[n] ->
            (forall (lvl in Level)
                ((cdfForRemoteLevel[taint[n], lvl] != nullCdf) ->
hasRettaints[cdfForRemoteLevel[taint[n], lvl], taint[hasClass[n]]])));
```

Any function whose address is taken in the program cannot have a function annotation.

```
constraint :: "FunctionPtrSinglyTainted"
    forall (e in Data_PointsTo) (isFunction(hasDest[e]) -> not
userAnnotatedNode[hasDest[e]]);
```

3.3.1.4 Constraints on the Cross-Domain Control Flow The control flow can never leave an enclave, unless it is done through an approved cross-domain call, as expressed in the following constraints. The only control edges allowed in the cross-domain cut are either call invocations or returns. For any call invocation edge in the cut, the function annotation of the function entry being called must have a CDF that allows (with or without redaction) the level of the label assigned to the callsite. The label assigned to

the callsite must have a node annotation with a CDF that allows the data to be shared with the level of the (taint of the) function entry being called.

```
constraint :: "NonCallRetControlEnclaveSafe"
    forall (e in Control_EnclaveSafe) (xdedge(e) == false);
constraint :: "XDCallBlest"
    forall (e in Control_Invocation) (xdedge(e) ->
userAnnotatedNode[hasDest[e]]);
constraint :: "XDCallAllowed"
    forall (e in Control_Invocation) (xdedge(e) -> allowOrRedact(edFunCdf(e)));
constraint :: "XDReturnAllowed"
    forall (e in Control_Return) (xdedge(e) -> allowOrRedact(esFunCdf(e)));
```

Note: The conflict analyzer is working with the annotated unpartitioned code and not the fully partitioned code which will includes autogenerated code. The actual cut in the partitioned code with autogenerated code to handle cross-domain communications will be between the cross-domain send and receive functions that are several steps removed from the cut in the `xdedge` variable at this stage of analysis. The autogenerated code will apply annotations to cross-domain data annotations that contain GAPS tags, and they will have a different label. So we cannot check whether the label of the arguments passed from the caller matches the argument taints allowed by the called function, or if the return taints match the value to which the return value is assigned. A downstream verification tool will check this.

3.3.1.5 Constraints on the Cross-Domain Data Flow Data can only leave an enclave through parameters or return of valid cross-domain call invocations, as expressed in the following constraints.

Cross-domain returns must be allowed by the CDF, but the taints need not match (can't enforce taint matching as the level must change). Cross-domain Points-To edges must be allowed by the CDF, but the taints need not match.

All children of returns must have the same taint as the return statement. Return statements in annotated functions must have a taint allowed by the CDF.

Note: The reason cross domain points-to edges are not disallowed completely is because passing arrays necessitates cross-domain points-to edges.

```
constraint :: "EnclaveSafeDataEdges"
    forall (e in Data_EnclaveSafe) (xdedge(e) == false);

constraint :: "ReturnChildMatchesReturn"
    forall (n in Stmt_Return) (
        forall (c in NodeIdx) (ancestor(n, c) -> taint[n] == taint[c])
    );

constraint :: "ReturnStmtMatchesRettaint"
    forall (n in Stmt_Return where annotFun(n)) (
```

```
        hasRettaints[funCdf(n), taint[n]]
    );

constraint :: "XDPointsToAllowed"
    forall (e in Data_PointsTo)
        (xdedge(e) -> (allowOrRedact(cdfForRemoteLevel[edTaint(e),
hasLabelLevel[esTaint(e)]]) /\ not isFunction(hasDest[e])));
```

3.3.1.6 Taint Propagation While the constraints on the control dependency and data dependency that we discussed governed data sharing at the cross-domain cut, we still need to perform taint checking to ensure that data annotated with different labels inside each enclave are managed correctly and only when the mixing of the taints is explicitly allowed by the user.

Labels can be coerced (i.e., nodes of a given PDG edge can be permitted to have different label assignments) inside an enclave only through user annotated functions.

Any data dependency or parameter edge that is intra-enclave (not in the cross-domain cut) and with different CLE label taints assigned to the source and destination nodes must be coerced (through an annotated function).

One may wonder whether a similar constraint must be added for control dependency edges at the entry block for completeness. Such a constraint is not necessary given our inclusion of the `UnannotatedFunContentTaintMatch` and `AnnotatedFunContent-Coercible` constraints discussed earlier. However, pointer dependencies are captured by a chain of points-to edges which may be intra-function edges. Therefore, we restrict intra-function points-to edges to have the same taint, even in annotated functions.

```
predicate intraFunEdge(DataEdge: e) =
    (hasFunction[hasSource[e]] != 0 /\ hasFunction[hasDest[e]] != 0 /\
     hasFunction[hasSource[e]] == hasFunction[hasDest[e]]);
constraint :: "intraFunPointsToTaintsMatch"
    forall (e in Data_PointsTo) (intraFunEdge(e) -> esTaint(e) == edTaint(e));
```

For ANY data edge between two function-external nodes, the taints must match.

```
predicate globalGlobalEdge(DataEdge: e) = (isGlobal[hasSource[e]] /\
isGlobal[hasDest[e]]);
constraint :: "externExternDataEdgeTaintsMatch"
    forall (e in DataEdge) (globalGlobalEdge(e) -> (esTaint(e) == edTaint(e)));
```

For ANY data edge between data within a function and a function-external node, the taints must match.

```
predicate srcFunExternEdge(DataEdge: e)  = (hasFunction[hasSource[e]] != 0 /\
hasFunction[hasDest[e]]   == 0);
predicate destFunExternEdge(DataEdge: e) = (hasFunction[hasDest[e]]   != 0 /\
hasFunction[hasSource[e]] == 0);
```

```
predicate funExternEdge(DataEdge: e)     = (srcFunExternEdge(e) \/
destFunExternEdge(e));
constraint :: "externDataEdgeTaintsMatch"
    forall (e in DataEdge) (funExternEdge(e) -> esTaint(e) == edTaint(e));
```

This leaves taint propagation for inter-function data edges. For non-XD return edges from a callee function to a callsite, if the callee (source) function is un-annotated, the taints must match and if the callee (source) function is annotated, the taint on the dest node must be in the callee's rettaints.

```
constraint :: "returnFromUnannotatedTaintsMatch"
    forall (e in Control_Invocation where not destAnnotFun(e))
        (taintsAgree(esTaint(e), edTaint(e)));
constraint :: "returnInRettaints"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (hasRettaints[edFunCdf(e), esTaint(e)]);
constraint :: "callsiteTaintMatchesAllReturns"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (forall (r in Stmt_Return where hasFunction[r] == hasDest[e])
            (taintsAgree(taint[r], esTaint(e))));
```

For non-XD argument passing edges, if the destination function is un-annotated, the taint of the argument must match the taint of the destination function and if the destination function is annotated, the taint of the argument must be in the argtaints of the function at that parameter index. All children of anything passed as an argument must have the same taint as the argument node.

```
constraint :: "argumentToUnannotatedTaintsMatch"
    forall (e in Control_Invocation where not destAnnotFun(e))
        (forall (arg_e in Data_ArgPass where hasSource[arg_e] == hasSource[e])
            (edTaint(arg_e) == edTaint(e)));
constraint :: "argumentInArgtaints"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (forall (arg_e in Data_ArgPass where hasSource[arg_e] == hasSource[e])
            (hasArgtaints[edFunCdf(e), hasParamIdx[hasDest[arg_e]],
taint[hasDest[arg_e]]]));
constraint :: "argumentChildren"
    forall (e in Data_ArgPass) (
        forall(n in NodeIdx where ancestor(hasDest[e], n))
            (taint[n] == edTaint(e))
    );
```

Inter-function points-to edges should always have the same taint.

```
predicate interFunEdge(DataEdge: e) =
    (hasFunction[hasSource[e]] != 0 /\ hasFunction[hasDest[e]] != 0 /\
     hasFunction[hasSource[e]] != hasFunction[hasDest[e]]);
constraint :: "interFunPointsToTaintsMatch"
```

```
    forall (e in Data_PointsTo)
        ((not xdedge(e) /\ interFunEdge(e)) -> esTaint(e) == edTaint(e));
```

Lastly, a node has the ALL label if and only if it is contained in an unpinned class. A class is pinned if: the class is annotated, anything in the class is annotated, or it depends on anything external to the class.

```
constraint :: "definitionForALL"
    forall (n in NodeIdx)
        (((hasClass[n] != 0 /\ not isPinned(hasClass[n])) \/ (isClass(n) /\ not
isPinned(n))) == (taint[n] == ALL));
```

3.3.1.7 Solution Objective

In this model, we require the solver to provide a satisfying assignment that minimizes the total number of call invocations that are in the cross-domain cut. Other objectives could be used instead.

```
var int: objective = sum(e in ControlDep_CallInv, l in nonNullEnclave where
xdedge[e,l])(1);
solve minimize objective;
```

Once the CAPO partitioning conflict analyzer has analyzed the CLE-annotated application code and determined that all remaining conflicts are resolvable by RPC-wrapping to result in a security-compliant cross-domain partitioned code, the conflict analyzer will produce a topology file (JSON) containing the assignment of every class to an enclave/level. A full-length version of the topology can be found in the appendix

3.4 Code Dividing and Refactoring

Once the CAPO partitioning conflict analyzer has analyzed the CLE-annotated application code, and determined that all remaining conflicts are resolvable by RPC-wrapping to result in a security compliant cross-domain partitioned code, the conflict analyzer will save the code in the refactored directory along with a topology file (JSON) containing the assignment of every function and global variable to an enclave/level. A sample topology JSON is provided below.

```
{
   "levels": ["orange", "purple"],
   "enclaves": ["purple_E", "orange_E"],
   "source_path": "./refactored",
   "functions": [
      {"name": "get_a", "level": "orange", "enclave": "orange_E", "file":
      "test1_refactored.c", "file_offset": 29},
      {"name": "main",  "level": "purple", "enclave": "orange_E", "file":
      "test1_refactored.c", "file_offset": 35},
      // ...
    ],
```

```
    "global_scoped_vars": [
      {"name": "globalScopeVarNotFunctionStatic", "level": "purple", "file":
      "test1_refactored.c", "file_offset": 5},
      // ...
    ],
    // ...
}
```

Given the refactored, annotated application, and the topology, the divider creates a `divvied` directory, divides the code into files in separate subdirectories (one per enclave), such that the source code for each function or global variable is placed in its respective enclave. Furthermore, all related code like type, variable, and function declarations, macro definitions, header includes, and pragmas are handled, so that the source in each directory has all the relevant code, ready for automated partitioning and code generation for RPC-wrapping of functions, and marshaling, tagging, serialization, and DFDL description of cross-domain data types.

This `divvied` source becomes the input to the GAPS Enclave Definition Language (GEDL) generator tool. The GEDL drives further code generation and modification needed to build the application binaries for each enclave.

The usage of the program divider is straightforward:

```
divider_plugin [-h] TOPOLOGYJSON [--output-dir=OUTPUT_DIR]
[--extra-arg=CLANG_ARGS]* --
```

The divider will always look in a directory named by the `source_path` field in the topology JSON. For example, in the sample above, it is called `refactored` under the working directory for any `*.cpp`, `*.c`, `*.hpp` or `*.h` files that may comprise the program. The –extra-arg option can be repeated multiple times to specify clang flags, e.g. header include paths.

3.5 RPC Generation

The CLOSURE C++ RPC generation is a manual process in this state of the prototype toolchain. Future work would build upon the corresponding tools developed for the C toolchain which generates the following artifacts:

- GEDL, a JSON document specifying all the cross domain calls and their associated data including the arguments and return type. This gedl is generated in an llvm `opt` pass which analyzes the divvied code.
- IDL, a text file with a set of C style struct data type definitions, used to facilitate subsequent serialization and marshaling of data types. The IDL syntax is based on C; an IDL file contains one or more C struct datatypes. Two structs are generated for each TAG request and response pair, with the in arguments in the request and

the out arguments in the response. The `idl_generator` script takes a gedl JSON and produces the idl.

- CODEC. For each struct in the IDL, a set of coding and decoding functions is generated to facilitate serialization to the remote enclave. The codecs consist of encode, decode, print functions for each of the structs, which handle byte order conversions between host and network byte order. The codecs can be generated using `hal_autogen`.
- RPC, remote procedure code that automates cross domain function invocation and replaces calls to those functions with ones that additionally marshall and send the data across the network using the generated codecs and IDL.
- DFDL, is an extension of XSD which provides a way to describe binary formats and easily encode/decode from binary to an xml infoset. CLOSURE has the ability to create DFDL schemas for each cross domain request/response pair with use of the `hal_autogen`.
- HAL Configurations. The `hal_autoconfig.py` produces a *HAL-Daemon* configuration.

In the meantime, we produce some of these manually to facilitate experimentation. See the C++ examples for details.

To work with the richer language constructs in C++, the RPC generator needs significant enhancement. It is expected that it needs to play the role of the AspectJ compiler in the Java toolchain. In addition, the CLOSURE library classes such as HalZmq and ClosureShadow and others will need to be implemented correspondingly in the C++ toolchain.

3.6 Marshalling/Serialization of Data Types

The general idea of marshalling/unmarshalling and serialization/deserialization of cross domain constructs remains the same as in the C toolchain. However, the current `idl_generator` and the `hal_autogen` scripts can only handle a flattened structure containing primitive types. They need to be extended to handle complex data types such as classes and nested structures. A depth-first tree traversal algorithm needs to be implemented to serialize and deserialize nested structures. For typedefs, the toolchain needs to take advantage of type information in llvm opt, and makes the actual types available to in gedl. Tools in the later phase are not expected to deal with typedefs directly.

3.7 Interfacing with HAL

The CLOSURE C++ toolchain's interface to the HAL daemon combines the approaches used in the CLOSURE C and Java toolchains.

With the CLOSURE C toolchain [10], partitioned application programs use the xdcomms API library to interface with the HAL daemon which exchanges data through the GAPS Devices.

In the Java toolchain, there is no separate xdcomms API library, and the interaction with the HAL daemon is handled by the code generated by the CodeGenJava tool. AspectJ-woven applications interact with the HAL through a CLOSURE library class. Details of marshaling/unmarshalling, serialization/de-serialization, and read/write to HAL, are abstracted and hidden from the applications

The C++ toolchain reuses the xdcomm API library to interface with the HAL daemon for data transport. For object-oriented features, such as object instantiation and method calls, since there is no reflection in C++ and AspectJ compiler, tools used in the C toolchain need to be modified and enhanced to automatically generate necessary code to work with C++ programs. This part is still in progress. In the meantime, see C++ example1 for code manually written to handle the HAL interface.

3.8 Example applications

3.8.1 Example 1

Example 1 is a simple example designed to gain insights and facilitate early development of CLOSURE toolchain for C++ programs. The targeted C++ features of the example are object instantiation and method calls, which, after partitioned by CLOSURE, are expected to be cross domain ones.

In this example, there are two classes `Foo` and `Bar` with no inheritance relationship between them. The annotation intention is to have `Bar` be on an enclave named "purple", and `Foo` to be present in both the "orange" and "purple" enclave. We pin the `main` function to a single enclave by annotating a variable inside it (`foo`) to be in orange. We annotate `Bar` with "PURPLE_SHAREABLE", signifying that all instances of `Bar` must be in the purple enclave and have the "PURPLE_SHAREABLE" label. Note that the rettaint of `Bar`'s constructor matches that of the class definition. `Bar::getValue` is a cross domain function returning `Bar::b`, which is shareable to the "orange" enclave. When partitioned, the call to `bar.getValue()` in `main` will become an RPC call.

```
#include "Bar.hpp"

#pragma cle def ORANGE {"level":"orange"}

class Foo
```

```
{
public:
  int a;

  int getValue()
  {
    return a;
  }

  Foo() {
    a = 1;
  }
};


int main(int argc, char **argv)
{
  __attribute__((cle_annotate("ORANGE")))
  Foo foo = Foo();
  Bar bar = Bar();
  int ret = foo.getValue() + bar.getValue();
  // printf("Result is: %d\n", ret);
  return ret;
}

#ifndef _BAR_
#define _BAR_

#pragma cle def PURPLE {"level":"purple"}

#pragma cle def PURPLE_SHAREABLE {"level":"purple",\
  "cdf": [\
    {"remotelevel":"orange", \
     "direction": "egress", \
     "guarddirective": { "operation": "allow"}}\
  ] }

#pragma cle def BAR_GETVALUE {"level":"purple",\
  "cdf": [\
    {"remotelevel":"orange", \
     "direction": "bidirectional", \
     "guarddirective": { "operation": "allow"}, \
     "argtaints": [["PURPLE_SHAREABLE"]], \
     "codtaints": ["PURPLE_SHAREABLE"], \
     "rettaints": ["PURPLE_SHAREABLE"] \
    }, \
    {"remotelevel":"purple", \
     "direction": "bidirectional", \
     "guarddirective": { "operation": "allow"}, \
     "argtaints": [["PURPLE_SHAREABLE"]], \
```

```
     "codtaints": ["PURPLE_SHAREABLE"], \
     "rettaints": ["PURPLE_SHAREABLE"] \
    } \
  ] }

#pragma cle def BAR {"level":"purple",\
  "cdf": [\
    {"remotelevel":"orange", \
     "direction": "bidirectional", \
     "guarddirective": { "operation": "allow"}, \
     "argtaints": [], \
     "codtaints": ["PURPLE_SHAREABLE"], \
     "rettaints": ["PURPLE_SHAREABLE"] \
    }, \
    {"remotelevel":"purple", \
     "direction": "bidirectional", \
     "guarddirective": { "operation": "allow"}, \
     "argtaints": [], \
     "codtaints": ["PURPLE_SHAREABLE"], \
     "rettaints": ["PURPLE_SHAREABLE"] \
    } \
  ] }


class __attribute__((cle_annotate("PURPLE_SHAREABLE"))) Bar
{
public:

  int __attribute__((cle_annotate("PURPLE_SHAREABLE"))) b;
  int c; // Assigned PURPLE or PURPLE_SHAREABLE


  int __attribute__((cle_annotate("BAR_GETVALUE"))) getValue()
  {
    return b;
  }


  __attribute__((cle_annotate("BAR"))) Bar() {
    b = 2;
  }

};

#endif /* _BAR_ */
```

3.8.1.1 Building Example 1 Example 1 can be built and analyzed using the familiar VSCode build tasks from the C Toolchain.

```
cd build/apps/examples/classes/example1/
code .
```

Use Ctrl-Shift-B to obtain the build tasks panel and run each task in numerical order.

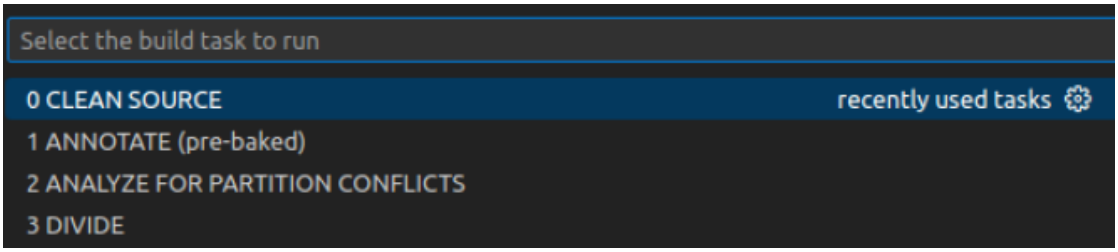

Figure 3: Figure: VSCode Tasks for C++ toolchain

4 Limitations and Future Work

4.1 Limitations and language coverage

The CLOSURE C++ prototype release supports up to and including the divider step for C++ programs featuring:

1. Function and Function calls
2. Class/struct/union inheritance (non-virtual)

3. Class/struct/union construction, destruction and method calls
4. Default constructors and destructors
5. Implicit destructor calls
6. Class fields and field accesses
7. Global scoped variables
8. Pointers and references
9. Operator overloading

4.2 Future Work

In the future, we would like to support more of C++98 and beyond, including but not limited to the following:

1. Initialization lists
2. Templates
3. Exceptions
4. Coroutines
5. Rvalue references

Other recommended improvements:

- Ability to exclude specified source files from the analysis. For now, users must manually exclude any source files (including those found the standard library) that they do not want to be analyzed.
- Additional precision to the alias analysis. Currently, the alias analysis does not take into account offsets provided by SVF, so it cannot differentiate record instances and their fields.

- Automation for the toolchain steps following division.

5 Appendices

5.1 The cross-domain cut specification: topology.json

The `topology.json` file is a description of level and enclave assignments produced by the conflict analyzer and is used as input for the code generator. It also contains information about the callee and caller that will be in the cut.

The `topology.json` contains:

1. the set of enclaves and levels relevant to the program
2. an assignment from each class to a level and an enclave
3. Callee and caller info for cross-domain calls

The `topology.json` manually generated for example1 is as follows:

```
{
   "enclaves": [
      {
         "name": "orange_E",
         "level": "orange",
         "assignedClasses": [
            "Foo"
         ]
      },
      {
         "name": "purple_E",
         "level": "purple",
         "assignedClasses": [
            "Bar",
            "Foo"
         ]
      }
   ],
   "levels": [
```

```
      "purple",
      "orange"
   ],
   "functions": [
      {
         "name": "main",
         "level": "orange",
         "enclave": "orange_E"
      }
   ],
   "global_scoped_vars": [
   ],
   "source_path":
   "/workspaces/cpp-closure/build/apps/examples/classes/example1/annotated"
}
```

5.2 Constraint Model in MiniZinc

The following contains type declarations for the MiniZinc model used within the conflict analyzer. These type declarations, along with a model instance generated in python are inputted to MiniZinc with the constraints to either produce a satisfiable assignment or some minimally unsatisfiable set of constraints.

```
%%%%%%%%%%%%%%%%
% Graph nodes %
%%%%%%%%%%%%%%%%

set of int: Decl_Var;         % A variable declaration. It will be marked with
global/static property
set of int: Decl_Function;    % A static global variable inside a function
set of int: Decl_Record;      % A class, struct or union definition
set of int: Decl_Field;       % A record field
set of int: Decl_Method;      % A record method. May have virtual/abstract
property
set of int: Decl_Param;       % A formal parameter of a function/method.
Contains information about parameter index
set of int: Decl_Constructor; % A constructor for a class/struct
set of int: Decl_Destructor;  % A destructor for a class/struct

set of int: Stmt_Decl;        % A statement which also has a declaration
set of int: Stmt_Call;        % A clang statement that represents a call to a
function, method, constructor or destructor
set of int: Stmt_Compound;    % A clang statement that only has other
statements as children
set of int: Stmt_Ref;         % A clang statement which holds a reference to a
decl
set of int: Stmt_Field;       % A clang statement which is a field accessor
set of int: Stmt_This;        % A clang statement which refers to 'this'
```

```
set of int: Stmt_Return;      % A return within a given function
set of int: Stmt_Other;       % A clang statement that does not transfer
control to another function

set of int: NodeIdx; % Spans all nodes

% Aggregate to unify constraints on functions, constructors, and methods
set of int: FunOrMethodOrCon = Decl_Function union Decl_Constructor union
Decl_Method union Decl_Destructor;

%%%%%%%%%%%%%%%
% Graph edges %
%%%%%%%%%%%%%%%

set of int: Struct_Field;                  % Connects a record to its field
| Decl.Record -> Decl.Field
set of int: Struct_Method;                 % Connects a record to its method
| Decl.Record -> Decl.Method
set of int: Struct_Constructor;            % Connects a record to its
constructor                      | Decl.Record -> Decl.Constructor
set of int: Struct_Destructor;             % Connects a record to its
destructor                       | Decl.Record -> Decl.Destructor
set of int: Struct_Inherit;                % Connects two records by
inheritance relation             | Decl.Record -> Decl.Record
set of int: Struct_Param;                  % Connects a function-like object to
its parameter         | Decl -> Decl.Param
set of int: Struct_Child;                  % When a statement has a child not
described by the above  | Stmt -> Stmt

set of int: Control_Return;                % A
function/method/constructor/destructor return          | Stmt.Return ->
Stmt.Call
set of int: Control_Entry;                 % Connects a function-like object to
it's body             | Decl -> Stmt

set of int: Control_FunctionInvocation;    % A direct call invocation to a
C-style function           | Stmt.Call -> Decl.Function
set of int: Control_MethodInvocation;      % A method call
| Stmt.Call -> Decl.Method
set of int: Control_ConstructorInvocation; % A constructor call
| Stmt.Call -> Decl.Constructor
set of int: Control_DestructorInvocation;  % A call to a destructor
| Stmt.Call -> Decl.Destructor

set of int: Data_PointsTo;                 % A points-to relation
| Decl -> Decl
set of int: Data_DefUse;                   % A def-use relation between
statements                     | Stmt.Decl -> Stmt.Ref
```

```
set of int: Data_ArgPass;                 % An argument pass to a call
instruction                   | Stmt.Call -> Stmt
set of int: Data_Return;                  % Connects data being returned to
the call site            | Stmt -> Stmt.Call
set of int: Data_Object;                  % Connects a method call/field to
the object being invoked | Stmt.Call/Stmt.Field -> Stmt
set of int: Data_FieldAccess;             % A field access
| Stmt -> Decl.Field
set of int: Data_InstanceOf;              % Connects a 'this' instance to a
class definition         | Stmt.This -> Decl.Record
set of int: Data_Decl;                    % Connects a decl statment to a decl
| Stmt.Decl -> Decl

set of int: EdgeIdx; % Spans all data and control edges

set of int: DataEdge =
    Data_PointsTo union Data_DefUse union Data_ArgPass union
    Data_Return union Data_Object union Data_FieldAccess union
    Data_InstanceOf union Data_Decl;

% Aggregates of enclave-safe and invocation edges
set of int: Control_EnclaveSafe =
    Struct_Field union Struct_Method union Struct_Constructor union
    Struct_Destructor union Struct_Inherit union Control_Entry union
Struct_Child union Struct_Param;

set of int: Control_Invocation =
    Control_FunctionInvocation union Control_MethodInvocation union
    Control_ConstructorInvocation union Control_DestructorInvocation;

set of int: Data_EnclaveSafe =
    Data_FieldAccess union Data_InstanceOf union
    Data_Object union Data_DefUse union Data_Decl union Data_ArgPass;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Containment mappings for nodes, edge endpoints, parameter indices, and node
properties %

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

array[NodeIdx]  of bool: isGlobal;
array[NodeIdx]  of int:  hasClass; % 0 for node not in a class, otherwise maps
to the Decl_Record containing the node

array[NodeIdx]  of bool: userAnnotatedNode;

array[NodeIdx]  of int:  hasFunction; % 0 for globals, otherwise maps to the
Decl_Function/Method/Constructor containing the node
```

```
array[EdgeIdx]  of int:  hasSource;
array[EdgeIdx]  of int:  hasDest;
array[NodeIdx]  of int:  hasParamIdx;

% Note: if n is a Decl_Record (i.e. a class declaration), then hasClass[n] == 0
if the class is not contained in another class,
% but if n is a Decl_Function then hasFunction[n] == n (i.e. a function
contains itself but a class does not)

%%%%%%%%%%%%%%%%%%%
% CLE input model %
%%%%%%%%%%%%%%%%%%%

enum Level;
enum Enclave;
array[Enclave] of Level: hasEnclaveLevel;

enum cleLabel;
enum cdf;
enum GuardOperation = {nullGuardOperation, allow, deny, redact};
enum Direction      = {nullDirection, bidirectional, egress, ingress};

int: MaxFnParams; % Upper bound on number of parameters for any function in the
program
set of int: paramIdx = 1..MaxFnParams;

array[cleLabel]                of Level:          hasLabelLevel;
array[cleLabel]                of bool:           isFunctionAnnotation;
array[cdf]                     of cleLabel:       fromCleLabel;
array[cdf]                     of Level:          hasRemotelevel;
array[cdf]                     of GuardOperation: hasGuardOperation;
array[cdf]                     of Direction:      hasDirection;
array[cdf]                     of bool:           isOneway;
array[cleLabel, Level]         of cdf:            cdfForRemoteLevel;

array[cdf, cleLabel]           of bool:           hasRettaints;
array[cdf, cleLabel]           of bool:           hasCodtaints;
array[cdf, paramIdx, cleLabel] of bool:           hasArgtaints;
array[cdf, cleLabel]           of bool:           hasARCtaints;

%%%%%%%%%%%%%%%%%%%%%%
% Decision variables %
%%%%%%%%%%%%%%%%%%%%%%

array[NodeIdx] of var Enclave:  nodeEnclave;
array[NodeIdx] of var cleLabel: taint;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Utility functions and predicates %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

predicate isFunction(NodeIdx: n)    = (n in FunOrMethodOrCon);
predicate isClass(NodeIdx: n)       = (n in Decl_Record);
predicate allowOrRedact(var cdf: c) = (hasGuardOperation[c] == allow \/
hasGuardOperation[c] == redact);

predicate annotFun(NodeIdx: n) = hasFunction[n] != 0 /\
userAnnotatedNode[hasFunction[n]];

predicate sourceAnnotFun(EdgeIdx: e) =
    (if hasFunction[hasSource[e]] != 0 then
userAnnotatedNode[hasFunction[hasSource[e]]] else false endif);

predicate destAnnotFun(EdgeIdx: e) =
    (if hasFunction[hasDest[e]] != 0 then
userAnnotatedNode[hasFunction[hasDest[e]]] else false endif);

predicate isInArctaint(var cleLabel: fan, var cleLabel: tnt, var Level: lvl) =
    (if isFunctionAnnotation[fan] then hasARCtaints[cdfForRemoteLevel[fan,
lvl], tnt] else false endif);

predicate xdedge(EdgeIdx: e) = (nodeEnclave[hasSource[e]] !=
nodeEnclave[hasDest[e]]) /\ nodeEnclave[hasSource[e]] != all_E /\
nodeEnclave[hasDest[e]] != all_E;

predicate taintsAgree(var cleLabel: t1, var cleLabel: t2) = t1 == t2 \/ t1 ==
ALL \/ t2 == ALL;

predicate child(NodeIdx: n, NodeIdx: c) = exists (e in Struct_Child)
(hasSource[e] == n /\ hasDest[e] == c);
predicate ancestor(NodeIdx: n, NodeIdx: c) = child(n, c) \/ exists (d in
NodeIdx) (child(n, d) /\ ancestor(d, c));

function var cleLabel: esTaint(EdgeIdx: e) = taint[hasSource[e]];
function var cleLabel: edTaint(EdgeIdx: e) = taint[hasDest[e]];

function var cleLabel: esFunTaint(EdgeIdx: e) = if sourceAnnotFun(e) then
taint[hasFunction[hasSource[e]]] else nullCleLabel endif;
function var cleLabel: edFunTaint(EdgeIdx: e) = if destAnnotFun(e)   then
taint[hasFunction[hasDest[e]]]   else nullCleLabel endif;

function var cdf: funCdf(NodeIdx: n) = if annotFun(n) then
cdfForRemoteLevel[ftaint(n), hasLabelLevel[taint[n]]] else nullCdf endif;

function var cdf: esFunCdf(EdgeIdx: e) = if sourceAnnotFun(e) then
cdfForRemoteLevel[esFunTaint(e), hasLabelLevel[edTaint(e)]] else nullCdf endif;
function var cdf: edFunCdf(EdgeIdx: e) = if destAnnotFun(e)   then
cdfForRemoteLevel[edFunTaint(e), hasLabelLevel[esTaint(e)]] else nullCdf endif;
```

```
function var cleLabel: ftaint(NodeIdx: n) = if hasFunction[n] != 0 then
taint[hasFunction[n]] else nullCleLabel endif;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Structural constraints on node enclave/level %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Nodes must have a non-null enclave which is shared within a function and
matches their level
constraint :: "NodeHasEnclave"            forall (n in NodeIdx) (nodeEnclave[n]
!= nullEnclave);
constraint :: "NodeEnclaveIsFunEnclave"   forall (n in NodeIdx)
((hasFunction[n] != 0) -> (nodeEnclave[n] == nodeEnclave[hasFunction[n]]));
constraint :: "NodeEnclaveIsClassEnclave" forall (n in NodeIdx) ((hasClass[n]
!= 0) -> (nodeEnclave[n] == nodeEnclave[hasClass[n]]));
constraint :: "NodeLevelAtEnclaveLevel"   forall (n in NodeIdx)
(hasLabelLevel[taint[n]] == hasEnclaveLevel[nodeEnclave[n]]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Classifying annotations %
%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Annotations on functions, methods, and constructors are function annotations
and can only be made by the user.
% No other node should have a function annotation.
constraint :: "FnAnnotationForFnOnly"
    forall (n in NodeIdx)          (isFunctionAnnotation[taint[n]] ->
isFunction(n));
constraint :: "FnAnnotationByUserOnly"
    forall (n in FunOrMethodOrCon) (isFunctionAnnotation[taint[n]] ->
userAnnotatedNode[n]);
constraint :: "annotationOnFunctionIsFunAnnotation"
    forall (n in FunOrMethodOrCon) (userAnnotatedNode[n] ->
isFunctionAnnotation[taint[n]]);

% Annotations on classes and their fields are node annotations, not function
annotations.
constraint :: "annotationOnClassIsNodeAnnotation"
    forall (n in Decl_Record) (userAnnotatedNode[n] -> not
(isFunctionAnnotation[taint[n]]));
constraint :: "annotationOnFieldIsNodeAnnotation"
    forall (n in Decl_Field)  (userAnnotatedNode[n] -> not
(isFunctionAnnotation[taint[n]]));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Structural taint relationships between nodes %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% All nodes in an un-annotated function must have the taint of the function
constraint :: "UnannotatedFunContentTaintMatch"
    forall (n in NodeIdx where hasFunction[n] != 0) ((not
userAnnotatedNode[hasFunction[n]]) -> (taint[n] == ftaint(n)));

% All nodes in an un-annotated class must have the taint of the class
% (may be ALL if multiple taints construct the class, but not if the class
accesses tainted data)
constraint :: "UnannotatedClassTaintsMatch"
    forall (n in NodeIdx where hasClass[n] != 0)
        ((not userAnnotatedNode[hasClass[n]]) -> (taint[n] ==
taint[hasClass[n]]));

% All nodes in an un-annotated class must be un-annotated
constraint :: "noAnnotatedDataForUnannotatedClass"
    forall (n in NodeIdx where hasClass[n] != 0)
        ((not userAnnotatedNode[hasClass[n]]) -> (not userAnnotatedNode[n]));

% Un-annotated constructors get the class taint
constraint :: "unannotatedConstructorGetsClassTaint"
    forall (n in Decl_Constructor)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));

% Un-annotated destructors get the class taint
constraint :: "unannotatedDestructorGetsClassTaint"
    forall (n in Decl_Destructor)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));

% Un-annotated methods get the class taint
constraint :: "unannotatedMethodGetsClassTaint"
    forall (n in Decl_Method)
        ((hasClass[n] != 0 /\ not userAnnotatedNode[n]) -> (taint[n] ==
taint[hasClass[n]]));

% Classes connected by an inheritance relationship share the same taint
constraint :: "inheritTaint"
    forall (e in Struct_Inherit) (esTaint(e) == edTaint(e));

% All nodes in an annotated function must have a taint in the ARCtaints
constraint :: "AnnotatedFunContentCoercible"
    forall (n in NodeIdx where (hasFunction[n] != 0) /\ (not isFunction(n)))
        (userAnnotatedNode[hasFunction[n]] -> isInArctaint(ftaint(n), taint[n],
hasLabelLevel[taint[n]]));

% Annotated constructors (which are only valid in annotated classes) must have
the class taint
% as their sole rettaint in every CDF
```

```
% TODO: doesn't validate that it is the only rettaint
constraint :: "annotatedConstructorReturnsClassTaint"
    forall (n in Decl_Constructor)
        (userAnnotatedNode[n] ->
            (forall (lvl in Level)
                ((cdfForRemoteLevel[taint[n], lvl] != nullCdf) ->
hasRettaints[cdfForRemoteLevel[taint[n], lvl], taint[hasClass[n]]])));

% Any function whose address is taken in the program cannot have a function
annotation
constraint :: "FunctionPtrSinglyTainted"
    forall (e in Data_PointsTo) (isFunction(hasDest[e]) -> not
userAnnotatedNode[hasDest[e]]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Control never leaves enclave except via valid XDC %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

constraint :: "NonCallRetControlEnclaveSafe"
    forall (e in Control_EnclaveSafe) (xdedge(e) == false);
constraint :: "XDCallBlest"
    forall (e in Control_Invocation) (xdedge(e) ->
userAnnotatedNode[hasDest[e]]);
constraint :: "XDCallAllowed"
    forall (e in Control_Invocation) (xdedge(e) -> allowOrRedact(edFunCdf(e)));
% constraint :: "XDReturnAllowed"
    % forall (e in Control_Return) (xdedge(e) -> allowOrRedact(esFunCdf(e)));

% caller -[Control_Invocation]-> callee <-[hasFunction]- Stmt_Return = Return
Edge



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Data never leaves enclave except via parameters or return for valid XDC %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

constraint :: "EnclaveSafeDataEdges"
    forall (e in Data_EnclaveSafe) (xdedge(e) == false);

% Cross-domain returns must be allowed by the CDF, but the taints need not
match
% (can't enforce taint matching as the level must change).
% constraint :: "XDReturnDataAllowed"
%     forall (e in Data_Return)
%         (xdedge(e) -> allowOrRedact(esFunCdf(e)));

% All children of returns must have the same taint as the return stmt
constraint :: "ReturnChildMatchesReturn"
```

```
    forall (n in Stmt_Return) (
        forall (c in NodeIdx) (ancestor(n, c) -> taint[n] == taint[c])
    );

% Return statments in annotated functions must have a taint
% which is one of the rettaints
constraint :: "ReturnStmtMatchesRettaint"
    forall (n in Stmt_Return where annotFun(n)) (
        hasRettaints[funCdf(n), taint[n]]
    );

% For any function called cross-domain, all its returns
% must have taints shareable with the callee level
constraint :: "XDReturnAllowed"
    forall (e in Control_Invocation where xdedge(e)) (
        forall (r in Stmt_Return where hasFunction[r] == hasDest[e])
            (allowOrRedact(cdfForRemoteLevel[taint[r],
hasLabelLevel[esTaint(e)]]))
    );

% Cross-domain Points-To edges must be allowed by the CDF, but the taints need
not match.
% Note: The reason cross domain points-to edges are not disallowed completely
% is because passing arrays necessitates cross-domain points-to edges.
constraint :: "XDPointsToAllowed"
    forall (e in Data_PointsTo)
        (xdedge(e) -> (allowOrRedact(cdfForRemoteLevel[edTaint(e),
hasLabelLevel[esTaint(e)]]) /\ not isFunction(hasDest[e])));

%%%%%%%%%%%%%%%%%%%%%
% Taint propagation %
%%%%%%%%%%%%%%%%%%%%%

% Data edges within un-annotated functions are fully constrained by
UnannotatedFunContentTaintMatch,
% because the source and destination are in the same function.

% Data edges within annotated functions are partially constrained by
AnnotatedFunContentCoercible,
% so we have that the source and destination must be in the function's
ARCtaints.
% Sometimes, however, pointer dependencies are captured by a chain
% of points-to edges which may be intra-function edges. Therefore
% we restrict intra-function points-to edges to have the
% same taint, even in annotated functions.
predicate intraFunEdge(DataEdge: e) =
    (hasFunction[hasSource[e]] != 0 /\ hasFunction[hasDest[e]] != 0 /\
     hasFunction[hasSource[e]] == hasFunction[hasDest[e]]);
constraint :: "intraFunPointsToTaintsMatch"
```

```
    forall (e in Data_PointsTo) (intraFunEdge(e) -> esTaint(e) == edTaint(e));

% For ANY data edge between two function-external nodes, the taints must match
predicate globalGlobalEdge(DataEdge: e) = (isGlobal[hasSource[e]] /\
isGlobal[hasDest[e]]);
constraint :: "externExternDataEdgeTaintsMatch"
    forall (e in DataEdge) (globalGlobalEdge(e) -> taintsAgree(esTaint(e),
edTaint(e)));

% For ANY data edge between data within a function and a function-external
node, the taints must match
predicate srcFunExternEdge(DataEdge: e)  = (hasFunction[hasSource[e]] != 0 /\
hasFunction[hasDest[e]]   == 0);
predicate destFunExternEdge(DataEdge: e) = (hasFunction[hasDest[e]]   != 0 /\
hasFunction[hasSource[e]] == 0);
predicate funExternEdge(DataEdge: e)     = (srcFunExternEdge(e) \/
destFunExternEdge(e));
constraint :: "externDataEdgeTaintsMatch"
    forall (e in DataEdge) (funExternEdge(e) -> taintsAgree(esTaint(e),
edTaint(e)));

% This leaves taint propagation for inter-function data edges.

% For non-XD return edges from a callee function to a callsite:
%   -If the callee (source) function is un-annotated, the taints must match
%   -If the callee (source) function is annotated, the taint on the dest node
%    must be in the callee's rettaints

% constraint :: "retEdgeFromUnannotatedTaintsMatch"
%     forall (e in Data_Return where not sourceAnnotFun(e))
%         (taintsAgree(esTaint(e), edTaint(e)));
% constraint :: "returnNodeInRettaints"
%     forall (e in Data_Return where sourceAnnotFun(e) /\ not xdedge(e))
%         (hasRettaints[esFunCdf(e), edTaint(e)]);


% The taint of the returned value of a call to an unannotated function must
match the taint
% of the unannotated function
constraint :: "returnFromUnannotatedTaintsMatch"
    forall (e in Control_Invocation where not destAnnotFun(e))
        (taintsAgree(esTaint(e), edTaint(e)));

% The taint of the returned value of a (not xd) call to an annotated function
% must match one of the rettaints
constraint :: "returnInRettaints"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (hasRettaints[edFunCdf(e), esTaint(e)]);
```

```
% The taint of the returned value of a (not xd) call to an annotated function
% must match the taints of all of the callee's returns
constraint :: "callsiteTaintMatchesAllReturns"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (forall (r in Stmt_Return where hasFunction[r] == hasDest[e])
            (taintsAgree(taint[r], esTaint(e))));


% ArgPass edges: The callsite node (Stmt_Call) has ArgPass edges to each of its
arguments, and call invocation edges
% to each potential callee. The ParamIdx is on the argpass edge destination and
the Decl_Param (unused).
% For non-XD argument passing edges:
%   -If the destination function is un-annotated, the taint of the argument
must match the taint of the destination function
%   -If the destination function is annotated, the taint of the argument must
be in the argtaints of the function at that parameter index
constraint :: "argumentToUnannotatedTaintsMatch"
    forall (e in Control_Invocation where not destAnnotFun(e))
        (forall (arg_e in Data_ArgPass where hasSource[arg_e] == hasSource[e])
            (taintsAgree(edTaint(arg_e), edTaint(e))));
constraint :: "argumentInArgtaints"
    forall (e in Control_Invocation where destAnnotFun(e) /\ not xdedge(e))
        (forall (arg_e in Data_ArgPass where hasSource[arg_e] == hasSource[e])
            (hasArgtaints[edFunCdf(e), hasParamIdx[hasDest[arg_e]],
taint[hasDest[arg_e]]]));

% All children of anything passed as an argument must have
% the same taint as the argument node
constraint :: "argumentChildren"
    forall (e in Data_ArgPass) (
        forall(n in NodeIdx where ancestor(hasDest[e], n))
            (taint[n] == edTaint(e))
    );

% Inter-function points-to edges should always have the same taint
predicate interFunEdge(DataEdge: e) =
    (hasFunction[hasSource[e]] != 0 /\ hasFunction[hasDest[e]] != 0 /\
     hasFunction[hasSource[e]] != hasFunction[hasDest[e]]);
constraint :: "interFunPointsToTaintsMatch"
    forall (e in Data_PointsTo)
        ((not xdedge(e) /\ interFunEdge(e)) -> taintsAgree(esTaint(e),
edTaint(e)));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Enabling shared classes %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% A class is pinned if:
```

```
% - the class is annotated
% - anything in the class is annotated
% - it depends on anything external to the class
% Note: relies on all edges following a "source depends on destination"
convention
% (as opposed to "destination depends on source"). Can be reversed, or
specialized per edge type.
% TODO: this will pin an inherited class, when we may want both to be unpinned.
Exclude the Struct_Inherit edge?
% TODO: this may be an expensive predicate. It can be pre-computed and placed
into an array. Not sure what the backend does with this.
predicate isPinned(Decl_Record: r) = (userAnnotatedNode[r] \/
    (exists (n in NodeIdx) (hasClass[n] == r /\ userAnnotatedNode[n])) \/
    (exists (e in EdgeIdx) (hasClass[hasSource[e]] == r /\ hasClass[hasDest[e]]
!= r)));

% A node has the ALL label if and only if it is contained in an unpinned class
constraint :: "definitionForALL"
    forall (n in NodeIdx)
        (((hasClass[n] != 0 /\ not isPinned(hasClass[n])) \/ (isClass(n) /\ not
isPinned(n))) == (taint[n] == ALL));
```

5.3 HAL Configuration Files

See the C documentation [10].

5.4 Dockerfile

5.4.1 Dockerfile for Source release

The following dockerfile is used to build a source release from scratch.

```
FROM ubuntu:focal
ARG DEBIAN_FRONTEND=noninteractive

RUN apt -y update

# general prereqs
RUN apt -y install \
    make \
    git \
    libz3-dev \
    sudo \
    cmake \
    python3 \
    python3-pip \
    unzip
```

```
# python dependencies
RUN pip install \
    intervaltree \
    pytest \
    z3-solver

# llvm-14 prereqs
RUN apt -y install \
    lsb-release \
    wget \
    software-properties-common \
    gnupg

# install llvm/clang 14 with API
RUN cd /tmp \
    && wget https://apt.llvm.org/llvm.sh \
    && chmod +x llvm.sh  \
    && ./llvm.sh 14 all
RUN apt install -y \
    clang-14 \
    libclang-14-dev \
    llvm-14

# link clang-14 binaries
RUN update-alternatives --install /usr/bin/c++ c++ /usr/bin/clang++-14 60 \
    && update-alternatives --install /usr/bin/clang clang /usr/bin/clang-14 60
    \
    && update-alternatives --install /usr/bin/opt opt /usr/bin/opt-14 60

# install minizinc
RUN cd /opt/ \
    &&  sudo mkdir minizinc \
    &&  sudo wget
    https://github.com/MiniZinc/MiniZincIDE/releases/download/2.5.5/MiniZincIDE-2.5.5-bundle-linu
    64.tgz
    \
    &&  sudo tar -xvzf MiniZinc* -C minizinc --strip-components=1

# make closure user
RUN useradd -ms /bin/bash closure \
    && usermod -aG sudo closure \
    && echo "%sudo ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
USER closure
ENV HOME /home/closure

# copy and build closure
COPY --chown=closure . /tmp/cpp-closure
RUN cd /tmp/cpp-closure \
```

```
    && sudo mkdir /opt/closure \
    && sudo chown closure /opt/closure \
    && BUILD_DIR=/tmp/cpp-closure/tmp-build INSTALL_DIR=/opt/closure make
    install

ENV PATH=/opt/minizinc/bin:/opt/closure/bin:$PATH
```

References

[1] Y. Solodkyy, "Simplifying the analysis of c++ programs," PhD thesis, 2013.

[2] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2359–2371. doi: 10.1145/3133956.3134066.

[3] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for information flow control in java programs - a practical guide," in *Proceedings of the 6th working conference on programming languages (ATPS'13)*, Feb. 2013, pp. 123–138.

[4] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the c language," *Journal of Automated Reasoning*, vol. 43, no. 3, pp. 263–288, Jul. 2009, doi: 10.1007/s10817-009-9148-3.

[5] K. Memarian *et al.*, "Into the depths of c: Elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN conference on programming language design and implementation*, 2016, pp. 1–15. doi: 10.1145/2908080.2908081.

[6] M. Batty, "The C11 and c++11 concurrency model," 2015. Available: https://api.semanticscholar.org/CorpusID:84182258

[7] "BRiCk." Github. Available: https://github.com/bedrocksystems/BRiCk

[8] "Visual studio code," *Visual Studio Code - Code editing. Redefined.* Microsoft, 2022. Available: https://code.visualstudio.com/

[9] M. J. Beckerle and S. M. Hanson, "Data format description language (DFDL) v1.0 specification," *Data Format Description Language (DFDL) v1.0 Specification*. The Apache Software Foundation, 2022. Available: https://daffodil.apache.org/docs/dfdl/

[10] "CLOSURE toolchain user manual for CLanguage." GitHub, 2022. Available: https://github.com/gaps-closure/gaps-closure.github.io/tree/develop/docs/C

[11] P. J. Stuckey, K. Marriott, and G. Tack, "The minizinc handbook," *The MiniZinc Handbook - The MiniZinc Handbook 2.5.5*. 2022. Available: https://www.minizinc.org/doc-2.5.5/en/index.html