Exploring and enforcing security guarantees via program dependence graphs

Exploring and Enforcing Security Guarantees via Program Dependence Graphs

Andrew Johnson
MIT Lincoln Laboratory and Harvard University, USA
ajohnson@seas.harvard.edu

Lucas Waye Scott Moore
Stephen Chong
Harvard University, USA
lwaye, sdmoore, chong@seas.harvard.edu

Abstract

We present PIDGIN, a program analysis and understanding tool that enables the specification and enforcement of precise application-specific information security guarantees. PIDGIN also allows developers to interactively explore the information flows in their applications to develop policies and investigate counter-examples.

PIDGIN combines *program dependence graphs* (PDGs), which precisely capture the information flows in a whole application, with a *custom PDG query language*. Queries express properties about the paths in the PDG; because paths in the PDG correspond to information flows in the application, queries can be used to specify global security policies.

PIDGIN is scalable. Generating a PDG for a 330k line Java application takes 90 seconds, and checking a policy on that PDG takes under 14 seconds. The query language is expressive, supporting a large class of precise, application-specific security guarantees. Policies are separate from the code and do not interfere with testing or development, and can be used for security regression testing.

We describe the design and implementation of PIDGIN and report on using it: (1) to explore information security guarantees in legacy programs; (2) to develop and modify security policies concurrently with application development; and (3) to develop policies based on known vulnerabilities.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls; F.3.2 [*Programming Languages*]: Semantics of Programming Languages—Program analysis; F.3.1 [*Programming Languages*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

Keywords Application-specific security, program dependence graph, graph query language

1. Introduction

Many applications store and compute with sensitive information, including confidential and untrusted data. Thus, application developers must be concerned with the information security guarantees their application provides, such as how public outputs may reveal confidential information and how potentially dangerous operations may be influenced by untrusted data. These guarantees will necessarily be application specific, since different applications handle different kinds of information, with different requirements for the correct handling of information. Moreover, these guarantees are properties of the entire application, rather than properties that arise from the correctness of a single component.

Current tools and techniques fall short in helping developers address information security. Testing cannot easily verify information-flow requirements such as "no information about the password is revealed except via the encryption function." Existing tools for information-flow security are inadequate for a variety of reasons, since they either unsoundly ignore important information flows, require widespread local annotations, prevent functional testing and deployment, or fail to support the specification and enforcement of application-specific policies.

We present PIDGIN, a system that uses *program dependence graphs* (PDGs) [16] to precisely and intuitively capture the information flows within an entire program[1] and a *custom PDG query language* to allow the exploration, specification, and enforcement of information security guarantees. PDGs express the control and data dependencies in a program and abstract away unimportant details such the sequential order of non-interacting statements. They are a great fit for reasoning about information security guarantees, since paths in the PDG correspond to information flows in the application. Our queries express properties of PDGs which thus correspond to information-flow guarantees about the application. Our approach has several benefits:

- PIDGIN *security policies are expressive, precise, and application specific*, since they are queries in a query language designed specifically for finding and describing information flows in a program. Queries can succinctly express *global* security guarantees such as noninterference [18], absence of explicit information flows, trusted declassification [24], and mediation of information-flow by access control checks.
- Developers can *interactively explore an application's information security guarantees*. If there is no predefined security specification then PIDGIN can be used to quickly explore security-relevant information flows and discover and specify the precise security policies that an application satisfies. If a policy is specified but not satisfied, then PIDGIN can help a developer understand why by finding information flows that violate the policy.
- PIDGIN *security policies are not embedded in the code.* PIDGIN policies are specified separate from the code. The code doesn't require program annotations nor does it mention or depend on PIDGIN policies. This enables the use of PIDGIN to specify secu-

[1] PDGs for a whole program are also called *system-dependence graphs* [26].

PLDI'15, June 13–17, 2015, Portland, OR, USA.

ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737957

rity guarantees for legacy applications without requiring annotations or other modifications.
- *Enforcement of security policies does not prevent development or testing.* Because the program code does not mention or depend on PIDGIN policies, the policies do not prevent compilation or execution. This allows developers to balance development of new functionality and maintenance of security policies.
- PIDGIN *enables regression testing of information security guarantees.* PIDGIN can be incorporated into a build process to warn developers if recent code changes violate a security policy that previously held. This includes information-flow properties that traditional test cases can not easily detect.

These benefits stand in contrast to existing techniques such as security-type systems (e.g., [58], Jif [41] and FlowCaml [49]) and existing PDG-based approaches to security (e.g., JOANA [21]).

In security-typed languages, global security policies are broken into many pieces and expressed via annotations throughout the program. This is problematic for at least three reasons. First, it is difficult to determine from these annotations how sensitive information is handled by the whole system, particularly in the presence of declassification [48]. Second, changing the security policy may require modifying many annotations. Third, supporting legacy applications using these techniques is often infeasible, as they require significant annotations or modifications to applications.

Dynamic or hybrid information-flow enforcement mechanisms (e.g., [3, 4, 10, 27, 31, 53]) are sometimes able to specify security policies separate from code, but interfere with the deployment of systems: they must be used during testing in order to ensure that enforcement does not conflict with important functionality.

Taint analysis tools (e.g., [11, 15, 32, 55, 56, 61]) are inevitably unsound because they do not account for information flow through control channels, and often do not support expressive application-specific policies. One of the most recent, FlowDroid [2], works with a pre-defined (i.e., not application-specific) set of sources and sinks and does not support sanitization, declassification, or access control policies. Because PIDGIN supports more expressive policies, we detect 159 of the 163 (=98%) vulnerabilities in the SecuriBench Micro [36] 1.08 test suite compared to Flowdroid's 117 (=72%).

Pre-defined policies (such as policies that might be enforced on all Android apps) can capture many security requirements of broad classes of applications. However, applications handle different types of sensitive information (e.g., bank account information, health records, school records, etc.) and what constitutes correct handling of this information differs between applications. Predefined policies cannot express these application-specific security requirements.

Previous PDG-based information security tools (e.g., [21, 22]) have many of the same issues as security-typed languages. For all but the simplest security policies, these tools require program annotations to specify policies, with the concomitant issues regarding legacy applications, modifying security policies, and understanding the system-wide security guarantees implied by the annotations.

Moreover, these techniques focus almost exclusively on enforcement of security guarantees and do not support exploration.

The primary contributions of this work are:
1. The novel insight that PDGs offer a unified approach that enables *exploration*, *specification*, and *enforcement* of security guarantees.
2. The design of an expressive language for precise, application-specific security policies, based on queries evaluated against PDGs.
3. The realization and demonstration of these insights and techniques in an effective and scalable tool.

PIDGIN produces PDGs for Java bytecode and then evaluates queries against these PDGs, either interactively or in batch mode. Our techniques are applicable to other languages.[2]

PIDGIN is both useful and scalable. We have used PIDGIN to discover diverse information security guarantees in legacy Java applications, and to specify and enforce information security policies as part of the development process for two new applications. We have analyzed programs ranging in size up to 330,000 lines of code (including library code); even for the largest program, construction of the PDG (including pointer analysis and dataflow analyses to improve precision) takes 90 seconds, and checking each of our policies on the PDG takes less than 14 seconds.

Security guarantees we have established using PIDGIN include: in a password manager, the master password is not improperly leaked; in a chat server application, punished users are restricted to certain kinds of messages; and in a course management system, the class list is correctly protected by access control checks. Moreover, we have developed security guarantees based on reported vulnerabilities in Apache Tomcat, and PIDGIN verifies that the security guarantees hold after the vulnerability is patched and fail to hold in earlier versions.

2. PIDGIN By Example

Consider the Guessing Game program presented in Figure 1a. This program randomly chooses a secret number from 1 to 10, prompts the user for a guess, and then prints a message indicating whether the guess was correct.

A program dependence graph (PDG) representation of this program is shown in Figure 1b. Shaded nodes are *program-counter nodes*, representing the control flow of the program. All other nodes represent the value of an expression or variable at a certain program point. There is a single summary node representing the formal argument to the `output` function. There are three nodes representing actual arguments, one for each call to `output`, and an edge from each to the formal argument. Edges labeled CD indicate control dependencies and other edges indicate data dependencies. Dashed edges and clouds show where we have elided parts of the PDG for clarity. (All other emphasis is for the exposition below. Program-counter nodes that are not relevant to the discussion have been removed for simplicity.)

Although Guessing Game is simple, it has interesting security properties that can be expressed as queries on the PDG.

No cheating! The program should not be able to cheat by choosing a secret value that is deliberately different from the user's guess. That is, the choice of the secret should be independent of the user's input. This policy holds if the following PIDGINQL query returns an empty graph:

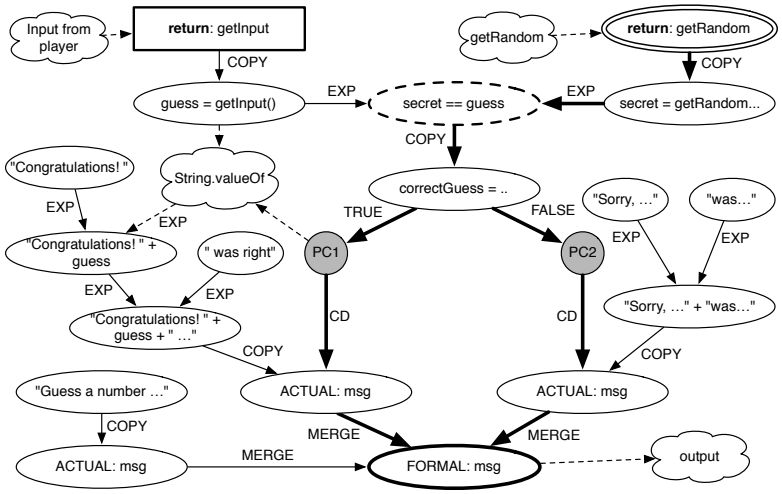
```
let input = pgm.returnsOf("getInput") in
let secret = pgm.returnsOf("getRandom") in
pgm.forwardSlice(input) ∩ pgm.backwardSlice(secret)
```

PIDGINQL is a domain specific graph query language that enables exploration of a program's information flows, and specification of information security policies. Constant pgm, short for *program*, is bound to the PDG of the program. Primitive expressions (such as forwardSlice) compute a subgraph of the graph to the left of the dot. Query expression pgm.returnsOf("getInput") evaluates to the node in the program PDG that represents the value returned from function `getInput` (shown in a rectangle in Figure 1b). This is the user's input. Similarly, the second line identifies the value re-

[2] We have generated PDGs for C/C++ programs by analyzing LLVM bitcode [30] produced by the clang compiler (`http://clang.llvm.org/`), and explored information security in these programs using the same query language and query evaluation engine. This paper focuses on our Java tool.

```
1  secret = getRandom(1, 10);
2
3  output("Guess a number "+
4    "between 1 and 10");
5  int guess = getInput();
6
7  bool correctGuess = (secret == guess);
8  if (correctGuess) {
9      output("Congratulations! "+
10       guess + " was right");
11 } else {
12     output("Sorry, your guess" +
13        " was incorrect");
14 }
```

(a) Guessing Game program



(b) PDG for Guessing Game program

Figure 1: Guessing Game program and simplified PDG

turned from function `getRandom` as the secret. This node is outlined with a double circle in the PDG.

Query expression pgm.forwardSlice(input) evaluates to the subgraph of the PDG that is reachable by a path starting from the query variable input. This is the subgraph that depends on the user input, either via control dependency, data dependency, or some combination thereof. Similarly, pgm.backwardSlice(secret) is the subgraph of the PDG that can reach the node representing the secret value. The entire query evaluates to the intersection of the subgraphs that depend on the user input and on which the secret depends, i.e., all paths from the user input to the secret.

For the PDG in Figure 1b, this query evaluates to an empty subgraph. This means that there are no paths from the input to the secret, and thus the secret does not depend in any way on user input.

Finding all nodes in the PDG that lie on a path between two sets of nodes is a common query, and we can define it as a reusable function in PIDGINQL as follows:

```
let between(G, from, to) =
  G.forwardSlice(from) ∩ G.backwardSlice(to)
```

This allows us to simplify our query. We can also turn our PIDGINQL query into a security policy (i.e., a statement of the security guarantee offered by the program) by asserting that the result of this query should be an empty graph. This is done in PIDGINQL by appending "is empty" to the query.

Noninterference Noninterference [18, 47] requires that information does not flow from confidential inputs to public outputs. For our purposes, the secret number (line 1, Figure 1a) is a confidential input, and output statements (lines 3, 9, 12) are publicly observable.

We can check whether noninterference holds between the secret and the outputs using a query similar to the one above:

```
let secret = pgm.returnsOf("getRandom") in
let outputs = pgm.formalsOf("output") in
pgm.between(secret, outputs)
```

Unlike our previous example the query does not result in an empty subgraph: there are paths from the secret to the output (marked in Figure 1b with bold lines). Indeed, this program does not satisfy noninterference, as the functionality of this program requires that some information about the secret is released.

From secret to output By characterizing all paths from the secret to the output we can provide a guarantee about what the program's public output may reveal about the secret.

Inspecting the result of the noninterference query above, we see there are two paths from the secret to the public outputs. (If there were many paths, we could have isolated one path to examine, by changing the last line to pgm.shortestPath(secret, outputs).) Both paths pass through the node for the value of expression "`secret == guess`". This means that the public output depends on the secret only via the comparison between the secret and the user's guess. We can confirm this by removing this node from the graph and checking whether any paths remain between the secret and the outputs. This can be expressed in PIDGINQL as:

```
1 let secret = pgm.returnsOf("getRandom") in
2 let outputs = pgm.formalsOf("output") in
3 let check = pgm.forExpression("secret == guess") in
4 pgm.removeNodes(check).between(secret, outputs)
5 is empty
```

Expression pgm.forExpression("secret == guess")[3] evaluates to the node for the conditional expression (outlined in Figure 1b with a dotted line). The fourth line removes this node from the PDG then computes the subgraph of paths from secret to outputs.

This query results in an empty subgraph, meaning we have described all paths from secret to outputs. Thus the program satisfies the policy: *The secret does not influence the output except by comparison with the user's guess.*

This is an example of trusted declassification [24] and is a pattern found in many applications. We capture this with a user-defined policy function asserting that all flows from srcs to sinks pass through a node in declassifiers.

```
let declassifies(G, declassifiers, srcs, sinks) =
G.removeNodes(declassifiers).between(srcs, sinks) is empty
```

Note that our policy is weaker than noninterference: the output does depend on the secret. Noninterference is too strong to hold in many real programs, and weaker, application-specific guaran-

[3] For presentation reasons we refer to the specific Java expression "secret == guess". In a more realistic example, a policy would likely refer instead to a function or class, which is less brittle with respect to code changes. However, the ability to refer to specific expressions allows developers to precisely specify queries and policies if needed.

tees are common. PDGs often contain enough structure to characterize these (potentially complex) security guarantees, which can be stated succinctly and intuitively given an expressive language to describe and restrict permitted information flows.

3. PDGs and Security Guarantees

PIDGIN allows programmers to explore a program's information flows and to express and enforce security policies that restrict permitted information flows. We achieve this using *program-dependence graphs* (PDGs) [16] to explicitly represent the data and control dependencies within a program. PIDGIN's PDGs represent control and data dependencies within a whole program. Annotations and meta-information encoded in PIDGIN PDGs enable precise and useful queries and security policies. In this section, we describe the structure of PIDGIN's PDGs and the different kinds of security guarantees that can be obtained from them.

3.1 Structure of PIDGIN PDGs

There are several kinds of nodes in PIDGIN PDGs. *Expression nodes* represent the value of an expression, variable, or heap location at a program point. *Program-counter nodes* represent the control flow of a program, and can be thought of as boolean expressions that are true exactly when program execution is at the program point represented by the node. In addition, *procedure summary nodes* facilitate the interprocedural construction of the PDG by summarizing a procedure's entry point, arguments, return value, etc. Finally, *merge nodes* represent merging from different control flow branches, similar to the use of phi nodes in static single assignment form [13]. Nodes also contain metadata, such as the position in the source code of the expression a node represents.

PIDGIN PDGs are context sensitive, object sensitive, and field sensitive. They are flow sensitive for local variables and flow insensitive for heap locations.

Edges of the PDG indicate data and control dependencies between nodes. To improve precision and enable more complex queries, edges in PIDGIN PDGs have labels that indicate *how* the target node of the edge depends on the value represented by the source node of the edge. Examples of these edge labels can be seen in Figure 1b. COPY indicates that the value represented by the target is a copy of the source. EXP indicates that the target is the result of some computation involving the source. Edges labeled MERGE are used for all edges whose target is a merge or summary node.

Label CD indicates a control dependency from a program-counter node to an expression node. An expression is control dependent on a program-counter node if it is evaluated only when control flow reaches the corresponding program point. An edge labeled TRUE or FALSE from an expression node to a program-counter node indicates that control flow depends on the boolean value represented by the expression node.

3.2 Security Guarantees from PDGs

As Section 2 demonstrated, paths in a PDG can correspond to information flows in a program, and PIDGIN allows developers to discover, specify, and enforce security guarantees.

Information security guarantees are application specific, since what is regarded as sensitive information and what is regarded as correct handling of that information varies greatly between applications. The query language PIDGINQL (described in Section 4) provides several convenient ways for developers to indicate sources and sinks, such as queries that select the values returned from a particular function. The ability for PIDGINQL to specify relevant parts of the graph means that the program does not require annotations for security policies. PIDGIN can be used to describe many complex policies. We next describe several kinds of security guarantees that developers can express using PIDGINQL.

Noninterference The absence of a path in a PDG from a source to a sink indicates that noninterference holds between the source and the sink. This result was proved formally by Wasserrab et al. [59]. As seen in Section 2, this is equivalent to the PIDGINQL query pgm.between(source, sink) evaluating to an empty graph.

Noninterference is a strong guarantee, and many applications that handle sensitive information will not satisfy it: the query pgm.between(source,sink) will result in a non-empty graph. For example, an authentication module doesn't satisfy noninterference because it needs to reveal some information about passwords (specifically, whether a user's guess matches the password).

Even when noninterference does not hold, developers need assurance that the program handles sensitive information correctly. For example, a developer may want the result of the authentication module to depend on the password *only* via an equality test with the guess. In the remainder of this section, we describe security guarantees that are weaker than noninterference and can be expressed as queries on PDGs.

No explicit flows A coarse-grained notion of information-flow control considers only *explicit* information flows and ignores *implicit* information flows [14]. This is also known as *taint tracking* and corresponds to considering only data dependencies and ignoring control dependencies.

Although arbitrary information may flow due to control dependencies, it can be useful and important to show that there are no explicit information flows from sensitive sources to dangerous sinks. Indeed, the prevalence of taint-tracking mechanisms (e.g., Perl's taint mode, and numerous systems [2, 32, 56, 61]) show that it is intuitive and appealing for developers to consider just explicit flows. Moreover, tracking only explicit flows leads to fewer false positives (albeit at the cost of more false negatives) [15, 29].

Restricting attention to data dependencies is straightforward with a PDG. Specifically, if all paths from sensitive sources to sensitive sinks have at least one edge labeled CD (i.e., a control dependency from a program-counter node to an expression node), then there are no explicit flows from the source to the sink. This can be expressed by the following PIDGINQL policy function:

```
let noExplicitFlows(sources, sinks) =
  pgm.removeEdges(pgm.selectEdges(CD))
     .between(sources, sinks) is empty
```

Expression pgm.removeEdges(pgm.selectEdges(CD)) selects all edges labeled CD in the PDG and removes them from the graph. Using this graph, expression between(sources, sinks) finds the subgraph containing all paths between sources and sinks. If this results in an empty graph the policy holds, and there are no explicit flows from the sources to the sinks.

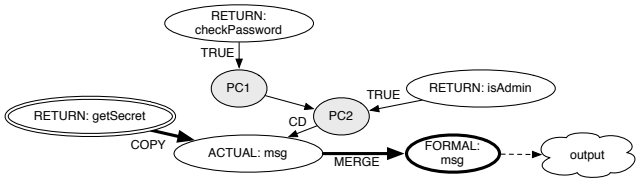
Often a program intentionally contains explicit flows (e.g., a program that prints the last four digits of a credit card number). To obtain guarantees in this case, a more precise policy is needed.

Describe all information flows In general, a developer can specify a security policy by describing all permitted paths from sensitive sources to dangerous sinks. This is because paths in the PDG correspond to information flows in the program. Using the query language, the developer can enumerate the ways in which information is permitted to flow. If, after removing paths corresponding to these permitted information flows, only an empty graph remains then all information flows in the program are permitted, and the program satisfies the security policy. The "no explicit flows" example can be viewed in this light (i.e., all paths from a source to a sink must involve a control dependency), but more expressive characterizations of paths are often necessary, useful, and interesting.

For example, consider a program which takes a (secret) credit card number and prints the last four digits. This is an intentional explicit flow, though most taint analysis frameworks would mark it

```
1  if (checkPassword(pwd))
2      if (user.isAdmin())
3          output(getSecret());
```

(a) Access control program



(b) Relevant fragment of PDG

Figure 2: Access control example

as a security violation. The following policy requires that all paths from the credit card number to the output go through the return value of method `lastFour`.

```
let ccNum = ... in
let output = pgm.formalsOf(''output'') in
let lastFourRet = pgm.returnsOf(''lastFour'') in
pgm.declassifies(lastFourRet, ccNum, output)
```

Recall that pgm.declassifies(lastFourRet, ccNum, output) (seen in Section 2) removes the nodes lastFourRet from pgm, and asserts that in the resulting graph there are no paths from ccNum to output.

This policy treats method `lastFour` as a *trusted declassifier* [24]: information is allowed to flow from ccNum to output provided it goes through the return value of `lastFour` because `lastFour` is trusted to release only limited information about credit card numbers. Determining whether `lastFour` is in fact *trustworthy* is beyond the scope of this work. Trustworthiness of `lastFour` could, for example, be achieved through a code review, or through formal verification of its correctness. Nonetheless, this PIDGINQL policy provides a strong security guarantee, and reduces the question of correct information flow in the entire program to the trustworthiness of one specific method.

Describe conditions for information-flow In some cases it is important to know not just the flows from sensitive sources to dangerous sinks, but also under what conditions these flows occur. Using PDGs, we can extract this information by considering control dependencies of nodes within a path. This is difficult for most existing information-flow analyses, as the conditions under which a flow occurs are not properties of the paths from sources to sinks.

For example, consider the program in Figure 2a, which is a simple model of an access control check guarding information flow. Secret information is output at line 3, but only if the user provided the correct password (line 1) and the user is the administrator (line 2). If we look at the relevant fragment of the PDG for this program (Figure 2b) we see that there is a single path from a sensitive source (the double-circled node for the return from the getSecret function) to a dangerous sink (the bold node representing the formal argument to output). By examining the control dependencies for one of the nodes on this path, we can determine that this flow happens only if both access control checks pass. All paths from the source to the sink are control dependent on both "`checkPassword`" and "`isAdmin`" returning true. We can describe this with:

```
1 let sec = pgm.returnsOf(''getSecret'') in
2 let out = pgm.formalsOf(''output'') in
3 let isPassRet = pgm.returnsOf(''checkPassword'') in
4 let isAdRet = pgm.returnsOf(''isAdmin'') in
5 let guards = pgm.findPCNodes(isPassRet, TRUE) ∩
6                pgm.findPCNodes(isAdRet, TRUE) in
7 pgm.removeControlDeps(guards).between(sec, out) is empty
```

Query	$Q ::= F\ Q \mid E$
Policy	$P ::= F\ P \mid E$ is empty $\mid p(A_0, \ldots, A_n)$
Function Definition	$F ::=$ let $f(x_0, \ldots, x_n) = E$; $\mid$ let $p(x_0, \ldots, x_n) = E$ is empty;
Expression	$E ::=$ pgm $\mid E.PE \mid E_1 \cup E_2 \mid E_1 \cap E_2$ $\mid$ let $x = E_1$ in $E_2 \mid x \mid f(A_0, \ldots, A_n)$
Argument	$A ::= E \mid EdgeType \mid NodeType$ $\mid JavaExpression \mid ProcedureName$
Primitive Expression	$PE ::=$ forwardSlice(E) $\mid$ backwardSlice(E) $\mid$ shortestPath(E_1, E_2) $\mid$ removeNodes(E) $\mid$ removeEdges(E) $\mid$ selectEdges($EdgeType$) $\mid$ selectNodes($NodeType$) $\mid$ forExpression($JavaExpression$) $\mid$ forProcedure($ProcedureName$) $\mid$ findPCNodes(E, $EdgeType$) $\mid$ removeControlDeps(E)
	$EdgeType ::=$ CD $\mid$ EXP $\mid$ TRUE $\mid$ FALSE $\mid \ldots$
	$NodeType ::=$ PC $\mid$ ENTRYPC $\mid$ FORMAL $\mid \ldots$

Figure 3: PIDGINQL grammar

Lines 1 and 2 find the appropriate PDG nodes for the secret and output functions, respectively. The expressions on lines 5 and 6 finds any program counter nodes in the PDG corresponding to program points that can be reached only when `checkPassword` and `isAdmin` return true. The primitive removeControlDeps(E) removes nodes from the graph that are control dependent on any program counter node in E. The graph pgm.removeControlDeps(guards) is, intuitively, the result of removing all nodes that are reachable only when the password is correct and the user is the admin. The following policy function captures this pattern:

```
let flowAccessControlled(G, checks, srcs, sinks) =
    G.removeControlDeps(checks).between(srcs, sinks)
    is empty
```

In the example above, access control checks protect information flow from a source to a sink. A simpler pattern is when access control checks guard execution of a sensitive operation. The following policy function asserts that execution of sensitiveOps (representing sensitive operations, such as calls to a dangerous procedure) occurs only when access control checks represented by checks succeed:

```
let accessControlled(G, checks, sensitiveOps) =
    G.removeControlDeps(checks) ∩ sensitiveOps
    is empty
```

4. Querying PDGs with PidginQL

We have developed PIDGINQL, a domain-specific language that allows a developer to explore information flows in a program, and to specify security policies that restrict information flows. PIDGINQL is a graph query language, specialized to express readable and intuitive queries relevant to information security. The grammar for PIDGINQL is shown in Figure 3. The grammar includes let statements, functions, graph composition operations, and primitives that are useful for expressing information security conditions.

Queries and expressions A query Q is a sequence of function definitions followed by a single expression. Expressions evaluate to graphs. There is a single constant expression, pgm (short for *program*), which always evaluates to the original program dependence graph for the program under consideration. A primitive expression

PE is a function on a graph: $E_0.PE$ evaluates expression E_0 to a graph G_0 and then the primitive expression returns a subgraph of G_0, computed according to the semantics of the specified operation (which we describe in more detail below and throughout the paper). Expression $E_1 \cup E_2$ evaluates E_1 and E_2 to graphs G_1 and G_2 respectively and returns the union of G_1 and G_2. Similarly, $E_1 \cap E_2$ evaluates both E_1 and E_2 and returns the intersection of the results. Expressions also include let bindings, variable uses, and invocations of user-defined functions.

Policies A policy P is a sequence of function definitions followed either by an assertion that expression E evaluates to an empty graph (E is empty) or an invocation of a user-defined policy function (which will assert that some expression evaluates to an empty graph). As discussed in Sections 2 and 3, if a query, Q, considers all information flows from sources to sinks, and removes only permitted flows, *and* Q results in an empty graph when evaluated on a program's PDG, the program contains only permitted information flows. Evaluating a policy results in an error if the assertion fails, i.e., if the query does not evaluate to an empty graph.

Queries are typically used when interactively exploring information flows, since non-empty query results can be examined and further explored to understand the information flows present in a program and discover security violations. Policies are useful for enforcement and regression testing (i.e., determining whether a modified program still satisfies a security guarantee).

Primitive expressions PIDGINQL contains several primitive operations for exploring information flows in programs and specifying restrictions on permitted information flows. These are described throughout the paper; due to space constraints we only briefly describe some here.

Expression forwardSlice is useful for selecting everything *influenced by* sensitive sources and backwardSlice for selecting everything that *influences* critical sinks. Both forwardSlice and backwardSlice may take another argument (not shown in the grammar) that controls the depth of the slice, for example to select immediate successors of a node.

For example, expression E_0.forwardSlice(E_1) evaluates the subexpressions to graphs G_0 and G_1 and computes the subgraph of G_0 that is reachable from any node in G_1. We improve the precision of slicing by including only nodes of G_0 that are reachable from a node in G_1 by a *feasible path* (i.e., a path where method calls and returns are appropriately matched). The call graph we use to construct the PDG is necessarily a finite approximation of the actual control flow of the program. Removing infeasible paths from slices, an example of CFL-reachability [44], greatly improves the precision of queries and policies, as it helps mitigate the imprecision that arises from this finite approximation.[4]

Expression E_0.shortestPath(E_1,E_2) is useful during exploration to find a simple (feasible) path remaining after executing a query. This helps identify vulnerabilities or missing security conditions.

Expression E_0.findPCNodes(E_1, $EdgeType$) is used to find program counter nodes in E_0 that correspond to control-flow decisions based on expressions in E_1. Edge type $EdgeType$ must be either TRUE or FALSE. If E_0 and E_1 evaluate to graphs G_0 and G_1 respectively, E_0.findPCNodes(E_1, TRUE) evaluates to the program counter nodes in G_0 that are reachable only by a TRUE edge from some expression node in G_1. That is, the program point corresponding to a program counter node in E_0.findPCNodes(E_1, TRUE) will be reached only if some expression in G_1 evaluates to true.

[4] Faster, but less precise primitive expressions (not shown in the grammar) are also provided that compute slices that may include infeasible paths.

Expression E_0.removeControlDeps(E_1) can be used in combination with findPCNodes, for removing nodes that are control dependent on a boolean expression. In Section 3, we use removeControlDeps to define access control policies.

Any primitive expression that takes a $ProcedureName$ or $JavaExpression$ as an argument will raise an error if it evaluates to an empty graph. This ensures that API changes, such as changing a method name, will trigger an error until a corresponding change is made to the PIDGINQL policy.

User-defined functions PIDGINQL functions are defined with let $f(x_0, \ldots, x_n) = E$ and let $p(x_0, \ldots, x_n) = E$ is empty. Function definitions are either graph functions (which will evaluate to a graph) or policy functions (which assert that some expression evaluates to an empty graph).[5] Functions are invoked with syntax $f(A_0, \ldots, A_n)$. We also support $A_0.f(A_1, \ldots, A_n)$ as alternative syntax to allow user-defined functions to be easily composed with other operations.

Examples of user-defined functions in Sections 2 and 3 are between, formalsOf, and returnsOf. For example, function entriesOf, which finds entry program-counter nodes in G for procedures matching $ProcedureName$, is defined as:

let entries(G, $ProcedureName$) =
  G.forProcedure($ProcedureName$).selectNodes(ENTRYPC)

User-defined functions are a powerful tool for building complex queries and policies. We have identified useful (non-primitive) operations and defined them as functions. In our query evaluation tool, these definitions are included by default, providing a rich library of useful functions, including between, formalsOf, returnsOf, entriesOf, declassifies, noExplicitFlows, and flowAccessControlled.

5. Implementation

PIDGIN has two distinct components. The first component analyzes Java bytecode, including JDK (up to 1.6) and library code, and produces PDGs. The second component evaluates queries against a PDG, and can be used either interactively or in "batch mode." Interactive mode displays results of queries in a variety of formats and is useful to explore information flows in a program, for example, to explore security guarantees in legacy programs or to find information flows that violate a given policy. The ability to interactively query a program to discover and describe information flows is a novel contribution of this work. Batch mode simply evaluates PIDGINQL queries and policies and is useful for checking that a program enforces a previously specified policy (e.g., as part of a nightly build process).

PDG construction Our implementation, which uses the WALA framework,[6] is approximately 22,700 lines of code. Of this, 7,500 lines implement a custom multi-threaded pointer analysis engine. A scalable pointer analysis is key to the scalability of PIDGIN; the multi-threaded engine significantly outperforms WALA's pointer analysis. The remaining code implements the PDG construction, including various dataflow analyses to improve the precision of the PDG. For example, we determine the precise types of exceptions that can be thrown, improving control-flow analysis, and therefore enabling more precise enforcement of security policies.

We construct a PDG for all code reachable from a specified `main` method via an interprocedural dataflow analysis. We use an object-sensitive pointer analysis (a 2-type-sensitive analysis with a 1-type-sensitive heap [50]). We use additional precision for Java

[5] For presentation purposes, we syntactically distinguish graph and policy functions; in the implementation using a policy function where a graph function is expected will result in an evaluation error not a parsing error.

[6] http://wala.sf.net/

Program	Size (LoC)	Pointer Analysis				PDG Construction			
		Time (s)		Nodes	Edges	Time (s)		Nodes	Edges
		Mean	SD			Mean	SD		
CMS	161,597	2.0	0.2	333,741	2,557,316	13.1	0.1	1,812,263	3,540,271
FreeCS	102,842	0.8	0.1	135,489	545,853	6.0	0.1	742,860	1,407,576
UPM	333,896	5.7	1.1	637,348	17,255,214	24.8	0.1	3,544,271	7,016,244
Tomcat	160,432	6.6	0.2	508,227	11,474,544	14.3	0.1	1,973,632	4,048,266
PTax	65,165	1.1	0.2	92,527	2,088,865	2.8	0.2	280,633	539,253

Figure 4: Program sizes and analysis results

standard library container classes (3-type-sensitive with a 2-type-sensitive heap) and string builders (1-full-object-sensitive [50]) to reduce false dependencies in these commonly used classes. The PDG construction analysis is context sensitive, object sensitive, and field sensitive. It is flow insensitive for heap locations, but achieves a form of flow sensitivity for local variables due to WALA's static single assignment representation [13].

We handle all Java language features except reflection. The PDG captures all control and data dependencies, but not dependencies due to concurrent races. Because our analysis is flow-insensitive for heap locations, all reads of a given heap location depend on all writes to that location, which soundly approximates concurrent access to shared data.

Like other practical Java pointer analyses (e.g., WALA's pointer analysis and Doop [8]), in order to scale we use a single abstract object to represent all `java.lang.String`s. For increased precision in the PDG, we (soundly) treat methods on `String` objects and objects of immutable primitive wrapper classes (`java.lang.Integer`, etc.) as primitive operations by replacing method calls with edges describing their effects. This is key to PIDGIN's scalability and precision. Different `String`s contain different information, and must be distinguished to enforce realistic security policies. Treating `String`s like primitive values in the PDG provides sufficient precision while permitting a scalable pointer analysis. In addition, we provide analysis result signatures for some native methods. For native methods without signatures, we assume that the return values of the methods depend only on the arguments and the receiver, and that the methods have no heap side-effects. These assumptions are potential sources of unsoundness in our analysis.

PidginQL Query Engine We implemented a custom query engine for PIDGINQL that evaluates queries against PDGs. Although PIDGINQL could be implemented using an existing graph query language and engine (such as Cypher[7] or Gremlin [25]), we used a custom engine for flexibility and fast prototyping.

The query evaluator is 8,700 lines of Java code. It implements call-by-need semantics and caches subquery results. Call-by-need reduces the graph expressions that must be evaluated. Caching improves performance, particularly when used interactively, since subqueries are often reused. When exploring information flows with PIDGIN, a user typically submits a sequence of similar queries.

6. Case Studies

In this section we present the results of applying PIDGIN. For three legacy applications there was no predefined specification and we used PIDGIN to explore the information flows and discover precise security policies that these applications satisfy. These were a web-based Course Management System (CMS); and two open-source applications, Free Chat-Server (FreeCS) and Universal Password Manager (UPM). For a fourth legacy application, the Apache Tomcat web server, we developed policies based on reported vulnerabilities and confirmed that the policies hold after patching, but fail on the unpatched version. We used our system to support simultaneous application and policy development for a small tax application we wrote ourselves, PTax. The diversity and specificity of these policies demonstrate the flexibility and expressivity of PIDGINQL.

[7] http://neo4j.com/developer/cypher-query-language/

Program	Policy	Time (s)		Policy
		Mean	SD	LoC
CMS	B1	5.5	0.06	3
	B2	3.9	0.06	5
FreeCS	C1	1.3	0.03	10
	C2	4.2	0.13	31
UPM	D1	11.7	0.12	7
	D2	13.3	0.13	12
Tomcat	E1	< 0.1	< 0.01	4
	E2	5.9	0.12	10
	E3	0.1	< 0.01	3
	E4	5.9	0.03	4
PTax	F1	0.4	< 0.01	4
	F2	1.3	0.01	14

Figure 5: Policy evaluation times

In addition we apply PIDGIN to the SecuriBench Micro benchmark [36], and in Appendices A and B we discuss using PIDGIN both to explore security guarantees of a legacy application and to specify and enforce policies during development of an application.

6.1 Analysis Performance

The first column of Figure 4 presents the lines of code analyzed, i.e., lines reachable from the specified `main` method, including JDK 1.6 and library code. For each Tomcat vulnerability, we wrote a test harness that exercises the component(s) containing the vulnerability, and ran PIDGIN on the harness. PIDGIN thus analyzes all code reachable from the test harness, which does not include all Tomcat components. Figure 4 shows results for only the largest harness.

Figure 4 also presents the performance of the pointer and PDG construction analyses, giving the mean and standard deviation (SD) of ten runs. Analyses were performed on a 16 vCPU Amazon EC2 instance using Intel Xeon E5-2666 processors with 30GB of RAM.

Figure 5 summarizes policy evaluation times for all policies discussed in this section, based on ten evaluations. Policy times are reported for a cold cache (i.e., with no previously cached results for subqueries). Each policy evaluates in under 14 seconds. The last column in Figure 5 gives the number of lines for each policy.

6.2 Course Management System (CMS)

CMS [7] is a J2EE web application for course management that has been used at Cornell University since 2005. We used a version of CMS that replaces the relational database backend with an in-memory object database. This version has previously been used to test performance in a distributed computing system [33]. CMS uses

the model/view/controller design pattern. We examined the security of the model and controller logic; views simply display the results.

Policy B1. *Only CMS administrators can send a message to all CMS users.*

This is a typical access control policy, ensuring that the function used to send messages to all users, is called only when the current user is an administrator.

```
let addNotice = pgm.entriesOf("addNotice") in
let isAdmin = pgm.returnsOf("isCMSAdmin") in
let isAdminTrue = pgm.findPCNodes(isAdmin,TRUE) in
pgm.accessControlled(isAdminTrue, addNotice)
```

Policy B2. *Only users with correct privileges can add students to a course.*

This five line policy is similar to Policy B1.

6.3 Free Chat-Server

Free Chat-Server is an open-source Java chat server that has been downloaded nearly 100,000 times.[8] Once the chat server has started, users can send messages, maintain friend lists, create, join and manage group chat sessions, etc. Administrators can ban, kick, and punish misbehaving users.

Policy C1. *Only superusers can send broadcast messages.*

We used PIDGIN to confirm that the ability to send messages to all users is available only to users with the right `ROLE_GOD`. This can be described with an access control policy similar to others previously presented. However, while exploring the information flows present in this program, we realized that our initial definition of what constituted a "broadcast message" was imprecise. PIDGIN enabled us to quickly find this apparent violation of the policy and refine our security policy appropriately.

Policy C2. *Punished users may perform limited actions.*

Misbehaving users can be disciplined by setting a `punished` flag in the object representing the user. In the PDG for Free Chat-Server, there are 357 sites where actions can be performed, all of which are invocations of the same method. We developed a PIDGINQL policy that precisely describes which actions a punished user may perform by using PIDGIN to interactively explore information flows, focusing on calls to the "perform action" method that were not access controlled by the `punished` flag. The final policy is 31 lines of PIDGINQL, the largest we have developed.

6.4 Universal Password Manager (UPM)

UPM is an open-source password manager. Users store encrypted account and password information in the application's database and decrypt them by entering a single master password. It has been downloaded over 90,000 times.[9]

Policy D1. *The user's master password entry does not explicitly flow to the GUI, console, or network except through trusted cryptographic operations.*

When we consider only the data dependencies in the program, the user's master password entry flows to public outputs only via the encryption and decryption operations in the trusted Bouncy Castle cryptography library.

Policy D2. *The user's master password entry does not influence the GUI, console, or network inappropriately.*

When we consider control dependencies, we find that the user's master password entry may influence public outputs, but only in appropriate ways (through trusted declassifiers). For example, an incorrect or invalid password triggers an error dialog box, and our policy accounts for this flow.

[8] http://sourceforge.net/projects/freecs/

[9] http://upm.sourceforge.net/

6.5 Apache Tomcat

Apache Tomcat[10] is a popular open source web server. Tomcat provides application developers with Java Servlet and Java Server Pages APIs, an HTTP server, and tools and management interfaces for server administrators. For several reported Tomcat vulnerabilities from the CVE database,[11] we developed PIDGINQL policies and confirmed that the policies fail to hold on vulnerable versions of Tomcat, and successfully hold on patched versions.

Note that the use of PIDGIN on a test harness provides stronger guarantees than a simple test case. Most importantly, PIDGIN can test information-flow properties (e.g., noninterference) which are not testable by a single test case. In addition, a single PIDGINQL policy on a test harness provides guarantees on many possible executions. For the Tomcat test harnesses, we effectively test all possible parameters of server requests, because PIDGINQL policies and the PDG construction do not examine specific string values.

Policy E1. *CVE-2010-1157: The BASIC and DIGEST authentication HTTP headers do not leak the local host name or IP address of the machine running Tomcat.*

The PIDGINQL policy asserts that there are no paths from the sources of the host name and IP address to the authentication headers. This is a standard noninterference policy and ensures the completeness of the fix.

Policy E2. *CVE-2011-0013: Data from web applications must be properly sanitized before being displayed in the HTML Manager.*

It should not be possible for client web applications to run arbitrary scripts in the HTML Manager, a component for use by Tomcat administrators. This vulnerability arose because some data from client web applications was not properly sanitized. The PIDGINQL policy identifies the sanitization functions and asserts that all data from client applications that is displayed by the HTML manager passes through a sanitization function. Note that the policy does not ensure the proper *implementation* of the sanitization functions, but identifies them as trusted code that needs to be inspected.

Policy E3. *CVE-2011-2204: A user's password should not flow into an exception which gets written to the log file.*

The PIDGINQL policy is a noninterference policy asserting that the password does not influence the arguments to any exception method. This includes the creation of exceptions that leaked the password prior to the fix for CVE-2011-2204, but also ensures that there were no similar leaks elsewhere in the code.

Policy E4. *CVE-2014-0033: Session IDs provided in the URL should be ignored when URL rewriting is disabled.*

The session ID from the request should not be used if URL rewriting is explicitly disabled. The PIDGINQL policy is a flow access controlled policy asserting that, if URL rewriting is disabled, then the session ID in the URL does not influence the session to which a request is associated.

6.6 PTax

PTax is a toy tax computation application. PTax supports multiple users who login with a username and password and input their tax information. This sensitive information is stored in a file to be accessed by the user at a later time, provided the user supplies the correct password. Before development, we defined a number of PIDGINQL policies we expected to hold. As development progressed, the policies were iteratively refined to reflect implementation choices (e.g., names of methods, signature of the authentication module), although the intent of the policies remained the same.

Policy F1. *Public outputs do not depend on a user's password, unless it has been cryptographically hashed.*

[10] http://tomcat.apache.org/

[11] http://cve.mitre.org/

```
let passwords = pgm.returnsOf(''getPassword'') in
let outputs = pgm.formalsOf(''writeToStorage'') ∪
              pgm.formalsOf(''print'') in
let hashFormals = pgm.formalsOf(''computeHash'') in
pgm.declassifies(hashFormals, passwords, outputs)
```

This is a trusted-declassification policy. The declassifies function ensures that the only information flow from the user's password to public outputs are through the argument to the hash function.

Policy F2. *Tax information is encrypted before being written to disk and decrypted only when the password is entered correctly.*

Policy F2 is a combined declassification policy and access control policy, whose exact statement depends on the specification of the `userLogin` method.

6.7 Micro-benchmark Results

Test Group	Detected	False Positives
Aliasing	12/12	0
Arrays	9/9	5
Basic	63/63	0
Collections	14/14	5
Data Structures	5/5	0
Factories	3/3	0
Inter	16/16	0
Pred	5/5	2
Reflection	1/4	0
Sanitizers	3/4	0
Session	3/3	1
Strong Update	1/1	2
Total	159/163	15

Figure 6: SecuriBench Micro results

To compare with other Java analysis tools, we ran PIDGIN on the SecuriBench Micro [36] 1.08 suite of 123 small test cases. We develop PIDGIN policies for each test and detect 159 out of a total of 163 vulnerabilities. We do not detect vulnerabilities due to reflection. We also miss an incorrectly written sanitization function, though our policy marks it as a trusted declassifier, and thus indicates it should be inspected or otherwise verified.

For many tests the policy is a simply noninterference, requiring that sensitive values from an HTTP request do not affect public output. For some tests there is an allowed implicit flow, and we developed appropriate policies. Some tests require domain-specific policies (e.g., the Sanitizers tests required application-specific declassification policies).

False positives were caused by known limitations of our tool, including imprecise reasoning about individual array elements, dead code elimination that required arithmetic reasoning (Pred), and flow-insensitive tracking of heap locations (Strong Update).

7. Related Work

PDGs for security In a series of papers, Snelting and Hammer (and collaborators) argue for the use of PDGs for information-flow control, due to the precision and scalability of PDGs. They have developed JOANA [21], an object sensitive and context sensitive tool for checking noninterference in Java bytecode [22], shown their techniques to be sound [59], and considered information flow in concurrent programs [17]. They also use path conditions to improve precision by ruling out impossible paths [54]. Hammer et al. [23] consider enforcement of a form of *where* declassification [48].

The key differences between our work and previous work using PDGs for information-flow control is that (1) our query language allows for expressive, precise, application-specific policies that are separate from code, whereas JOANA requires program annotations and supports a limited class of policies; (2) we seek to use the PDG to enable *exploration* of security guarantees of programs in addition to *enforcement* of explicitly specified security guarantees; and (3) PIDGIN scales to larger programs. The largest reported use of JOANA is on a program with about 63,000 lines of code (excluding the JDK 1.4 library, which is approximately 100k lines of code total) for a scalability test where no security policy is specified. For this example JOANA is only able to generate a context-insensitive PDG and this takes about a day [20, 52].

Program dependence graphs were introduced by Ferrante et al. [16], along with an algorithm to produce them. PDGs were presented as an ideal data structure for certain intra-procedural optimizations. Program slicing for an interprocedural extension to PDGs is introduced by Horwitz et al. [26] and made more precise by Reps [44] using *CFL reachability*. Program slicing is useful for describing security guarantees and is built into PIDGINQL as primitive expressions forwardSlice and backwardSlice. Reps and Rosay [45] define *program chopping*, of which the PIDGINQL function between, defined in Section 2, is an example. Abadi et al. [1] develop a core calculus of dependency. Although they do not directly consider program dependence graphs, they show that program slicing and information flow type systems can be translated to this calculus. Cartwright and Felleisen [9] give a denotational semantics to PDGs derived from the semantics of the original program. Bergeretti and Carré [6] use structures similar to PDGs to automatically find bugs in *while* programs and increase program understanding.

Yamaguchi et al. [60] use intraprocedural PDGs together with abstract syntax trees to detect vulnerabilities in C code. Vulnerabilities (e.g., buffer overflows) are identified using *graph traversals*, which are similar to some of our graph queries. Unlike PIDGIN, the vulnerabilities their tool found were each contained within a single function, and their tool does not support whole program security policies. Furthermore, they consider only properties of a single program execution rather than application-specific information-flow properties such as those described in Sections 3 and 6. As common with bug-finding tools, their tool does not attempt to guarantee the absence of vulnerabilities even if none are found.

Kashyap and Hardekopf [28] use PDGs to infer *security signatures* describing how information flows within small (under 5k lines) JavaScript browser add-ons. These signatures can then by used by an auditor to decide whether an add-on should be accepted. PIDGIN is similarly focused on increasing program understanding. Unlike our work, where policies can be application specific, they use a predefined set of sources and sinks. In addition to distinguishing control and data dependencies, their PDG edges contain annotations to indicate which edges may be more likely to carry relevant information. These additional annotations could also benefit PIDGIN, for example to help prioritize potential policy violations to present to the user.

Legacy applications and policy inference PIDGIN supports discovering information security guarantees for legacy applications. Rocha et al. [46] present a framework that allows declassification policies to be specified for legacy applications. Policies are separate from code and enforcement of policies is checked using *expression graphs*, which, like PDGs, capture data and control dependencies. Policies are specified as graphs that describe which expression graphs can be declassified. Unlike the framework of Rocha et al., PIDGIN supports a rich class of policies and allows developers to *explore* the information flows in an application, and thus provides support for deciding what policy is appropriate for an application. By contrast, Rocha et al. only discuss declassification and do not consider how developers produce policies. Moreover, we have implemented our approach for Java bytecode; to the best of our knowledge, Rocha et al. do not implement their framework, nor consider how to extend to a full-fledged programming language.

Other work seeks to infer security policies for existing programs. Vaughan and Chong [57] use a data-flow analysis to infer expressive information security policies that describe what sensitive information may be revealed by a program. King et al. [29], Pottier and Conchon [43], Smith and Thober [51], and the Jif compiler [40, 41] all perform various forms of type inference for security-typed languages. Mastroeni and Banerjee [39] use refinement to derive a program's semantic declassification policy. We do not currently support automatic inference of security policies from a PDG. We instead provide the developer with tools and abstractions to help them explore the information flows in a program.

Several analyses infer explicit information flows (e.g., [34, 35, 37]). While efficient and practical, these analyses do not track implicit flows and may be inadequate in settings where strong information security is required. As described in Section 3, PIDGIN also supports exploration of explicit information flows, and policies for explicit information flows.

Enforcement of expressive policies Many tools and techniques seek to *enforce* expressive and strong information security policies. Security-type systems (e.g., [41, 49, 58]) are the main technique used to enforce such policies. The survey by Sabelfeld and Myers [47] provides an overview of these security policies and enforcement techniques. More recently, Banerjee et al. [5] combine security-types with an expressive logic for describing a program's declassification policy, and Nanevski et al. [42] use an expressive type-theoretic verification framework to specify and enforce rich information-flow properties. The security guarantees we consider in Section 3.2 are related to the security policies considered in these previous works. The absence of paths from sources to sinks corresponds to noninterference. Requiring all paths to go through certain nodes (such as the formal argument of a sanitization function) is a form of trusted declassification (e.g. [24, 38]). Reasoning about the conditions under which potentially dangerous information flows occur is similar to reasoning about *when* declassification is permitted [12, 48]. Restricting attention to only explicit information flows is equivalent to a static taint analysis (e.g., [2, 19, 34, 35, 37, 56]).

8. Conclusion

Program dependence graphs precisely capture the information-flows within programs. We present the novel insight that because individual paths within a PDG correspond to particular information-flows within a program, queries on PDGs offer a unified approach for the exploration, specification, and enforcement of security guarantees.

Using this insight, we have designed and implemented PIDGIN. PIDGIN combines program dependence graphs (PDGs) with an expressive query language. By using the query language to describe paths in the PDG, developers can understand how information flows within a program and express precise, application-specific security guarantees including noninterference, trusted declassification, and access-controlled information flows.

PIDGIN is a practical tool. We have used PIDGIN to explore the information security of legacy applications, to specify and enforce information security during development, and to extract policies from known vulnerabilities. PIDGIN scales to Java applications with over 300k lines. Our case studies demonstrate that PIDGIN can express (and verify enforcement of) interesting application-specific security policies, some of which are difficult or impossible to express using existing tools and techniques.

Acknowledgments

We thank Eddie Kohler, Andrew Myers, the Programming Languages Group at Harvard, and the reviewers for their helpful comments. This work is supported by the National Science Foundation under Grant No. 1054172 and Grant No. 1421770 and by the Air Force Research Laboratory. The Lincoln Laboratory portion of this work was sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[2] 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 *Proc. ACM Conf. on Program Language Design and Implementation*, 2014.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009.

[4] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. 2008 IEEE Symposium on Security and Privacy*, 2008.

[6] J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. on Programming Languages and Systems*, 1985.

[7] C. Botev, H. Chao, T. Chao, Y. Cheng, R. Doyle, S. Grankin, J. Guarino, S. Guha, P.-C. Lee, D. Perry, C. Re, I. Rifkin, T. Yuan, D. Abdullah, K. Carpenter, D. Gries, D. Kozen, A. Myers, D. Schwartz, and J. Shanmugasundaram. Supporting workflow in a course management system. In *Proc. 36th SIGCSE technical symposium on Computer science education*, 2005.

[8] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. 24th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, New York, NY, USA, 2009. ACM.

[9] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[10] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proc. 23rd Annual Computer Security Applications Conference*, 2007.

[11] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proc. 2009 ACM workshop on Secure web services*, 2009.

[12] S. Chong and A. C. Myers. Security policies for downgrading. In *Proc. 11th ACM conference on Computer and communications security*, 2004.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 1991.

[14] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Usenix Conference on Operating Systems Design and Implementation*, 2010.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 1987.

[17] D. Giffhorn and G. Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 2014.

[18] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, 1982.

[19] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information flow analysis of android applications in droidsafe. In *Proc. 2015 Network and Distributed System Security Symposium*, 2015.

[20] J. Graf. Speeding up context-, object- and field-sensitive SDG generation. In *Proc. of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[21] C. Hammer. *Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), Fak. f. Informatik, 2009.

[22] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 2009.

[23] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, 2006.

[24] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006.

[25] F. Holzschuher and R. Peinl. Performance of graph query languages: Comparison of Cypher, Gremlin and native access in Neo4j. In *Proc. Joint EDBT/ICDT 2013 Workshops*, 2013.

[26] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 1988.

[27] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *Proc. 2013 IEEE Symposium on Security and Privacy*, 2013.

[28] V. Kashyap and B. Hardekopf. Security signature inference for javascript-based browser addons. In *Proc. 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[29] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. International Conference on Information Systems Security*, 2008.

[30] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. 2004 International Symposium on Code Generation and Optimization*, 2004.

[31] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. *Proc. 11th Annual Asian Computing Science Conference*, 2006.

[32] D. Li. Dynamic tainting for deployed Java programs. In *Proc. ACM international conference companion on Object oriented programming systems languages and applications*, 2010.

[33] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proc. ACM SIGOPS Symposium on Operating systems principles*, 2009.

[34] Y. Liu and A. Milanova. Static analysis for inference of explicit information flow. In *Proc. 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2008.

[35] Y. Liu and A. Milanova. Practical static analysis for inference of security-related program properties. In *Proc. IEEE 17th International Conference on Program Comprehension*, 2009.

[36] B. Livshits. Securibench Micro, 2006. `http://suif.stanford.edu/~livshits/work/securibench-micro/`.

[37] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proc. ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[38] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proc. 2nd ASIAN Symposium on Programming Languages and Systems*, 2004.

[39] I. Mastroeni and A. Banerjee. Modelling declassification policies using abstract domain completeness. *Mathematical Structures in Computer Science*, 2011.

[40] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, 1999.

[41] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, N. Nystrom, D. Zhang, O. Arden, J. Liu, and K. Vikram. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2014.

[42] A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM Trans. on Programming Languages and Systems*, 2013.

[43] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming*, 2000.

[44] T. Reps. Program analysis via graph reachability. In *Proc. 1997 International Symposium on Logic Programming*, 1997.

[45] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proc. 3rd ACM SIGSOFT symposium on Foundations of software engineering*, 1995.

[46] B. Rocha, S. Bandhakavi, J. den Hartog, W. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *Proc. 2010 IEEE Symposium on Security and Privacy*, 2010.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[48] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, 2005.

[49] V. Simonet. The Flow Caml System: documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003.

[50] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proc. 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2011.

[51] S. F. Smith and M. Thober. Improving usability of information flow security in Java. In *Proc. 2007 Workshop on Programming Languages and Analysis for Security*, 2007.

[52] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab. Checking probabilistic noninterference using JOANA. *Information Technology*, 2015.

[53] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. 4th ACM Symposium on Haskell*, 2011.

[54] M. Taghdiri, G. Snelting, and C. Sinz. Information flow analysis via path condition refinement. In *International Workshop on Formal Aspects of Security and Trust*, 2010.

[55] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *Proc. ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[56] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. ANDROMEDA: accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*, 2013.

[57] J. A. Vaughan and S. Chong. Inference of expressive declassification policies. In *Proc. 2011 IEEE Symposium on Security and Privacy*, 2011.

[58] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 1996.

[59] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009.

[60] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. 2014 IEEE Symposium on Security and Privacy*, 2014.

[61] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM Operating Systems Review*, 2011.

```
let sources = pgm.returnsOf(''askUserForPassword'') in
let sinks = pgm.formalsOf(''javax.swing∗'')
            ∪ pgm.formalsOf(''sun.swing'')
            ∪ pgm.formalsOf(''PrintStream.print∗'')
            ∪ pgm.formalsOf(''HTTPTransport∗'') in
let declassifiers =
  // Trusted Bouncy Castle functions
  pgm.forProcedure(''CBCBlockCipher.decryptBlock'')
  ∪ pgm.forProcedure(''AESEngine.encryptBlock'')
  ∪ pgm.forProcedure(''AESEngine.decryptBlock'')
  ∪ pgm.forProcedure(''AESEngine.packBlock'') in
pgm.explicit.declassifies(declassifiers, sources, sinks)
```

Figure 7: PIDGINQL policy expressing Policy D1

A. Using PIDGIN for legacy code

The interactivity of PIDGIN was essential to understanding the security guarantees provided by legacy case-study programs and writing queries that describe these guarantees. We illustrate the interactive query and policy generation process by describing how we developed Policy D1 for Universal Password Manager (UPM).

Find sources and sinks UPM protects a user's passwords by encrypting them with a single master password. We investigated confidentiality guarantees regarding the master password that is entered by the user and used to decrypt the database containing the user's password. Inspecting the application code, we found that the master password is returned from the `askUserForPassword` method. These return values are *sources*.

By regarding the return values of `askUserForPassword` as sensitive sources, we are trusting the implementation to correctly handle data from the input: a Java Swing widget. The method is 11 lines of code and uses standard Java Swing API calls. We also trust the Swing library. This is a common way to use PIDGIN: to reduce trust in an entire application to trust in well-designed and well-maintained libraries and a small amount of application code.

We identified three different places that data may leave the application: 1) the GUI (via the Swing API); 2) the console; and 3) the network (via a custom `java.net.HTTPTransport` class). Formal arguments to methods in these three locations are *sinks*.

Try simple queries Unsurprisingly, most of the interesting work is done by the application after asking for a password and before presenting or sending results and there are paths between the sources and sinks. We first narrowed our focus to data dependencies, and tried a simple policy, pgm.noExplicitFlows(sources, sinks).

Investigate counterexamples This policy failed, revealing that there are some paths via only data dependencies. We wanted a strong policy that describes *how* data dependencies allow information about the master password to leak from the application. To begin this process, we found a counterexample using the shortestPath query, pgm.explicit.shortestPath(sources, sinks), where the expression pgm.explicit returns the data dependencies in pgm.

The resulting path converts the password to bytes and uses those bytes in a decryption function in the Bouncy Castle cryptography library to decrypt the password database. Bouncy Castle (`https://www.bouncycastle.org/`) is one of the most widely used open source Java cryptography libraries and is clearly trusted by the UPM code. Therefore we can trust the password to not leak (except via cryptographic computations) once it enters the Bouncy Castle decryption and encryption functions.

Create a* PIDGINQL *policy The final policy is shown in Figure 7. Whereas the informal description of Policy D1 is vague, the PIDGINQL policy is strong, precise, checkable, and clarifies when flows from the master password to public output are appropriate.

B. Using PIDGIN for new development

Often new development begins with an incomplete and imprecise security specification that evolves as development progresses. PIDGIN policies are flexible and, because they are not embedded in the program text, can be easily modified along with the informal security specification and the code itself. They can also be used for regression testing as the code changes.

We illustrate this process by describing the use of PIDGIN throughout the development of a toy conference management system, PChair. Access control policies in conference management systems can be intricate and complex. In the end there were fourteen separate PIDGIN security policies for PChair. Most of these policies restrict access to sensitive data (author names, reviews, etc.) and ensure proper permissions for sensitive operations (e.g., accepting a paper).

Define an informal policy. Before beginning development we wrote down the policies we desired informally. For example, one policy was initially: *Only authors of a paper, reviewers of a paper, and PC members can see a paper's reviews.*

Implement initial version of the program and* PIDGIN *policy. PChair uses role-based access control. We used simple functions to check whether the current user has a particular role, and then referred to these functions in our policies. Thus, our policies rely on the correctness of these functions, which were deliberately designed to be easy to understand. This implementation simplified PIDGINQL policy specification; most policies used flowAccessControlled to check whether the correct roles were held on every path where sensitive information was accessed.

Update policies when the specification is modified. As the functionality of the application evolved, the security policies also evolved. For example, we added a system administrator role. System administrators have superuser-like abilities, and we modified our informal specifications and PIDGIN policies accordingly. Because PIDGIN policies are not spread out throughout the code base (as, e.g., security-type annotations) updating the policies was straightforward, and accomplished easily.

Regression testing security policies. We automated regression testing, checking policies before accepting a commit to our source repository. Timely notification of the security policy failure allowed us to easily identify and fix several violations. An interesting failure happened for Policy 1, shown below.

Policy 1. *A paper's acceptance status can be released only to an author of the paper after the notification deadline, or to PC members without conflicts.*

The PIDGIN policy ensures that all flows from return values of `isAccepted` to the client are protected by the correct access check.

```
... // output = errors or responses sent to the client
... // define deadline, role, and conflict checks
let isAccepted = pgm.returnsOf(''isAccepted'') in
let check = (pgm.findPCNodes(isAuthorOf, TRUE)
             ∩ pgm.findPCNodes(notifyDeadlinePast, TRUE))
          ∪ (pgm.findPCNodes(isPC, TRUE)
             ∩ pgm.findPCNodes(hasConflict, FALSE)) in
pgm.flowAccessControlled(check, isAccepted, output)
```

During development, we discovered that this policy was not enforced. After the notification deadline, only accepted papers can be updated. If a user tries to update a rejected paper or update a paper before the deadline, an error message is displayed. However, which error message was displayed revealed information about whether or not the paper had been accepted. This implicit information flow leaked information about the paper's acceptance. PIDGIN provided enough information to identify and fix this subtle violation.