A Humane, Graph-Based Representation of Programs and Analyses

Jon Mathews, Theodore Murdock, Jeremías Sauceda,
Ahmed Tamrawi, Suresh Kothari
EnSoft Corp., Ames, Iowa

August 13, 2024

Just as high-level program languages are designed with affordances for humans, effort must be invested in designing humane representations for program analyses. Human-machine collaboration must be enabled to tackle difficult problems now, and to help inspire creative solutions to automate or semi-automate verification in the future.

We present XCSG, an eXtensible Common Software Graph designed to enable human-machine collaboration to tackle difficult verification problems in multi-million line programs. XCSG represents the semantics of the software, blends various common analyses, provides a basis for composing new analyses, and provides affordances required for human-machine collaboration.

A case study is presented based on a verification tool which was prototyped and deployed using XCSG. The tool has been used to verify safety and security properties in 37MLOC, and 8 confirmed bugs have been found to date. The study details how a useful analysis can be integrated into XCSG, take advantage of existing affordances for human collaboration, and compose with existing analyses.

We also present the associated query language which is used to help rapidly prototype projections of the graph.

1 Introduction

We focus on developing program analyses for auditing software, as well as designing analyses to help the human audit software.[1] But why focus on human-machine collaboration? In [8], Brooks puts forward the thesis that, "*intelligence amplifying* systems can, at any given level of available systems technology, beat [Artificial Intelligence] systems. That is, a machine *and* a mind can beat a mind-imitating machine working by itself." Amplifying intelligence is especially crucial for solving hard problems for which automation by itself does not scale, particularly for large software.

1.1 The Humane Program Analysis Problem

For many program analyses, even those targeted at well-defined properties, it is necessary to involve a human in the process of deciding whether a program behavior is a feature, or a defect.

There are two major components of the auditing system: the machine and the human. Both of these components require time to operate, but there is also time consumed in their collaboration. A large body of research is focused on precision and scalability of program analyses [21, 23, 2, 24, 13]. If one compares the process of auditing software to running a maze of dependencies and possible program states, this body of research could be characterized as trying to reduce the size of the maze; this is an important research goal. However, given the state of the art, the machine is not going to find the exit by itself. If one considers the human's necessary role, it is in helping to decide where to turn at key junctions. For efficiency of the auditing process, the machine must represent the current state of analysis in a manner which helps the human make the next key decision, so that the machine can continue to help the human run the maze.

Therefore, we operate under the assumption that program analyses should provide a useful next step, they should not be an end in themselves.

In this whitepaper, we explore the following research question: What are the key features of a system supporting human-machine collaboration in software auditing? We present XCSG as a representative system designed to enable human-machine collaboration to tackle difficult verification problems in multi-million line programs.

[1]In this context, by *audit* we include systematic verification of critical safety and security properties in multi-million line code bases.

2 Motivations and Background

2.1 Why Graphs?

We believe graphs are the central feature of the system, partly as they have a long history of being useful structures for proving properties in mathematics. Graphs have been used in compilers for decades; Allen [6] introduced graphs for use in program optimization.

But what should the graph represent? For a single program analysis, the required semantics and representations can be specified. For a human, it is difficult to know in advance what will be needed. Fortunately, it has become tractable to put whole-program representations in graph databases; there is increasing interest in using graph databases to help search and analyze software [27, 15, 33, 18]. The important question is no longer whether we can put entire Abstract Syntax Trees (ASTs) and program analyses into a database, rather, how should those abstractions be structured to facilitate human-machine collaboration?

A weak form of collaboration would be to expect the human to simply read and interpret the graph. However, the graph should not be an end in itself. The intermediate results may be expressed as a graph, but that graph should facilitate human interaction and further automated analyses. In essence, it should keep the human-machine collaboration flowing. Ideally, analyses would be designed to help the human stay focused on the most critical questions, but would also extend the human's reach with further analyses aimed at resolving the answers to specific questions. This mode of collaboration can help overcome issues with tractability of more complex or fine-grained analyses, and keep the intermediate analysis results more succinct.

Although many analysis results can be imported into a graph database, we cannot put all possible analyses into the database; a trivial example being an infinite trace from a non-terminating program. Further, there are now many software systems comprising millions of lines of code each. And so we operate under the additional design pressure of focusing first on those analyses and representations which can scale. Further, it is important to consider how these are blended in the graph representation, especially as this affects the creation and composition of analyses and impacts the collaboration between the machine and the human.

2.2 Graphs in Program Analysis

As noted earlier, graphs have a long history of being useful structures for proving properties in mathematics, and have long been used in compilers, going back to Allen's work on program optimization [6].

Program Dependence Graphs (PDGs) [10] unified control and data flow, and System Dependence Graphs (SDGs) [17] extended them for interprocedural analysis. More recently, PDGs have been adapted for object-oriented languages, in particular for Java Virtual Machine bytecode [14]. These scalable graph representations were created in part for optimizing compilers – clarifying human intent to the machine. One of our goals is enabling collaboration with the human – in part, clarifying machine-oriented communication back to the human. The early work on PDGs includes speculation about their suitability for use in software development environments [22], particularly for slicing [30, 31]. However, in [28], Waters makes a particularly compelling argument for an analysis that aids both automation and human reasoning.

2.3 Locality

Waters presents a method for analyzing loops based on *plan building methods* (PBMs), which are presented as a way to make it easier to understand loops and to guide proofs of correctness [28]. In making the case, Waters uses an example of a for loop which sums the elements of an array. In a text-based representation, the initialization of the variable used to accumulate the result is necessarily separated from the statement which performs the accumulation, in part due to the loop control statement. Waters goes on to argue that separating two elements that have a close relationship makes it harder to understand the effects of the loop.

In light of these issues, graphs provide another obvious benefit: they can help display related entities next to each other despite lexical separation.

2.4 Blending Analyses

Graphs serve as a common substrate to combine various analyses. As noted earlier, [27] puts forward the idea that multiple analyses may be combined in a single graph database, examples including structural, semantic (as in resolving identifier bindings), control flow, data flow, and run-time information.

Part of their rationale is that many queries are likely to require information from multiple analyses to express.

We aim for a graph representation which blends analyses in a consistent form. A uniform representation is helpful for inspiring creative solutions to difficult problems. Further, we seek analyses which can be composed with one another, both for automation and human-machine collaboration.

3 XCSG: A Humane, Graph-Based Representation of Programs and Related Analyses

3.1 Drivers of Design – What is XCSG?

First, some background context: the platform we use to develop XCSG provides a facility for viewing the text-based form of a program next to a graph-based view. The two are linked: clicking on the text translates to nodes and/or edges in graph, and vice-versa. The platform allows us to create plug-ins which respond to the selections in either view, and respond in terms of a graph we calculate (such as data flow graph in response to clicking on a variable). This enables collaboration between the human and the machine; the basic user interface elements can be seen in Figure 1.

```
public class T extends java.lang.Object
{
    int f;
    public void m(int)
    {
        T r0;
        int i0;
        r0 := @this: T;
        i0 := @parameter0: int;
        r0.<T: int f> = i0;
        return;
    }
}
```

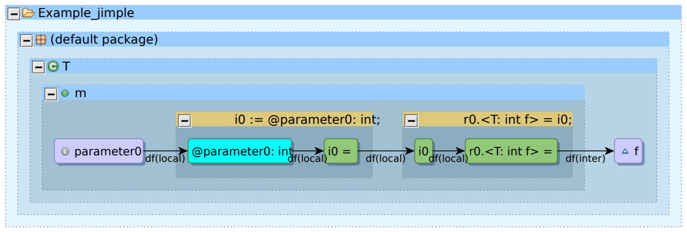


Figure 1: Enabling Human-Machine Collaboration: The Text and Graph are Linked.

To ease interaction, the representation should ease navigation of commonly used relationships, and provide correspondence back to the text form. This is accomplished by annotating the graph with source correspondence, which enables the correspondence between text and graph mentioned above.

To enable analysis XCSG must represent the whole program. This could be accomplished by importing the Abstract Syntax Tree (AST), but for our purposes this would be merely necessary, and hardly sufficient. This is a serious design question, and will take the rest of this section to explain.

The equivalence classes used to model the kinds of nodes and edges ultimately affect the utility and succinctness of expression. In general, our design choices come down to deciding how to model the key entities and relationships, and what those equivalence classes should be. This is true for deciding how to group nodes, but also for edges as well, as it affects transitive traversal of the graph.

Much of our interaction with XCSG is through a query language, which is described in Section 4. For now, it suffices to know that it provides a way to specify traversals over subsets of edges. However, it deliberately does not provide sophisticated pattern matching found in other graph query languages [3, 32] (but it does support edges with multiple kinds, more on this later). The minimality of the query language has put pressure on us to carefully choose equivalence classes for nodes and edges, where otherwise we might have simply transliterated the AST. This process results in a model that is easier to use.

3.2 XCSG for Jimple

The design goals of XCSG include the ability for additional layers of analysis to be incorporated into an XCSG graph (making it eXtensible), and to provide consistent representations of the semantics of multiple programming languages (making it a Common software graph); for the remainder of this section, however, we primarily focus on how XCSG represents the Jimple intermediate representation of Java bytecode [4], and the design considerations behind that representation. XCSG captures semantics of the whole program; there is enough information to translate from XCSG back to Jimple. [2]

The focus of this section is on describing the primary aspects of the design: the equivalence classes of the nodes, and especially the edges.

The graphs described by XCSG are directed; in practice, the underlying graph database is expected to support traversal in either direction. Nodes

[2]Jimple itself is already a step up from Java bytecode in terms of easing analysis and understanding – Jimple has less than 20 operations, where Java bytecode has more than 200 [4]. Further, Jimple is translated from the stack-based bytecode to three-address code, which is easier to read.

and Edges may have attributes, and typically have names for display purposes, as well as source correspondence for interaction with a text-based form of the program. Nodes and Edges may have *kinds*, a term used to avoid ambiguity with the types of the language being represented in XCSG. In practice, kinds are implemented in terms of constant strings. However, XCSG describes relationships between kinds. For example, a static method has the kind *ClassMethod*, which is a kind of *Method*. In practice, this helps a user query by broader or narrower equivalence classes. For example, a query for the kind *Type* implicitly includes *Java.Class*, *Java.Interface*, *Java.Enum*, and *Primitive*.

At present, the analyses covered by XCSG could be broadly characterized as: declarative structure, calls, control flow, and data flow. While the equivalence classes of nodes is relatively straightforward to describe, the equivalence classes of edges has proven an interesting design problem. Achieving locality in a visual representation is aided by creating edges between related nodes, but traversal is affected by the choice of direction and the equivalence classes of edges.

The remainder of this section is organized not by nodes, but by how edges are used to model relationships, as this is the key to understanding XCSG. While not intended to be exhaustive, this discussion is intended to be sufficient to understand the example analyses presented in Section 5. Additional documentation of XCSG is hosted at [5].

3.3 Everything is Connected

Nodes are organized mostly according to lexical nesting using *Contains* edges, the subgraph of which forms a tree. This serves two purposes. First, when visualized, *Contains* edges are displayed as nodes nested within nodes; we have found that this is helpful to give context for the location of nodes, especially those farther down in the *Contains* hierarchy. Second, it provides a convenient way to write a query which transitively includes everything under a given package, type or method, depending on the use case. This extends all the way down to data flow, a fact which we exploit later in our case study in Section 5.

The declarative structure of the program covers entities such as types and methods. It also includes containers such as *Project* and *Library*, which are not necessarily part of the program, but have been useful in practice for selecting the declarations thereunder.

A typical path along *Contains* edges down to the parameter of a method goes through nodes with the following XCSG kinds: *Project*, *Package*, *Type*, *Method*, *Parameter*, as can be seen in Figure 1. The outermost node is the *Project*, the rest are nested per the hierarchy. Also shown in the figure is a path through *DataFlow_Edge*s, which proceeds from the *Parameter* on the left, through *DataFlow_Node*s which are nested in *ControlFlow_Node*s (yellow), and ends at a *Field* on the right. The names of nodes for declarative entities are their simple names, as the fully qualified names are implied by the *Contains* hierarchy.

3.4 Avoid Overconnecting

Supertype and *Overrides* relationships are modeled between *Types* and *Methods*, respectively. Typing relations are modeled based on immediate relationships between types, as opposed to transitive closure. While [27] argued for a similar representation on the basis that many query languages support transitive closure, we do so because displaying a type hierarchy with full transitive closure of the typing relationships would be unwieldy. On a somewhat related note, XCSG currently models *Methods* in the location where they are declared, as opposed to replicating them to *Types* which inherit them.

3.5 Use Edges Consistently

*Parameter*s and *Field*s are kinds of *Variable*s, which have *TypeOf* edges to refer to their declared types. Likewise, *DataFlow_Node*s also have a *TypeOf* edge, as these represent strongly typed expressions and local variables. Having the same edge even across different layers of the model is useful in this case, because *Variable*s and *DataFlow_Node*s are also connected via *DataFlow_Edge*s as part of the data flow model.

In XCSG, the *TypeOf* edge points to the *Type* representing the declaration. The types could have been represented as string attributes, but wherever possible we have used edges. This accomplishes two goals: first, it encourages the use of a node to represent identity instead of complex string encodings, and second, it normalizes access to entities via traversal instead of by attribute value. Using the value of an attribute of one node to find another node is more awkward to write as a query, as it requires one to remember both the attribute key for the source node as well as a (probably) different attribute key for the target node. Using an edge simply requires one to remember the kind of the edge and its direction.

3.6 Use Edges To Summarize

Call graphs are one of the most frequently used tools when understanding programs. The call relationship is a useful summary of potentially many instances of invocation in a single method.

In an object-oriented language, invocations may be resolved by dynamic dispatch. When viewed from static analysis, invocations may be resolved to many possible targets. At a coarse level, these possible invocations from calling context to target method are summarized in a call graph. In XCSG, these are represented by *Call* edges between *Methods*.

Call edges in XCSG may be created based on many different analyses of varying precision [9, 21]. In 3.8.3, we describe how XCSG represents call sites, which can be used to enable such analyses.

3.7 Pick Your Kinds Well

XCSG describes relationships between kinds, which can be applied to edges as well. This allows the modeling of equivalence classes of edges, which in turn affects transitivity.

For example, the *Supertype* kind has subkinds *Extends* and *Implements*, corresponding to the Java keywords. A traversal using the *Extends* kind would encounter a discontinuity at the transition from *Java.Class* to *Java.Interface*. Using the *Supertype* kind to traverse over both subkinds reaches all related types.

Choosing equivalence classes for edges is especially important in the data flow and control flow models as these are the basis for composing new analyses.

As mentioned earlier, the data flow model connects *Variable*s and *DataFlow_Node*s using *DataFlow_Edge*s to indicate flow. The kind *DataFlow_Edge* has subkinds *LocalDataFlow* and *InterproceduralDataFlow*, to distinguish between local flows and flows involving method invocation or heap access. In practice, a client analysis of the data flow analysis uses the *DataFlow_Edge* kind to traverse both. However, a data flow analysis which creates or refines the possible flows can take advantage of semantics implied by these kinds. Since there are no stack directed pointers in JVM bytecode, local flows can be calculated flow sensitively without an interprocedural analysis. *InterproceduralDataFlow* edges correspond to flows that a points-to analysis may or may not create depending on sensitivity; points-to is still an area of active research [23, 7, 13, 2, 24].

In the next section, we discuss exceptions to transitivity – the edge kinds required to express important roles for instructions, and how those are related to data flow.

3.8 Enable Analyses

The data flow model is by far the most rich in terms of representation. In the remainder of this section, we primarily discuss how the data flow model captures the semantics of instructions, thereby allowing other analyses to be written in terms of XCSG.

3.8.1 Operators and Assignments

The data flow representation has may subkinds, roughly corresponding to the various instructions found in Jimple.

The data flow representation is fine-grained with respect to operators. The operands connect to an *Operator* via *DataFlow_Edge*s, which leads to a straightforward visualization, shown in abstracted form in Figure 2. In the figure, all edges are of the kind *DataFlow_Edge*, but the edges have an extra tag to distinguish left vs. right. The semantics of the operation are captured in the specific subkind of *DataFlow_Node*, in this case *Addition*. Finally, the result flows to another subkind, *Assignment*. The equivalence classes have been carefully chosen to allow queries to pass through based on the common parent kinds, yet retain enough detail to enable fine-grained analyses to use the same representation as a basis. *Assignments* are distinct from *Operator*s, as these represent no transformation of the underlying bits.

*Instantiation*s represent references to new instances and are typically immediately passed to a class initializer.

3.8.2 Blending Layers

Another interesting feature of the data flow representation is the use of a *DataFlowCondition* node to represent the use of a value to influence a condition. In visualizations focused on the data flow layer, flows terminating at

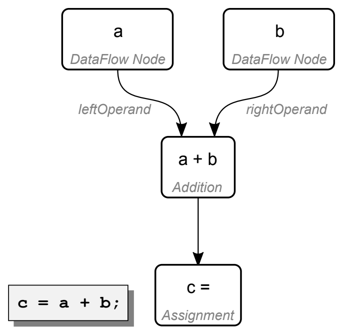


Figure 2: XCSG Operator

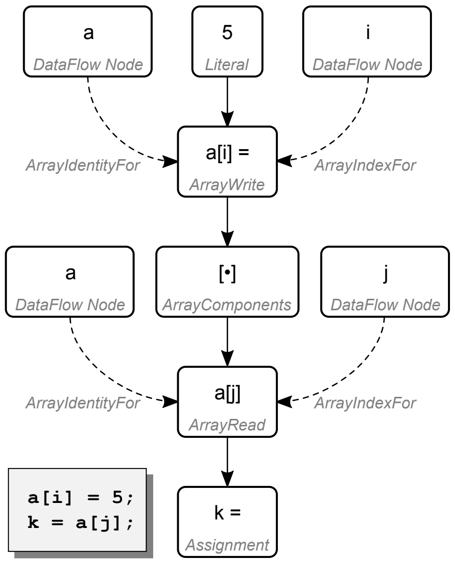
a *DataFlowCondition* are a signal to the human that the current flow could result in information flow. Since the *DataFlowCondition* node is immediately under the *ControlFlowCondition*, it is possible to navigate programmatically from one to the other, and to retrieve the successors in the control flow graph.

In bytecode, synchronization is implemented using instructions which lock and unlock an object, called enter monitor and exit monitor, respectively. In XCSG, monitors are modeled with separate *DataFlow_Node*s much like *DataFlowConditions*, for similar reasons.

3.8.3 Array and Field Access, Call Sites, and Instantiation

The remaining data flow subkinds discussed here are some of the more complicated and interesting ones. Analysis developers interested in points-to analyses should find these particularly relevant. These are analogous to *Operator*s, but are distinct as they generally have unique kinds for operand edges. These operand kinds are not subkinds of *DataFlow_Edge*, to prevent a query for direct data flow from spilling over into all data dependencies automatically.

Array and Field Access Array access is modeled similarly whether in a rvalue or lvalue context, using *ArrayRead* or *ArrayWrite* kinds, respectively, as can be seen in Figure 3. An *ArrayRead* has two named operands, one for the array reference and one for the index expression. These named operand kinds are not *DataFlow_Edge*s, and hence are shown as dashed lines.



```
a[i] = 5;
k = a[j];
```

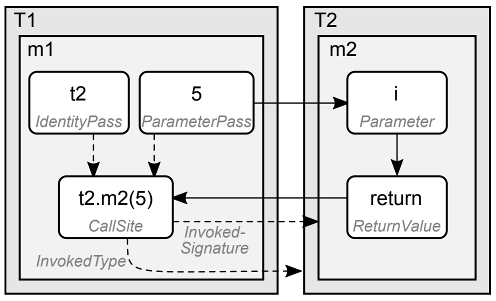
Figure 3: Array access in XCSG

*DataFlow_Edge*s flow through the *ArrayComponents* node, which represents an abstract object on the heap.

Instance field access is analogous, the access nodes having parent kind *InstanceVariableAccess*. The flows, however, connect to the *Field* node instead of an *ArrayComponents* node. Without further qualification of these edges, a simple traversal corresponds to a field-sensitive but heap-insensitive data flow analysis. The *ArrayComponents* node differs from a *Field* in that, to represent flows conservatively, there would have to be a single *ArrayComponents* node for all array accesses. Given a main method, a points-to analysis would produce *ArrayComponents* nodes corresponding to abstract objects, and connect to the *ArrayAccess* nodes via *InterproceduralDataFlow* edges.

Instance fields require a base object operand in addition, and so there is an additional node called an *InstanceVariableAccess* interposed in the flows to serve a role analogous to an *Operator*. The representation varies slightly for an access which is an rvalue vs an lvalue, the former having subkind *InstanceVariableRead*, the latter *InstanceVariableWritten*. Flows into and out of static fields are directly connected to the *Field*, as the parent *Type* is implied by the *Contains* edge between them.

In terms of human affordances, the representation has source correspondence on the access nodes, which means that selections in a text editor correspond to both the field identifier and the point in control flow at which the field was accessed, which may be leveraged by interactive analyses.



```
class T1 {                  class T2 {
  void m1(T2 t2) {            int m2(int i) {
    t2.m2(5);                   return i;
  }                           }
}                           }
```

Figure 4: XCSG CallSite

Call Sites and Array Instantiation A call site is perhaps one of the most sophisticated models, as it represents the combination of an arbitrary number of parameters, as well as a target object in the case of instance methods. Figure 4 illustrates the *CallSite* model. In XCSG, a *CallSite* is a *DataFlow_Node* representing the value returned by invoking the method, similar to an *Operator*; if void, the node still exists as a placeholder for the event occurring. *CallSite*s have subkinds corresponding to the dispatch mode, to distinguish between static and dynamic dispatch. *CallSite*s representing static dispatch are connected via an *InvokedFunction* edge, and dynamic dispatch via an *InvokedSignature* edge to a representative *Method* node. The actual arguments to a *CallSite* have kinds *ParameterPass* (and *IdentityPass*, corresponding to the receiver of an instance method), and are connected to the *CallSite* via edges with kinds *ParameterPassedTo* and *IdentityPassedTo*. These edges have attributes to indicate the index of each argument. An analysis of potential call targets may represent flows to the invoked methods by connecting the actual arguments to the formal arguments via *InterproceduralDataFlow* edges, and likewise from the *Return* nodes of possibly invoked methods back to the *CallSite* node.

The model for *ArrayInstantiation* is somewhat similar to a *CallSite*, but lacks an edge to a method, and supports an arbitrary number of edges corresponding to the expressions establishing the length of each dimension. The node represents the reference to the heap. *Instantiation* of a non-array type is similar, but does not require extra edges.

3.8.4 Array Instantiation

Finally, the allocation of arrays is modeled using an *ArrayInstantiation* node. Both single and multiple dimension arrays are folded into the same representation. *ArrayInstantiation* is somewhat similar to a *CallSite*, but lacks an edge to a method, and has an arbitrary number of edges corresponding to the expression establishing the length of each dimension. The node itself represents the reference to the abstract heap object, which would eventually flow to the array reference edge of an *ArrayAccess*.

In the next section, we elaborate on the query language that we use to select subgraphs for interactive display purposes.

4 Interactive Query

In the previous section, we described how XCSG represents whole programs and certain common program analyses. Although XCSG stands by itself, some of the pressure that influences design choices behind it can be traced to the query language used most often to interact with it.

4.1 Design Considerations

The query language is implemented as an embedded domain-specific Language (DSL) in Java, and follows a builder pattern, taking inspiration from [11, 12]. The rationale behind making a DSL is that it adds a layer of abstraction for expressing what to select from the graph, provides some conciseness of expression, and leaves a layer of indirection permitting query optimization. The rationale behind making it embedded is that it avoids recreating the useful features already present in an imperative language such as Java.

The query language is usually informally referred to simply as *Q*, which is the simple name of the Java interface.[3] Q is used to describe what to select

[3]The single letter name was chosen to reduce the number of keystrokes required to

from a graph, the expression yielding a subgraph. By the builder pattern, almost all methods in the interface Q return an expression of type Q, and chaining method calls effectively specifies the query AST. A chain of Q expressions can be evaluated by calling the `eval()` method, which transitions to a graph API suitable for imperative implementations. Methods of Q are responsible for ensuring that the subgraph returned is a proper graph, where edges are present iff the incident nodes are as well. From the perspective of Q, the entire graph database is an expression called `universe()`. Query results are therefore confined to returning subgraphs of the *universe*.

The primary use case behind the design of Q is enabling an analysis designer to quickly draft single line queries to select relevant portions of the graph – in essence, enabling them to "look around the corner" from their current position, and bring together related but lexically non-local elements of the program. Unlike many other graph query languages [3, 32], Q deliberately unions all matches at each step. For example, given several *Method* nodes as an origin, a query for a call graph returns a single unified graph, as opposed to returning individual matches to a pattern.

One of the first considerations in the design of Q was making it relatively easy to express traversals of the graph, particularly call graphs. We introduce the semantics of Q gradually, in terms of examples.

First, we note that kinds are denoted using string constants from the XCSG interface, e.g. `XCSG.Method` is a string constant representing a *Method* kind. Queries for XCSG kinds are implicitly handled by the underlying graph database such that any subkinds are also included in the result. Further, the underlying graph database indexes graph elements by their kinds and attribute values, for better performance. For the remainder of this section, we will assume the string constants are statically imported, for succinctness of expression.

In Q, the expression
`universe().nodesTaggedWithAny(Method)` selects the subgraph consisting of all *Method*s, and no edges. This shows how the builder notation is used to chain together queries, where `universe()` is the entire graph database, and `nodesTaggedWithAny` selects matching nodes from the expression on the left.

A more interesting example is a forward call graph, which can be expressed as `edges(Call).forward(x)`, where `x` is a previous query for

write queries in Java.

the origin *Method*s. But this expression is hiding a lot of behavior that we have to unpack to fully explain what it does.

First, `edges(Call)` is a statically imported convenience method equivalent to
`universe().edgesTaggedWithAny(Call)`. This query
`edgesTaggedWithAny(Call)`, selects edges of kind *Call*. [4] However, a subtle point is that this selection process retains all nodes from the expression on the left. As a result, the query
`edgesTaggedWithAny` keeps every node from the universe. This is by design: if we are trying to build a call graph from a given method, we usually expect to see that method in the result, even if it is not connected. As implemented, this works as expected. Alternatively, if edge selection had been implemented to first filter nodes by connectivity, and if the method happened to be unconnected by *Call* edges, it would lead to the counter-intuitive result of an empty call graph.

The composition of the builder notation raises another point of the design: the input (the graph on the left of the expression) can be an arbitrary subgraph of the universe. Since Q is embedded in Java, it can be composed with algorithms written in an imperative style, not necessarily written using Q. In the implementation we use, there is a lower level API oriented around graph data structures, with direct access to the nodes and edges. This lower level graph API has been used to write algorithms for calculating dominators, strongly connected components, transitive closure, and other algorithms. These algorithms can be interleaved with Q as needed, providing a path to evolving from queries that start out as a single line, to sophisticated program analyses.

4.2 Query Expressions and Use Cases

Query expressions in Q correspond to methods on the interface, and can be roughly categorized as providing 1) selection by tags or attributes, 2) set-based expressions, 3) traversals.

We give a brief overview of each category, pointing out unusual aspects and typical uses.

[4] *Tags* are the way that XCSG *kinds* are encoded in the implementation; we use the terms interchangeably.

4.2.1 Selection Expressions

The selection expressions are, of course, expected because they are necessary to match nodes and edges for other queries. Selection by tags and attributes work generally as expected, with the following caveats. First, tags are allowed to have relationships to one another, such that selection by one tag implies that others should also be selected. This was used to implement XCSG, which specifies relationships between kinds such that a kind such as *InstanceMethod* is also a *Method*. Second, selections may pertain to either nodes or edges; when selecting nodes all edges are excluded, and when selecting edges all nodes are retained. This was intentional, as described in 4.1, to lean towards making creation of traversals for things like call graph relatively straightforward.

In addition to selection by tags and attributes, there are a few convenience expressions for selecting common named entities in XCSG. For example, `universe().methods("foo")` selects all methods named *foo*. This could otherwise be written as `universe().nodesTaggedWithAny(Method) .selectNode(name, "foo")`. Note that chaining expressions in this case is effectively an intersection of the tag and the attribute value.

4.2.2 Set Expressions

Set expressions include `union`, `intersection`, and `difference`. Union and intersection can be thought of as operations on the underlying node and edge sets of the graphs represented by the query expressions. In general, graphs obtained from Q never contain duplicate references to underlying graph elements, as the underlying graphs are comprised of sets of nodes and sets of edges.

The `difference` expression is peculiar in that it necessarily excludes edges incident on removed nodes; in practice we have found that `difference` is best used when the query expressions correspond to graphs with only nodes. However, there is a `differenceEdges` expression for specifying that only edges be removed from the expression on the left.

In addition to the set expressions, the expression `a.induce(b)` adds edges from `b` to `a`, where an edge is added iff `a` contains both of the nodes it is incident on. For example, the call graph edges between given methods can be induced by `methods.induce(edges(Call))`.

4.2.3 Traversal Expressions

In principle, we need only a single traversal expression that takes direction and path length. In practice, there are many for convenience.

The expression `forward` was previously described in 4.1; it essentially returns the reachable subgraph starting from the given nodes, along the direction of the edges. The expression `reverse` is the same, but walks against the direction of the edges.

Traversal expressions are essentially a cross product of direction and path lengths of 1 step, n steps, and infinite steps. In addition, `between` is logically equivalent to the intersection of `forward` and `reverse`. The `betweenStep` expression is useful for finding immediate relationships between two subgraphs. For example,
`edges(Call).betweenStep(a,b)` gives the induced call graph between the methods in `a` and `b`, where the edges must start in `a` and end in `b` in a single step.

Probably the most common traversals are along the *Contains* edges, as these can be used to give lexical context to the query result, or to select the subgraph under a package, type or method. The composability of the queries makes it easy to put the elements corresponding to a complex analysis results in context; given an arbitrary subgraph x, the enclosing lexical entities can be added with the expression `edges(Contains).reverse(x).union(x)`.

In addition to the syntax already shown, there are alternate forms of the traversal expressions which simply swap the operands. For example, a call graph can also be expressed as
`x.forwardOn(edges(Call))`. In practice this form is sometimes more convenient as one usually starts by selecting nodes, then expecting to traverse along some subset of edges. This form puts the nodes first, making it more natural to use the builder notation to continue a previous expression.

All of the traversal expressions discussed up until this point are designed to include the origin nodes in the result of the query (the rationale given in 4.1), but this is not always a desired behavior. There are many occasions where including the origin nodes in the result is awkward or more verbose; one simple example is in obtaining the immediate members of a type, exclusive of the type itself. For this reason, the expressions `predecessors` and `successors` exist to return the immediate predecessors or successors of the given nodes.

4.2.4 Example Use Cases

For a flavor of how queries are used in practice, we offer a few short examples.

First, data flow is traversed similarly to call graphs, using the query in Listing 1, which illustrates how to get a forward data flow graph from a given variable.

```
Q dataFlow = edges(DataFlow_Edge).forward(someVariable);
```

Listing 1: Data Flow Query

Second, a complete control flow graph for a method can be obtained using the queries in Listing 2. The query starts by matching everything under the method, filters by control flow nodes, then induces the control flow edges.

```
Q body = edges(Contains).forward(someMethod);
Q cfgNodes = body.nodesTaggedWithAny(ControlFlow_Node);
Q cfg = cfgNodes.induce(edges(ControlFlow_Edge));
```

Listing 2: Control Flow Graph Query

5 Case Study: Enabling Intelligence Amplification

In this section, we demonstrate how a new analysis can be integrated into XCSG and leverage the existing graph representation to 1) facilitate human-machine collaboration and 2) facilitate composition with other analyses, together enabling the goal of intelligence amplification.

Precursors of the XCSG graph representation have been used to build tools for interactive program analysis of Android apps, specifically for searching for sophisticated, novel malware which defies a priori description, making completely automatic detection difficult or impossible [16].

Since then, XCSG has been used to develop sophisticated analysis techniques to aid in the verification of critical safety and security properties of the Linux kernel [25], with an approach called *Evidence-Enabled Collaborative Verification* (EECV). This approach was applied to three versions of the Linux kernel, totaling 37MLOC. To date, the approach has found 8 confirmed bugs in the Linux kernel. Details of the complete set of techniques used to scale up verification, specifically, the classification techniques used to partition the verification instances are presented in [19, 26].

In this section, we present a case study based on one of the core concepts behind the EECV approach: the *Projected Control Graph* (PCG). We present the PCG model because of its value for human-machine collaboration: in the process of auditing large code bases, it is important to focus the human's attention on the key aspects of the program. One way to accomplish this is through the PCG.

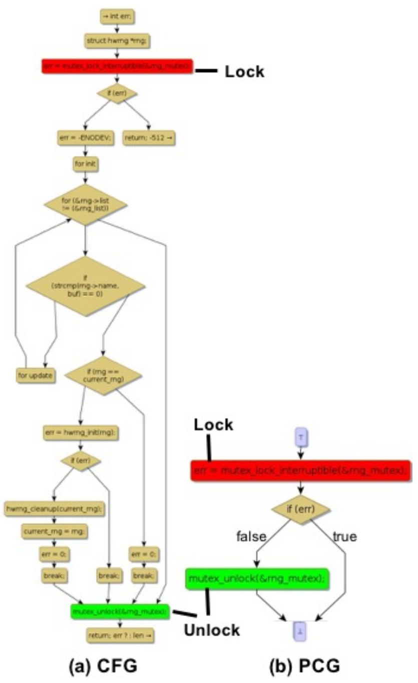


Figure 5: Example CFG and PCG with Respect to Events

The PCG is a compact derivative of the Control Flow Graph (CFG), defined with respect to a set of problem-specific program artifacts. Mathematically, it can be precisely defined using the homomorphism concept from abstract algebra. We refer to [26] for mathematical details of PCG. PCG is designed to address two important verification challenges: (a) exponential growth of number of paths in CFG, and (b) the exponential complexity of checking path feasibility. PCG retains only the nodes necessary to retain all distinct traces, but there is one and only one path corresponding to each distinct trace. A CFG may have a very large number of distinct paths, but only a very small number of distinct traces. PCG optimizes the verification of each and every path by by retaining only one path per trace. [26] presents an efficient algorithm to transform a CFG to an PCG in linear time with respect to nodes and edges. Figure 5 compares the two with respect to selected events in the CFG, corresponding to lock and unlock events.

We have implemented an algorithm for calculating an PCG in terms of XCSG, and reified the resulting graph in the XCSG graph database. The XCSG-based PCG model is calculated using the control flow graph, and takes *ControlFlow_Node*s to specify relevant program artifacts. Parameterization in terms of program artifacts allows a PCG to be composed with existing analyses – other analyses can perform arbitrary calculations to arrive at relevant program artifacts. Or, it can take advantage of existing affordances and be driven by user selections of text or nodes (as described in Section 3), allowing the user to quickly construct PCGs based on arbitrary program artifacts. These capabilities enable an analysis designer to quickly prototype new ideas for analysis using the user interface and query language to get quick first approximations of interesting events, and over time, develop powerful analyses which calculate the events automatically.

There are other interesting ways we leveraged XCSG when integrating the PCG analysis.

First, to make it easier to select a wider range of program artifacts, we added a small bit of code to walk from a *DataFlow_Node* up to the enclosing *ControlFlow_Node*. This allows us to quickly inspect the PCG with respect to *DataFlow_Node*s such as an *Assignment*. With a click, we can quickly tell whether the selected node is within a loop. If we select multiple nodes, we can see how their execution traces are related.

Second, PCGs as applied to Linux in C did not have to consider exceptional control flow, as an analysis in Java might. Because of the way XCSG models *ExceptionalControlFlow_Edge*s, we had the opportunity to quickly run an experiment where the PCGs take as input both normal and exceptional control flow. We were able to trivially parameterize our initial algorithm for producing PCGs to make calculations on any given subgraph, making it easy to toggle inclusion of exceptional control flow edges. Because the semantics of an exceptional control flow edge differ from a regular control flow edge in that the relevant program statement may or may not have executed before the exception was raised, we are still in the process of working through implications for other analyses that would build off of our PCG representation. However, we were delighted that we could start experimenting almost immediately.

Third, once we implemented support for selection of *DataFlow_Node*s, we realized that we could create a view which crosses data flow traversal with an PCG. There are already interactive views which step along data flow, which are useful in themselves. But often one encounters interesting events along these data flows, and the next question becomes "which of these statements are executed first?" To answer this question immediately, we can manually select these data flow nodes to define relevant program artifacts for a PCG. But if we decide this view is useful, with just a little more effort we can create a new view which automatically projects the PCG with respect to the events implied by data flow graph traversal.

In these ways, XCSG has made it easier for analysis developers to take advantage of affordances for collaboration with the human to solve hard problems, as well as enabled composition through the graph, all to help enable intelligence amplification.

Our long term research goal is to apply PCGs to new classes of analysis problems, using XCSG as a base.

6 Implementation and Performance

To give a sense of performance, we give some statistics about the translation process as well as an algorithm that runs as a post-processing step using XCSG itself as input. Programs are first translated into Jimple, then translated to XCSG using version 2.6.1 of Atlas, available at [1]. Experiments were performed on a 1.8 GHz quad core Intel Xeon CPU (E5-2603 v2) with 128GB RAM and running Ubuntu 14.04.3 LTS.

Two programs were translated: OpenJDK 7u80_b32, and a small "hello world" program. The results are shown in Table 1. We used Class Hierarchy Analysis (CHA) [9] to produce conservative estimates for *Call* and *InterproceduralDataFlow* edges.

When converted to Jimple text, the OpenJDK is over 4.2 million non-blank lines. The translation of OpenJDK takes 77 minutes to complete and takes 16GB in RAM to store the full in-memory graph. 20,925 types in OpenJDK 7 were translated.

For contrast, we also translated a "hello world" – nearly the smallest possible Java program. For this program, we used SPARK [20] from [4] to conservatively calculate which OpenJDK API classes might be loaded during a run of this program, and translated only those classes. This included 2,505 types.

As part of ongoing experiments, we implemented the loop identification algorithm from [29], which analyzes the XCSG control flow model and annotates it with results. This analyzer takes less than a minute to run on the OpenJDK.

	XCSG Kinds	All OpenJDK	HelloWorld + Subset of OpenJDK
Node	Method	171,279	21,002
	Type	20,925	2,505
	Variable	408,317	44,879
	ControlFlow	2,676,789	297,214
	DataFlow	7,853,827	833,740
	Other	101,912	12,580
	Total	**11,233,049**	**1,211,920**
Edge	Call	3,611,359	128,172
	InterproceduralDataFlow	11,945,382	463,461
	Other	32,497,748	3,505,174
	Total	**48,054,489**	**4,096,807**
Translation Time		77 minutes	4 min
In-Memory Database		16GB	2GB

Table 1: XCSG Representation in Terms of Graph Elements

7 Related Work

7.1 Graphs in Program Analysis

Graphs are clearly central to XCSG. We have already discussed work related to graph representations of programs in Section 2. As many interesting analyses have already been developed in terms of graphs, we are interested in exploring how such analyses could be integrated into XCSG in ways that enable composition of analyses while ensuring that the same representation facilitates human-machine collaboration.

One such analysis was presented in Section 5, based on *Projected Control Graphs* (PCGs) [26]. PCGs have some relationship to the *control dependence* edges defined by PDGs [10]; given control dependence edges, it should be possible to construct an PCG. However, we hasten to note that a graph in terms of control dependence edges is distinct from a PCG: PCG is derived using a homomorphic mapping from a Control Flow Graph (CFG), with semantics which should be immediately familiar to anyone familiar with CFGs. PDG on the other hand is derived using the concept of dominator relationships. PCG allows us to create a much smaller graphs to capture the essential control flow behavior, focused with respect to a specific set of program artifacts. This focus is an important aspect, and will drive our future research.

7.2 Graph Query in Program Analysis

[27] reported on a prototype which imports Java programs into a Neo4J [3] graph database. They put forward a concept of *overlays*, examples including structure and data flow. They noted that many queries would inherently use information from multiple overlays, and gave several examples in terms of Neo4J. The prototype represents the program by translating Abstract Syntax Tree (AST) nodes more or less directly from the internals of javac (for example, a method declaration in the graph database would be given the name of the corresponding AST node in javac: "JCMethodDecl"). Compared to [27], XCSG has much more in common with PDGs and SDGs, as instruction-level detail is captured via the control and data flow representation. In addition, XCSG aims to normalize the nomenclature used and specifies relationships between kinds.

[15] presents a tool for querying large C/C++ programs, also implemented using [3]. They note that, while they feel their current model allows for a large number of succinctly expressed queries, as of Neo4J version 2, the node label feature would allow them to express relationships between kinds of nodes, allowing further succinctness. However, Neo4J does not currently support multiple labels on edges, which is a feature leveraged by XCSG. They note a possible workaround, but add that "specifying matches in general becomes at best less succinct and at worst impossible"

[33] describes an approach for discovering vulnerabilities by representing the ASTs, CFGs, and PDGs, again in a Neo4J graph database. They demonstrate that their approach can discover vulnerabilities in the Linux kernel, but note that detection is limited by the boundaries of static analysis. By contrast, XCSG is designed to enable a human and a machine to overcome immediate limitations, and to enable research into automation for program analysis.

Of all the related work, [18] may be the closest in spirit. Among their contributions is, "the novel insight that PDGs offer a unified approach that enables *exploration*, *specification*, and *enforcement* of security guarantees." With respect to the exploration aspect, they note that this is helpful when programs have no predefined security policies, necessitating discovery of application-specific ones. For many hard program verification problems, we expect to require such exploration, and this has greatly influenced our approach with XCSG. They also have a domain-specific language (DSL) called PidginQL. The grammar of PidginQL as described is very similar to the *Q* query language described in Section 4; one major distinction is that Q is implemented as an embedded DSL, making integration with Java code relatively seamless. Some of the user defined functions they describe are implemented in Java, using Q as a convenience for accessing the graph.

8 Conclusion

We have presented the eXtensible Common Software Graph, a humane, graph-based representation of whole programs and related analyses. XCSG has been designed to enable human-machine collaboration to tackle difficult verification problems in multi-million line programs. It accomplishes this not only by including a complete representation of the program semantics, but more importantly through careful choices about the equivalence classes of both nodes and their relationships, blending layers of analyses to enable more sophisticated queries. It is these equivalence classes which directly influence the ease and succinctness of human interactions with the program model, enabling more efficient human-machine collaboration.

Ultimately, human-machine collaboration must be enabled to tackle difficult problems now, and to help inspire creative solutions to automate or semi-automate verification in the future. We have presented a case study based on the underlying analysis used to verify critical safety and security properties in the Linux kernel to illustrate how a new analysis can be blended into XCSG, thereby leveraging the existing affordances for human collaboration and enabling composition with other analyses. As new analyses are blended into XCSG, they further enhance the ability of the human to compose analyses in previously unanticipated ways, thereby *amplifying* intelligence through human-machine collaboration. Maturing XCSG would not be possible without practitioners. We would like to thank the many post docs, graduate students, and undergraduate students at Iowa State University and our colleagues at EnSoft who used XCSG and it's many drafts during the DARPA APAC program and those who continue under the DARPA STAC program. In particular we would like to thank Tom Deering and Ben Holland for their many detailed suggestions to improve XCSG and the creative ways in which they have used it. We would also like to thank Nikhil Ranade and Kevin Korslund for their support in preparing this paper.

References

[1] Atlas Website.

[2] DOOP: Framework for Java Pointer Analysis.

[3] Neo4j Graph Database.

[4] Soot - A Java optimization framework.

[5] XCSG - Extensible Common Software Graph.

[6] F. E. Allen. Program optimization. *Annual Review in Automatic Programming*, 5, 1969.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[8] F. P. Brooks Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.

[9] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[11] M. Fowler. *Domain-specific languages.* Pearson Education, 2010.

[12] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 855–865, New York, NY, USA, 2006. ACM.

[13] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow analysis of Android applications in DroidSafe. In *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[14] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Secur.*, 8(6):399–422, Oct. 2009.

[15] N. Hawes, B. Barham, and C. Cifuentes. Frappé: Querying the Linux kernel dependency graph! In *Proceedings of the GRADES'15*, page 4. ACM, 2015.

[16] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade. Security toolbox for detecting novel and sophisticated Android malware. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 733–736, Piscataway, NJ, USA, 2015. IEEE Press.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[18] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 291–302, New York, NY, USA, 2015. ACM.

[19] S. Kothari, P. Awadhutkar, A. Tamrawi, and J. Mathews. Modeling lessons from verifying large software systems for safety and security. In *2017 Winter Simulation Conference (WSC)*, pages 1431–1442. IEEE, 2017.

[20] O. Lhoták. Spark: A flexible points-to analysis framework for Java. 2002.

[21] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, Oct. 2008.

[22] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 177–184, New York, NY, USA, 1984. ACM.

[23] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. *SIGPLAN Not.*, 46(1):17–30, Jan. 2011.

[24] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM.

[25] A. Tamrawi and S. Kothari. Evidence-enabled collaborative verification of software for safety and cybersecurity. *Submitted to IEEE International Conference on Software Testing, Verification and Validation (ICST 2016).*

[26] A. Tamrawi and S. Kothari. Projected control graph for computing relevant program behaviors. *Science of Computer Programming*, 163:93–114, 2018.

[27] R.-G. Urma and A. Mycroft. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming*, 97:127–134, 2015.

[28] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. Softw. Eng.*, 5(3):237–247, May 1979.

[29] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In H. Nielson and G. Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer Berlin Heidelberg, 2007.

[30] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[31] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[32] P. T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, Apr. 2012.

[33] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 590–604, Washington, DC, USA, 2014. IEEE Computer Society.