Exploring and Enforcing Application Security Guarantees via Program Dependence

Andrew Johnson
Lucas Waye
Scott Moore
and
Stephen Chong

TR-04-14

Computer Science Group
Harvard University
Cambridge, Massachusetts

Exploring and Enforcing Application Security Guarantees via Program Dependence Graphs

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

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

ABSTRACT

We present PIDGIN, a program analysis and understanding tool that allows developers to explore the information flows that exist in programs and specify and enforce security policies that restrict these information flows. PIDGIN uses *program-dependence graphs* (PDGs) to precisely capture the information flows within a program. PDGs can be queried using a custom query language to explore and describe information flows in programs. A developer can specify strong information security policies by asserting that specific queries return no results (i.e., asserting the absence of certain information flows in the program). To check whether a program satisfies a security policy, a developer can simply evaluate the query against a program's dependence graph.

The query language is expressive, supporting a large class of precise, application-specific security guarantees. PIDGIN can be used to explore information security guarantees in legacy programs, or to support the specification, enforcement, and modification of information security requirements during program development.

We describe the design and implementation of PIDGIN and report on using PIDGIN both to explore security guarantees in existing open-source applications, and to specify and enforce security guarantees during application development.

1. INTRODUCTION

Many computer applications store and compute with sensitive information, including confidential and untrusted data. Thus, application developers must be concerned with how public outputs of the system may reveal confidential information and how potentially dangerous operations may be influenced by untrusted data.

However, applications are often developed without a clear specification of information security requirements. Even if application development starts with a clear information security specification, the specification typically changes during the lifecycle of the application. Current tools and techniques for building applications with strong information security (e.g., [18, 24, 39, 40, 48]) do not make it easy either to understand the information security of an existing program, or to modify a security policy as an application changes. Other tools, such as taint-tracking systems (e.g., [1, 51]), may be easier to use, but support limited classes of security policies.

*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 author and are not necessarily endorsed by the United States Government.

We present PIDGIN, a system that enables strong application-specific information security for unmodified programs written in existing programming languages. PIDGIN allows developers both to understand the information flows in a program and to specify and enforce restrictions on these flows. To express application-specific policies, one must understand application requirements, and how they map to code. PIDGIN helps this process, providing high-level views of an application's information flow.

PIDGIN uses *program-dependence graphs* (PDGs) [15] to precisely and intuitively capture the information flows within a program. By issuing queries in a custom query language, developers can interrogate a program's dependence graph to understand its information flows. By asserting that specific queries return no results, a developer can specify strong information security policies (i.e., the absence of certain information flows), and can easily check whether a program enforces such policies (by executing the query against a program's dependence graph).

Our approach to exploring and enforcing information security has several benefits that support both the retroactive exploration of information security guarantees offered by existing applications, and the specification, enforcement, and modification of information security requirements during application development.

- PIDGIN *security policies are expressive, precise, and application specific*, since they are queries in an expressive query language designed specifically for finding and describing information flows in a program. Queries can succinctly express *global* security guarantees such as noninterference, absence of explicit information flows, trusted declassification [21], and mediation of information-flow by access control checks.
- Developers can *quickly explore an application's information security guarantees.* If there is no predefined specification then PIDGIN can be used to explore the security-relevant information flows in a program 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 the policy is not satisfied by finding information flows in the program that violate the policy.
- PIDGIN *security policies are not embedded in the code.* PIDGIN policies are specified separate from the code, but may refer to entities in the code (such as classes, methods, or even specific expressions). The code does not mention or depend on PIDGIN policies. This enables the use of PIDGIN to specify security guarantees for legacy applications without requiring modification to the application.
- *Enforcement of security policies does not prevent devel-*

opment or testing. Because the program code does not mention or depend on PIDGIN policies, the policies do not prevent compilation or execution. This makes it possible for developers to choose a balance between development of new functionality and maintenance of security policies. PIDGIN can, however, be incorporated into a nightly build process to warn developers if recent code changes violate a security policy that previously held.

These benefits stand in contrast to common techniques for enforcing strong information security in programs such as security-type systems [53] (e.g., Jif [39] and FlowCaml [46]). In security-typed languages, security policies are broken into many pieces and expressed via annotations throughout the program. In the presence of declassification [45], it can be extremely difficult to determine from these annotations how sensitive information is handled by the system as a whole. Changing the security policy may require modifying many program annotations. In addition, supporting legacy applications using these techniques is often infeasible, as they require significant annotations or modifications to the applications. In contrast, PIDGIN policies are expressed as queries that are separate from the code, in a single location, and easily modifiable.

Dynamic or hybrid information-flow enforcement mechanisms (e.g., [2, 3, 8, 24, 29, 48]) 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. PIDGIN is purely static and does not have any affect on deployed systems.

Static taint analysis tools (e.g., [9, 14, 30, 50, 51, 55]). are inevitably unsound because they do not account for information flow through control channels, and do not support expressive application-specific policies. The most recent, FlowDroid [1], 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. In the SecuriBench Micro [34] suite of tests PIDGIN found 159 of a possible 163 vulnerabilities while FlowDroid, unable to support certain classes of vulnerabilities, finds 117.

Previous work (e.g., [18, 20]) has used PDGs to *enforce* certain information security guarantees. The primary contributions of this work are:

1. The ability to easily express and enforce precise, application-specific security policies for unmodified programs;
2. The design of an expressive policy language based on graph queries used to specify and check these policies;
3. The novel insight that PDG representations also enable *exploration* and understanding of security guarantees in legacy applications;
4. The realization and demonstration of these techniques in an effective tool.

Our implementation of PIDGIN supports the Java programming language, although the techniques are applicable to other languages. We have used PIDGIN both to discover a diverse set of information security guarantees in several legacy Java applications, and to specify and enforce information security policies as part of the development process for a simple tax application and a conference management system. The legacy applications range in size up to 22,000 lines of application code and over 450,000 lines including library code. The security guarantees include: in a password manager, the confidential database is protected by a master password and the master password is not improperly leaked; in a chat server application, only highly privileged users are allowed to send broadcast messages and punished users are restricted to certain kinds of messages; and in a course management system, the high-integrity class list is correctly protected by access control checks.

The rest of this paper is structured as follows. In Section 2 we present an overview of our approach by exploring the information security guarantees of a simple guessing game program and how these guarantees are expressed as queries in PIDGINQL, our custom graph query language. Section 3 describes the structure of the PDGs we generate and security guarantees that can be inferred from them. Section 4 presents the query language, and Section 5 describes PIDGIN's implementation. We relate our experience using PIDGIN to discover, specify, and enforce information security guarantees in Section 6 (with additional details in the appendices). We discuss related work and conclude in Sections 7 and 8.

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 actual argument to the formal argument. Edges labeled CD indicate control dependencies and other edges indicate data dependencies. Dashed edges and clouds indicate 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 elided for simplicity.)

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

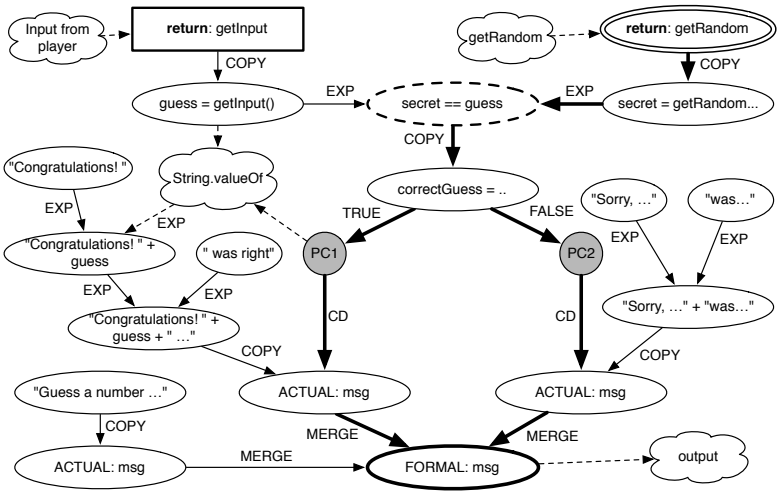
No cheating! The program should not be able to cheat by choosing a secret value that is deliberately different from the user's guess. That is, the choice of the secret should be independent of the user's input. This policy holds if the following PIDGINQL query returns an empty graph:

```
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

```
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

user's input. The second line of the query identifies the node representing the secret value (i.e., the node representing the value returned from function `getRandom`) and binds variable secret to the result. 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 in the forward slice of the node denoted by variable input, that is, the nodes and edges that are reachable by a path starting from the user input. This is all of the nodes that depend 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; it contains the nodes on which the secret value may depend. 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 the 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)
```

For a user-defined function f, we allow $A_0.f(A_1, \ldots, A_n)$ as alternative syntax for the expression $f(A_0, A_1, \ldots, A_n)$. This allows us to simplify our query to:

```
let input = pgm.returnsOf("getInput") in
let secret = pgm.returnsOf("getRandom") in
pgm.between(input, secret)
```

We can turn this 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 [17, 44] requires that information does not flow from confidential inputs to public outputs. For our purposes, the secret number (line 1 in Figure 1a) is a confidential input, and output statements (lines 3, 9, and 12) are publicly observable.

We can check whether noninterference holds using a policy that is similar to the one above:

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

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

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

Inspecting the result of the noninterference query above, we see that 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 representing 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 that this is the case 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")[1] evaluates to the node for the conditional expression (outlined with a

[1]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

dotted line in Figure 1b). 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 that we have described all paths between secret and outputs. Thus the program satisfies the following policy: *The secret does not influence the output except through the comparison with the user's guess.*

This is an example of trusted declassification [21] and is a common pattern found in many applications. We capture this with a user defined policy function asserting that all flows from sources to sinks pass through a node in declassifiers.

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

Using this function we change the last line of our policy to:

```
pgm.declassifies(check, secret, outputs)
```

Note that this 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, guarantees 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) [15] to explicitly represent the data and control dependencies within a program. PIDGIN's PDGs are mostly standard, and represent control and data dependencies within a whole program (a representation also known as a *system dependence graph* [23]). In this section, we describe the structure of PIDGIN's PDGs and describe 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 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 [12]. Each node also contains metadata, such as the position in the source code of the expression the node represents.

PIDGIN PDGs are context sensitive: nodes associated with a procedure have multiple copies, one for each analysis context in which the procedure occurs. Our analysis contexts arise from an object-sensitive pointer analysis, and thus our PDGs are object sensitive. They are also flow sensitive, in that a local variable within a procedure may have several nodes in the PDG representing its value at different program points. However, PIDGIN treats heap locations flow insensitively: a given abstract heap location[2] has only one node in the PDG to represent values stored in that location. The latter is not a fundamental restriction but a trade off between PDG size and precision.

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 value represented by 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, for example the edges from the program-counter nodes to the actual arguments in Figure 1b. 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 the 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. PIDGIN allows developers to discover, specify, and enforce information security guarantees by using a program's PDG to explore and restrict the information flows present in the program.

Security guarantees are application specific, since what is regarded as sensitive information and what is regarded as correct handling of sensitive information varies greatly from application to application. The query language PIDGINQL (described in Section 4) provides several convenient ways for developers to indicate sources and sinks, such as queries that select nodes that represent the values returned from a particular function. PIDGIN can be used to describe many complex policies, however, there are similarities in the kinds of security guarantees that developers can express using PDGs.

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. [54]. 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, a web application that receives untrusted input from a client and must construct a database query using this untrusted data does not satisfy noninterference from the untrusted input to the database. Similarly, 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 precisely specify queries and policies if needed.

[2]An *abstract heap location* is a static analysis entity that represents zero or more *concrete heap locations* (e.g., fields of objects) that may exist at run time. Abstract heap locations are typically determined by pointer analysis.

assurance that the program handles sensitive information correctly. For example, a developer may want all untrusted input to be passed to a special `escapeSQL` function before being given to the database or may want the return 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 [13]. 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 [1, 9, 30, 51, 55]) 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) [14, 26] .

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
```

Query pgm.removeEdges(pgm.selectEdges(CD)) selects all edges labeled CD in the PDG and removes them from the graph. Using this graph, query between(sources, sinks) finds the subgraph containing paths between sources and sinks. If the whole query results in an empty graph, then there are no explicit flows between the source and sinks.

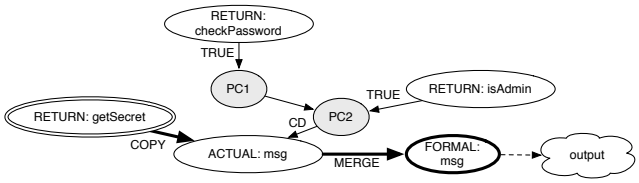
Often a program intentionally contains explicit flows (e.g. a program which prints out the last four digits of a credit card number). To obtain guarantees in this case, a more precise policy is needed.

All information flows described. 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., the policy requires that 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 (most taint analysis frameworks would mark this as a security violation). The following PidginQL policy requires that all paths from the credit card number to the output go through the return value of

```
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

method, `lastFour`.

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

Recall pgm.declassifies(lastFourRet, ccNum, output) (defined in Section 2) which removes the nodes lastFourRet from graph 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* [21]: 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.

Conditions for information-flow described. 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 path from source to sink.

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). But 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. That is, the potentially dangerous information flow happens only when the user has correctly authenticated as the administrator: all paths from the sensitive source to the dangerous sink are control dependent on both "`checkPassword`" and "`isAdmin`" returning true. We can describe this with the policy:

```
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 pgm.removeControlDeps(pgm.[isAdRet && isPassRet])
6       .between(sec,out) is empty
```

The queries on lines 1 and 2 find the PDG nodes for the return values of the secret and output functions, respectively. The query pgm.[isAdRet && isPassRet] 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. Intuitively, the graph pgm.removeControlDeps(pgm.[isAdRet && isPassRet]) is the result of removing all nodes that are reachable only when the password is correct and the user is the admin.

User-defined functions can express such access control policies. The following function asserts that information flows in graph G from sources to sinks only when access control checks represented by checks succeed:

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

The example policy is now more readable and intuitive:

```
let sec = pgm.returnsOf("getSecret") in
let out = pgm.formalsOf("output") in
let isPass = pgm.returnsOf("checkPassword") in
let isAd = pgm.returnsOf("isAdmin") in
pgm.flowAccessControlled(pgm.[isAd && isPass],sec,out)
```

In the example above, access control checks protect information flow from a confidential source to a public output. A simpler pattern is when access control checks guard whether a sensitive operation can be executed. The following policy function asserts that execution of sensitiveOps (representing some sensitive operation, such as calls to an `executeQuery` function that executes SQL queries) 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 E_0 to a graph G_0, and the primitive operation 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.

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 $\mid$ FORMAL $\mid$ EXPR $\mid \ldots$

Figure 3: PidginQL grammar

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, then 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.

Policies are not typically used when interactively exploring information flows in a program, since the non-empty result of a query can be examined and further explored to understand the information flows present in a program and/or discover security violations. Policies are, however, useful for enforcement and regression testing to determine whether a modified program still satisfies a security guarantee.

Primitive operations. 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 the immediate successors of a node.

Expression E_0.shortestPath(E_1,E_2) is useful during exploration to find a simple path remaining after executing a query, which can help 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. It is useful to combine the results of findPCNodes queries. For example, pgm.findPCNodes(a, TRUE) ∩ pgm.findPCNodes(b, FALSE) will evaluate to the program counter nodes for program points that are reached only when an expression denoted by a evaluates to true *and* an expression denoted by b evaluates to false. We add syntactic sugar to describe these combinations using boolean expressions in square brackets, simplifying the example to pgm.[a && !b].

Finally 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 restriction is to avoid trivially empty query results due to mistyped specifications of sources or sinks, and to ensure that API changes, such as changing a method name, will trigger an evaluation error until a corresponding change is made to the PIDGINQL policy.

User-defined functions. PIDGINQL allows functions to be defined with the two constructs 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).[3] 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, returnsOf, and entriesOf. For example, the function formalsOf(G,*ProcedureName*), which finds all formal arguments in G for procedures matching *ProcedureName*, is defined as:

```
let formalsOf(G, ProcedureName) =
    G.forProcedure(ProcedureName).selectNodes(FORMAL)
```

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, declassifies, noExplicitFlows, and flowAccessControlled.

5. IMPLEMENTATION

PIDGIN has two distinct components. The first component analyzes Java programs and produces PDGs. The second component evaluates queries against a PDG, and can be used either interactively or in "batch mode". The interactive mode displays results of queries in a variety of formats, and is useful for developers to explore the 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). In this section we describe the implementation of these two components.

5.1 PDGs for Java programs

We construct PDGs for Java programs using an interprocedural analysis implemented in the Accrue Interprocedural Java Analysis Framework, a framework for pointer analysis and interprocedural analysis of Java programs we have developed and publicly released. Accrue enables whole-program analysis of Java 1.6 source code by performing flow-insensitive pointer analysis and using the resulting points-to graph to guide the analysis of code. For PDG construction, we use a *uniform hybrid pointer analysis* [25], which provides benefits of both object-sensitive and call-site sensitive analyses. (Specifically, we use a uniform 2-type-sensitive analysis with 1-context-sensitive heap analysis.)[4] Accrue efficiently manages interprocedural analyses, and re-analyzes code in a given context only when necessary.

Accrue comprises about 36,000 non-comment non-blank lines of code. The interprocedural analysis pass that constructs PDGs is approximately 5,000 additional lines of code.

PDG construction pass. We construct PDGs for Java programs via an interprocedural dataflow analysis. The analysis is flow sensitive for local variables, but for scalability is flow insensitive for heap locations. PDG construction is also context sensitive: a method may be analyzed multiple times, which produces a larger PDG but simplifies precise querying of the resulting PDG (cf. Reps and Rosay [42]).

The PDG construction analysis handles all language features of Java 1.6 except reflection and concurrency. We reason about implicit `RuntimeException`s that may be thrown by statements and expressions. We create special PDG nodes to identify control-flow branches due to these exceptions.

The precision of the PDG-construction analysis is improved by several analyses provided by the Accrue framework, including a non-null analysis (which determines when expressions are never null) and a precise exception analysis (which determines the exact types of exceptions that may be thrown by methods). These analyses improve the precision of the control-flow graph on which the PDG-construction dataflow analysis is performed.

Library functions. For both the pointer analysis and PDG construction we analyze the source code of most of the JDK at the same time we analyze the application code to ensure precision with respect to library functions. We use signatures for some of the core classes (e.g. `java.lang.String` and `java.lang.System`) in order to improve efficiency and precision. Signatures directly describe the analysis results for methods and constructors instead of examining the relevant source code. In addition, we provide analysis result signatures for some native methods. For native methods without signatures and library functions for which we do not have source code, we assume that the return values of the meth-

[3]For presentation purposes, we syntactically distinguish graph functions and policy functions; in the implementation using a policy function where a graph function is expected will result in an evaluation error rather than a parsing error.

[4]While a single pointer analysis was sufficient for the case studies in Section 6. Accrue can be run with different pointer analyses to trade-off between precision and run time.

Program	Policy	Time (s)		Policy
		Mean	SD	LoC
CMS	B1	5.3	0.6	3
	B2	4.1	0.05	5
FreeCS	C1	4.0	0.08	6
	C2	26.9	0.2	37
UPM	D1	6.6	0.1	6
	D2	6.2	0.04	4
	D3	5.0	0.06	11
Game	E1	0.043	0.001	3
	E2	0.41	0.01	4
PTax	F1	0.67	0.01	4
	F2	0.40	0.01	4
	F3	2.0	0.03	7
PChair	G1	0.47	0.01	13
	G2	0.44	0.01	12

Figure 5: Query evaluation times

ods 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.

5.2 PidginQL Query Engine

We have implemented a custom query engine for our query language PidginQL that evaluates queries against PDGs. We created a custom query engine for flexibility and fast prototyping of the query language. We believe the query language could be implemented on top of an existing graph query language and engine such as Cypher or Gremlin [22].

The query evaluator is written in 5,800 lines of Java code. It implements call-by-need semantics and caches subquery results. This 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 demonstrate the feasibility of our approach by using Pidgin to develop application-specific security policies for off-the-shelf programs and to support new development.

We have applied Pidgin to six Java programs. 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 medium-sized web-based Course Management System (CMS); and two medium-sized open-source applications, Free Chat-Server (FreeCS) and Universal Password Manager (UPM). We also used Pidgin to develop the Guessing Game example in Figure 1. For two small applications we wrote ourselves, PTax and PChair, we used our system to support simultaneous application and policy development. In Appendix A we present the results of running on the SecuriBench Micro benchmark [34]; in which we detect 159 of the 163 vulnerabilities. The diversity and specificity of the policies described below demonstrate the flexibility and expressivity of PidginQL.

6.1 Analysis performance

The first two columns of Figure 4 present the number of non-comment non-blank lines of code and the number of procedures in each program. As described in Section 5.1, we analyze JDK source code in addition to application code; numbers in parentheses indicate lines of code and number of procedures including JDK source code.

Figure 4 also summarizes the performance of the pointer and PDG construction analyses for each program, giving the mean and standard deviation (SD) of ten runs. All analyses were performed on a quad core, 2GHz, Intel Core i7 MacBook Pro laptop with 16GB of RAM.

Figure 5 summarizes query evaluation times for all queries discussed in this section, based on ten evaluations. Query times are reported for a cold cache (i.e., with no previously cached results for subqueries). All queries evaluated in under 30 seconds, and all but one in under 7 seconds. The last column in Figure 5 gives the number of lines for each policy. Note that query size depends on the complexity of the policy, but appears to be independent of program size.

6.2 Course Management System (CMS)

CMS [6] 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. CMS uses the model/view/controller design pattern. We examined the security of the model and controller logic; views simply display the final computed 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
pgm.accessControlled(isAdmin, addNotice)
```

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

This five line query is similar to Policy B1.

6.3 Free Chat-Server

Free Chat-Server is an open-source Java chat server that has been downloaded from Sourceforge[5] over 90,000 times. Once the chat server has started, users can send messages to one another, maintain friend lists, create, join and manage group chat sessions, and perform other actions. Administrators can ban, kick, and punish misbehaving users.

Policy C1. *Only "God" users 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 only perform certain 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

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

Program	Size (w/ JDK source)		Pointer Analysis				PDG Construction			
	LoC	Procedures	Time (s)		Nodes	Edges	Time (s)		Nodes	Edges
			Mean	SD			Mean	SD		
CMS	15,508 (466,874)	1,345 (47,019)	550	35.0	368,561	5,062,102	448	23.9	1,907,260	3,651,370
FreeCS	22,149 (448,215)	1,340 (45,077)	126	5.1	335,380	4,281,565	300	13.8	1,688,069	3,290,920
UPM	4,052 (332,396)	532 (32,260)	146	10.5	296,894	4,248,109	366	6.4	1,695,513	3,006,171
Game	30 (166,319)	6 (17,351)	17	1.0	80,946	670,315	34	0.4	422,323	770,291
PTax	262 (167,875)	24 (17,560)	18	1.2	90,360	790,848	36	0.3	481,835	873,028
PChair	320 (166,448)	55 (17,455)	17	1.5	84,030	696,254	35	1.3	438,940	798,292

Figure 4: Program sizes and analysis results

"perform action" method that were not access controlled by the `punished` flag. The final policy is 37 lines of PidginQL, the largest we have developed.

6.4 Universal Password Manager (UPM)

UPM is an open-source Java password management application. 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 from Sourceforge[6] almost 70,000 times.

Policy D1. *A database is opened only after the master password is checked or when creating a new database.*

Method `doOpenDatabaseActions` is called to open the password database. We confirmed that this occurs only in the `newDatabase` method or when protected by appropriate checks of the master password.

Policy D2. *The user's master password entry does not explicitly flow to the GUI, console, or network.*

When we consider only the data dependencies in the program, there is no information flow from the user's password entry to any of the Java Swing GUI classes or any other public output. The PidginQL policy is:

```
let passwordInput = pgm.returnsOf("askUserForPassword") in
let output = pgm.formalsOf("javax.swing") or
             pgm.formalsOf("OutputStream") or
             pgm.formalsOf("HTTPTransport") in
pgm.noExplicitFlows(passwordInput,output)
```

Policy D3. *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). Note that an incorrect or invalid password results in an error dialog box, and our policy accounts for this information flow.

6.5 Guessing Game

We implemented a Java version of the program listed in Figure 1. The following policies, described in more detail in Section 2, hold for the Guessing Game.

Policy E1. *The secret does not depend in any way on the user input.*

Policy E2. *The secret does not influence the output except through the comparison with the user's guess.*

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 (specifically, their salary). 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.*

This can be expressed as the PidginQL policy:

```
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 trusted-declassification policy is similar to Policy E2. 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.*

The PidginQL query is similar to that for Policy F1.

Policy F3. *Tax information is not decrypted unless the user's password is entered correctly.*

Policy F3 is an access control policy, whose exact statement depends on the specification of the `userLogin` method.

6.7 PChair

PChair is a toy conference management system. There are five user roles: author, reviewer, program committee member, program chair, and system administrator. A user may have multiple roles. PChair handles the submission, revision, and reviewing of papers. We analyzed only the backend which maintains and controls access to the review and paper databases.

Access control policies in conference management systems can be intricate and complex. For example, several information leaks have been found and fixed in HotCRP [27]. We used Pidgin throughout development, updating policies as the implementation evolved. Several features were added during development, affecting the functionality of the system and requiring modifications to the PidginQL policies. For example, we added the ability for authors and PC members to list conflicts, and needed to update Policy G1 to prevent PC members from viewing reviews of conflicted papers. In total, we defined fourteen policies for PChair. We present two representative policies here.

Policy G1. *A review can be viewed only by an authorized user.*

Naturally, the complexity of the PidginQL policy for Policy G1 lies in the definition of "authorized user," which is very specific to the conference management application. For example, PC members can access any review after the review submission deadline has passed, unless they have a conflict, and authors of a paper can access reviews for that paper after the notification deadline. The PidginQL describing the

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

condition for safe review access is:

```
let check = pgm.[isAdmin ||
        (isAuthorOf && notifyDeadlinePast) ||
        (isPC && reviewDeadlinePast && !hasConflict) ||
        isReviewerOf]
```

where isAdmin, isAuthorOf, etc refer to the return values of functions with the same names.

An additional policy (not shown) confirms that calls to `getReview` and the access control checks (`isAuthorOf` and `hasConflict`) refer to the same paper, i.e., the same argument is passed to each.

Policy G2. *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.[(isAuthorOf && notifyDeadlinePast) ||
                (isPC && !hasConflict)] 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 status. PIDGIN provided enough information to identify this subtle violation and find the bug, which was easily fixed by checking the notification deadline before checking acceptance status.

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, an object-sensitive and context-sensitive tool for checking noninterference in Java bytecode [18], shown their techniques to be sound [54], and considered information flow in concurrent programs [16]. They have also used path conditions to improve the precision of PDGs, ruling out impossible paths in the PDG [20, 49]. Hammer et al. [19] consider enforcement of a form of *where* declassification [45] using PDGs.

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; and (2) we seek to use the PDG to enable *exploration* of security guarantees of programs in addition to *enforcement* of explicitly specified security guarantees. Existing techniques for improving the precision and scalability of PDGs are applicable to our work. We seek to benefit from these techniques in the future.

Program dependence graphs were introduced by Ferrante et al. [15], along with an algorithm to produce them. PDGs were presented as an ideal data structure for certain intraprocedural optimizations. Program slicing for an interprocedural extension to PDGs is introduced by Horwitz et al. [23]. Program slicing is useful for describing security guarantees and is built into PIDGINQL as primitive expressions forwardSlice and backwardSlice. Reps and Rosay [42] generalize slicing on PDGs and define *program chopping*, of which the PIDGINQL function between, defined in Section 2, is an example. Cartwright and Felleisen [7] formalize PDGs and give a denotational semantics derived from the semantics of the original program. Bergeretti and Carré [5] use structures similar to PDGs to automatically find bugs in *while* programs and increase program understanding.

Legacy applications and policy inference. PIDGIN supports discovering information security guarantees for legacy applications. Rocha et al. [43] present a framework that allows declassification policies to be specified for legacy applications. Policies are separate from code (but, like ours, may refer to program entities such as functions). Enforcement of policies is checked using *expression graphs*, which, like PDGs, capture data and control dependencies. Security policies are specified as graphs that describe which expression graphs can be declassified. The security policies of PIDGIN similarly provide a mechanism to describe what information flows are permitted in a legacy application. Unlike the framework of Rocha et al., PIDGIN supports a rich class of security policies and allows developers to *explore* the information flows in an application, and thus provides support for deciding what security policy is appropriate for an application. By contrast, Rocha et al. only discuss declassification and do not consider how developers produce appropriate security policies. Moreover, we have implemented our approach for the Java programming language; to the best of our knowledge, Rocha et al. do not implement their framework, nor consider how to extend their techniques to a full-fledged programming language.

Other work seeks to infer security policies for existing programs. Vaughan and Chong [52] 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. [26], Pottier and Conchon [41], Smith and Thober [47], and the Jif compiler [38, 39] all perform various forms of type inference for security-typed languages. Mastroeni and Banerjee [37] 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 present in a program.

Several analyses infer explicit information flows (e.g., [32, 33, 35]). 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., [39, 46, 53]) are the main technique used to enforce such policies. The survey by Sabelfeld and Myers [44] provides an overview of these security policies and enforcement techniques. More recently, Banerjee et al. [4] combine security-types with an expressive logic for describing a program's declassification policy and Nanevski et al. [40] 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. [21, 36]). Reasoning about the conditions under which potentially dangerous information flows occur is similar to reasoning about *when* declassification is permitted [10, 45]. Restricting attention to only explicit information flows is equivalent to a static taint analysis (e.g., [1, 32, 33, 35, 51]).

8. CONCLUSION

We have designed and implemented PIDGIN, a tool that allows developers to understand how information flows within a program, and to specify and enforce strong information security policies. PIDGIN uses program dependence graphs (PDGs) to capture dependencies in programs, and has an expressive query language that enables exploration and specification of information security.

PIDGIN supports the Java programming language, but the techniques are applicable to other languages. Indeed, we have generated PDGs for C/C++ programs by analyzing LLVM bitcode [28] produced by the clang compiler [11], and explored information security in these programs using the same query language and query evaluation engine.

We have used PIDGIN both to explore the information security of legacy applications and to specify and enforce information security during development. For each application, we were able to express (and verify enforcement of) interesting application-specific security policies, some of which would have been difficult or impossible to express using existing tools and techniques.

We believe that this approach has the potential to make strong information security guarantees accessible to non-security specialists.

Acknowledgments

We thank Andrew Myers for providing access to a Java version of CMS. This research is supported in part by the National Science Foundation under Grant No. 1054172.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, 2014.

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

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

[4] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. 2008 IEEE Symposium on Security and Privacy*, 2008.

[5] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 1985.

[6] Chavdar Botev, Hubert Chao, Theodore Chao, Yim Cheng, Raymond Doyle, Sergey Grankin, Jon Guarino, Saikat Guha, Pei-Chen Lee, Dan Perry, Christopher Re, Ilya Rifkin, Tingyan Yuan, Dora Abdullah, Kathy Carpenter, David Gries, Dexter Kozen, Andrew Myers, David Schwartz, and Jayavel Shanmugasundaram. Supporting workflow in a course management system. In *Proc. 36th SIGCSE technical symposium on Computer science education*, 2005.

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

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

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

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

[11] clang. clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[13] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol 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.

[15] 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*, 9(3):319–349, 1987.

[16] Dennis Giffhorn and Gregor Snelting. Probabilistic noninterference based on program dependence graphs. Technical Report 6, Karlsruhe Institute of Technology, 2012.

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

[18] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[19] Christian Hammer, Jens Krinke, and Frank Nodes. Intransitive noninterference in dependence graphs. In *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, 2006.

[20] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[21] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2006.

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

[23] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, 1988.

[24] Catalin Hritcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your IFCException are belong to us. In *Procedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[25] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[26] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proc. International Conference on Information Systems Security*, 2008.

[27] Eddie Kohler. Hot crap! In *Proceedings of the Conference on Organizing Workshops, Conferences, and Symposia for Computer Systems*, 2008.

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

[29] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. *Proc. 11th Annual Asian Computing Science Conference*, pages 75–89, 2006.

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

[31] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proc. ACM SIGOPS Symposium on Operating systems principles*, 2009.

[32] Yin Liu and Ana 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.

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

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

[35] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proc. ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

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

[37] Isabella Mastroeni and Anindya Banerjee. Modelling declassification policies using abstract domain completeness. *Mathematical Structures in Computer Science*, 2011.

[38] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, MIT, 1999.

[39] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2012.

[40] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2), 2013.

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

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

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

[44] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[45] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. 18th IEEE Computer Security Foundations Workshop*, 2005.

[46] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003.

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

[48] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proc. 4th ACM Symposium on Haskell*, 2011.

[49] Mana Taghdiri, Gregor Snelting, and Carsten Sinz. Information flow analysis via path condition refinement. In *International Workshop on Formal Aspects of Security and Trust*, 2010.

[50] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Proc. ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.

[51] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. ANDROMEDA: accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering*, pages 210–225, 2013.

[52] Jeffrey A. Vaughan and Stephen Chong. Inference of expressive declassification policies. In *Proc. 2011 IEEE Symposium on Security and Privacy*, 2011.

[53] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

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

[55] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM Operating Systems Review*, 2011.

APPENDIX

A. 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
Sum	159/163	15

Figure 6: SecuriBench Micro results

To compare with other Java analysis tools, we ran PIDGIN on the SecuriBench Micro [34] 1.08 suite of 123 small test cases. All the tests are smaller than the guessing game of Section 6, and take about the same amount of time to generate the graphs and execute the queries.

We develop PIDGIN queries for each test. We 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 of the tests the policy is a simple noninterference policy stating that sensitive values from an HTTP request should not affect public output. For a few tests there is an allowed implicit flow, and we developed the policies to account for them. Some tests require domain specific policies (e.g., the Sanitizers tests required application-specific declassification policies).

The false positives were caused by known limitations of PIDGIN. Most were caused by failing to track flow through individual array elements. Two Pred tests rely on arithmetic reasoning to identify dead code. The two Strong Update false positives are due to our flow-insensitive tracking of heap locations.

B. 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. As we did not write these programs, we first familiarized ourselves with the source code, and then attempted to develop queries that described security guarantees that the programs offered. We were occasionally surprised when a relatively simple policy failed. We would then inspect the paths that remained (often with the shortestPath operation), which helped us to understand the information flows in the program and refine the query until we had a policy that the program satisfied.

In this section, we illustrate the interactive query and policy generation process by describing in more detail how we developed Policy D2 and Policy D3 for the Universal Password Manager (UPM).

Generate a program dependence graph. Before querying, we first generated a program dependence graph for UPM. As discussed in Section 6, this took about 2.5 minutes for the pointer analysis and 6 minutes to construct the PDG. We do this relatively expensive analysis once, serialize the PDG to disk, and use the same PDG for many queries.

Find sources and sinks. UPM protects a user's passwords by encrypting them with a single master password. The application prompts the user for the master password, and then uses this to decrypt the database containing the user's passwords. If the master password is incorrect, the decryption will fail.

We decided to investigate confidentiality guarantees regarding the master password that is entered by the user. Inspecting the application code, we found that the master password is returned from the `askUserForPassword` method. The return values of this method are *sources*.

```
let srcs = pgm.returnsOf("askUserForPassword") in ...
```

Note that we chose to regard the return values of this method as sensitive sources. In doing so, we are trusting the implementation of `askUserForPassword` to correctly handle data received from the input: a Java Swing widget. We could use PIDGIN to learn more about how the password is handled by this method, but `askUserForPassword` is 11 lines of code and uses standard Java Swing API calls to create a dialog box with a password field, so we inspected it by hand. 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 (via `java.io.PrintStream`); and 3) the network (via a custom `java.net.HTTPTransport` class). Formal arguments to methods in these three locations are *sinks*.

```
let sinks = pgm.formalsOf("javax.swing*")
          ∪ pgm.formalsOf("PrintStream.print*")
          ∪ pgm.formalsOf("HTTPTransport*") in ...
```

Try simple queries. We first checked if there are *any* paths between the sources and sinks.

```
pgm.between(sources, sinks)
```

This results in a subgraph of almost 1,000,000 nodes, more than half the original PDG. The large size of this graph is due to paths through JDK library code, and inherent imprecision in the PDG analysis. (When graphs are too large, the PIDGINQL user interface automatically presents a textual summary of the graph, rather than a more data-rich presentation.)

We narrowed our focus to just data dependencies, and tried another simple query.

```
pgm.noExplicitFlows(srcs, sinks)
```

This query evaluated to the empty graph: there are no explicit information flows from the password to public outputs.

Investigate counterexamples. The above policy gives us a security guarantee regarding how the application handles the master password, but we wanted to find a stronger policy explaining how the control dependencies allow information about the master password to leak from the application. To begin this process, we found a counterexample by using the shortestPath operation.

```
pgm.shortestPath(sources, sinks)
```

The resulting path (ending in the Java Swing library) consists of 8 edges and contains a CD edge from the node for the Java expression `password == null`. This means that on this path, output to the GUI may reveal whether the master password is `null` (perhaps due to the user not entering any text). Based on our understanding of the application, this is correct functionality so this expression is a valid declassifier.

After removing flows described above, the shortest remaining counterexample is caused by a `NullPointerException` that is implicitly thrown when the master password is null at runtime (i.e., when accessing a method of the character array, `password`, if `password` is null then the exception is thrown). We define a function to find implicit null pointer exceptions in the prelude. We include it here for completeness (lines 9-13 of Figure 7). Similar to the above, this does not reveal useful information about the user's master password, so this is another declassifier.

All remaining implicit flows are through the *control flow* of the decryption function, which uses the master password to decrypt the database containing a user's passwords. In this function, control flow branches depending on whether the password was correct, either throwing an exception or successfully decrypting the database. All that is revealed about the master password is whether the decryption was successful. Therefore, we find the program counter node corresponding to the entry point of the decryption function (line 17 of Figure 7) and add this to our declassifiers.

Create a PidginQL policy. The final policy is shown in Figure 7. This policy was developed incrementally and interactively. Whereas the informal description of Policy D3 is vague, the PIDGINQL policy is a strong, precise, checkable policy that clarifies which information flows from the master password to public output are appropriate.

C. 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

```
1  let srcs = pgm.returnsOf("askUserForPassword") in
2  let sinks = pgm.formalsOf("javax.swing*")
3              ∪ pgm.formalsOf("PrintStream.print*")
4              ∪ pgm.formalsOf("HTTPTransport*") in
5
6  let nullCheck =
7      pgm.forExpression("password == null") in
8
9  let findImplicitNull(G, possibleNullExpr) =
10     let copies = pgm.selectEdges(COPY)
11                      .forwardSlice(possibleNullExpr) in
12     pgm.forwardSlice(copies,1)
13         .selectNodes("IMPLICIT_NULL_CHECK") in
14
15 let implicitNullChecks = pgm.findImplicitNull(srcs) in
16
17 let decryptEntry = pdg.entries("decrypt") in
18
19 let declassifiers = nullCheck ∪
20                     implicitNullChecks ∪
21                     decryptEntry in
22 pgm.declassifies(declassifiers, srcs, sinks)
```

Figure 7: PIDGINQL policy expressing Policy D3

along with the informal security specification and the code itself. PIDGIN policies can be used for regression testing to ensure that changes to the code do not cause policy violations.

We illustrate this process by describing the use of PIDGIN throughout the development of a conference management system, PChair. In the end there were fourteen separate PIDGIN security policies for PChair. These policies either restrict access to sensitive data (author names, paper content, reviews, etc.) or ensure proper permissions for sensitive operations (e.g. accepting a paper or moving a deadline), along with a single trusted declassification policy (PC members can learn *if* they have a conflict even if they cannot see the conflicting paper). This exposition focuses on Policy G2 discussed in Section 6, which specifies when information about paper reviews may be revealed.

Define an informal policy. Before beginning development we wrote down the policies we desired informally. Policy G2 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 application and Pidgin policy. We implemented PChair using role-based access control, as this closely mirrored our informal specification. 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 simple and easy to understand.

The initial version of Policy G2 directly implements the informal specification:

```
... // output = errors or responses sent to the client
let isAuthorOf = pgm.returnsOf("isAuthorOf") in
let isPC = pgm.returnsOf("isPCMember") in
let isReviewer = pgm.returnsOf("isReviewer") in
let getReview = pgm.returnsOf("getReview") in
let check = pgm.[isAuthor || isPC || isReviewer] in
pgm.flowAccessControlled(check, getReview, output)
```

We first gather the possible outputs, messages sent to the client. Then find the return values for the access checks and ensure that at least one of them is true on all flows from `getReview` to the client. The only way to access a review is by calling `getReview` (a simple PIDGIN sub-policy ensures that this is the case).

Update policies when the specification is modified. We iteratively added new features to PChair during development. 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, which required changes to many of our informal specifications and PIDGIN policies: the informal specification for Policy G2 became: *Only authors of a paper, reviewers of a paper, PC members, and systems administrators can see a paper's reviews.* The access control check in the PIDGINQL policy added `isAdmin` as an acceptable condition.

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 used PIDGIN to check enforcement of security policies whenever new code was committed to our source repository. The commit would fail unless all security policy checks succeeded. Thus, as functionality evolved, the PIDGINQL policies were required to evolve with them.

This automated regression testing of security policies was useful several times. In one case, due to incorrect refactoring of a security-relevant piece of functionality (a missing negation), a security policy failed. Timely notification of the security policy failure allowed us to easily identify and fix the security violation. In another case, changing the name of a method in the code but not the security policy caused a security policy to fail with an evaluation error (when a `returnsOf` operation evaluated to an empty graph), requiring us to ensure that the security policy was up to date with respect to the code.

The final PIDGINQL policy for Policy G2 is shown below, and accounts for additional functionality added to the application, including notification deadlines and recording reviewer/paper conflicts.

```
... // output = errors or responses sent to the client
let isAuthorOf = pgm.returnsOf("isAuthorOf") in
let isPC = pgm.returnsOf("isPCMember") in
let isReviewerOf = pgm.returnsOf("isReviewerOf") in
let isAdmin = pgm.returnsOf("isAdmin") in
let notifyDeadlinePast =
    pgm.returnsOf("notifyDeadlinePassed") in
let reviewDeadlinePast =
    pgm.returnsOf("reviewDeadlinePassed") in
let hasConflict = pgm.returnsOf("hasConflict") in
let getReview = pgm.entriesOf("getReview") in
let check = pgm.[isAdmin ||
        (isAuthorOf && notifyDeadlinePast) ||
        (isPC && reviewDeadlinePast && !hasConflict ||
        isReviewerOf)] in
pgm.flowAccessControlled(check, getReview, output)
```