Privacy-Preserving Programming Using Sython

Michael Gaiman, Rahul Simha and Bhagirath Narahari
The George Washington University
Washington, DC 20052
{mgaiman, simha, narahari}@gwu.edu

Abstract— **Programmers often have access to confidential data that are not strictly needed for program development. Broad priveleges from accounts given to programmers allow them to view files, database table entries or even variables in team members' code that are not critical to their own code. The risk inherent in such unchecked access to possibly private and sensitive data is exacerbated in cases where software development is part of a larger functioning system with data already in place, and is especially severe in cases where development is contracted out to third parties. This paper focuses on the problem of providing developers with a programming language that incorporates privacy-preserving features. We present Sython, a preliminary prototype based on the Python programming language that incorporates such features, examining both implementation and the appearance of the system as viewed by a programmer. The main purpose of this paper is to explore the use of language syntax and underlying support for secure variables so that data owners can contract out programming tasks without worrying about information leakage.**

Keywords: Privacy, software security, programming languages, security, information-flow security.

Contact author: Rahul Simha, The George Washington University

I. Introduction

For the past 50 years, software development has been characterized by a programming culture that prefers minimal restrictions on access to either machine internals or data. Such freedom to view or manipulate data is even celebrated in programming languages like C and in most early operating systems. Unfortunately, this "programmer knows best" culture clashes directly with the increasingly strong line taken by owners of data who want access to be tightly controlled and audited. Several factors have recently contributed to a heightened need to restrict access including: sensational stories of data theft reported in the media [1], a growing legal framework to deal with privacy issues (examples: HIPAA [2], TBPA [5], the European Data Protection Directive [3], [4]), and most recently, the large-scale outsourcing of software development to third parties [46].

This paper presents Sython, a prototype enhancement of the well-known Python programming language [37] to enable programmers to develop software while restricting their access to those data specified by the owners as private. The purpose is to allow software development to be contracted out while maintaining the privacy of data in-house. A key goal of the project is to retain programming flexibility and the expressiveness of the programming language while safeguarding data. In particular, programmers ought to be able to program as with a standard language, and debug their programs as if they were using the actual data, but without actually seeing the data.

Sython provides an approach to use sensitive data during the development and testing cycles without actually exposing the actual values of the data to any third parties (whether trusted, semi-trusted, or untrusted). This is accomplished by separating sensitive data from other non-sensitive data and then storing sensitive data in a completely different process and on a logically different machine. Data in secure variables is randomized before a programmer can view the contents for the purpose of debugging. The separation is likely to be physical in practice and therefore provides a level of physical security during development as opposed to traditional systems in which data values are stored in the executing process even if abstracted away as part of an *abstract data-type*) or behind an Application Programmer Interface (API).

The rest of this paper is organized as follows. Section II provides background by reviewing related work. Section III discusses the overall architecture of Sython. Sections IV and V describes the current prototype implementation of Sython. Section VI contains a discussion and points out potential applications. The last section is a summary of the paper along with suggestions for future work.

II. Related Work

This paper's focus is at the intersection of security and programming languages. Both sub-disciplines are established enough not to require any review here. However, the intersection is relatively recent and itself features several different threads of active research. Before describing these research efforts, we note that, in general, programming languages could be said to provide certain kinds of restrictions on data usage. For example, the `private` declaration modifier in Java [29] hides access to data from other programmers. In this case, however, it is merely a mechanism geared towards the software engineering goal of encapsulation rather than to prevent programmers from viewing the data. Note that the data itself lies within the same process and can be accessed directly in memory; thus, a C program that is loaded along with the Java Virtual Machine can freely examine any variable.

A variety of independent efforts over the years in the general area of computer security [9] have sought to address problems such as role-based access control [21], file access control in operating systems [43], or API support for implementing security [38]. These efforts are complementary to our work and are neither aimed at programming language design nor, in many cases, even software development.

Closer to our efforts are those that fall in the domain of programming languages and security, an area that has seen a sharp growth in research this past decade. These efforts tackle issues from enhancing safety in languages like C to obfuscating compilers. We now briefly review some of this work. Several papers contain reviews of this material, for example [13], [17], [22], [44], [49].

Compilers are used for watermarking code and for code obfuscation that is aimed at protecting intellectual property in distributed executables; a survey and taxonomy of such techniques is presented in [14], [15], [16]. Compiler analysis of code is also used to insert additional snippets of code that perform checksums to assess code integrity and reveal tampering [11], [28]. Similarly, compilers are used to instrument code with hidden keys that can be checked in hardware [50]. This type of work is aimed at transforming code to prevent code understanding and tampering, and is not focused on addressing privacy concerns.

A large body of work has emerged in the area of static source analysis – see the survey in [17]. Among these are tools [12], [23], [48] that examine code for potential security holes. Similarly, there are also numerous runtime-system tools, such as StackGuard [47], that are aimed at preventing certain kinds of attacks. Proof-carrying code (PCC) [7], [36] is another tool that can be incorporated into the compilation process to help identify and guarantee safety properties of code.

Another type of research in this general area, one that is somewhat closer to ours, includes efforts aimed at language design and runtime support for security. These include modifications to the C programming language [30], [35], typed assembly language [25], [32] and runtime support for ensuring safety [20]. All of the above efforts are focused on protecting software from tampering, protecting developers from inadvertently creating attack opportunities, and protecting the system from malicious software.

Among the papers that explicitly consider proper access to data are those that commonly fall under the banner of *information flow security* [31], [34], [33], [39], [41]. This term has been used to refer to the spread of information among the various components of a large software system (including files, operating system, user memory and applications). These papers focus on the larger issue of data access in software systems and their solution approach is to perform static analysis on code. Myers [34], for example, describes their implementation of JFlow a Java-based information-flow security implementation. In contrast, our approach is to let the programmer write code in any which way, but to dynamically enforce privacy preservation through physical separation. Our focus is also somewhat more narrow – we are interested in new programming language features that enable restricted access to variables by the programmer for the explicit purpose of data privacy. Our model has only two players: the data owner who contracts out programming work, and the programmer. We will use the term *privacy-preserving programming* to distinguish our framework from the more general, and complementary, information-flow security paradigms mentioned above.

Note that the concept of data privacy in software development is hardly new. Database systems have provided *views* for several decades [19]. However, such views typically hide the data, do not permit programmer access, and don't generally allow the programmer to develop and debug applications that manipulate the data. Furthermore, these systems are often proprietary and define views in terms of database tables. They are therefore more restrictive than simply declaring any kind of variable in a program as private. Finally, in the area of programming languages, abstract data types hide actual variables and use methods to enforce proper use. However, their purpose is not privacy because these values are set or retrieved using methods; instead, abstract data types allow for checking of values, and for hiding the implementation from other programmers. Note that our work is also quite different from the relatively recent area of privacy-preserving data mining [6], [45] which is mainly concerned with providing accurate aggregate properties of data while preventing inference of the actual data values.

III. Sython: Overview

Sython[1] is an extension to the Python [37] programming language. It extends Python's syntax to denote a variable as *secure*, which is then stored in a physically separate location controlled by the data owner; in some circumstances, an owner might allow separation to be enforced by operating system processes. Sython also adds a number of new built-in functions that support the creation of, and operations on, secure variables. These are variables controlled by the data owner, whose contents are not visible to the programmers that manipulate such variables in the course of their programming work. Operations on such variables are restricted so that programmers may not infer their contents. This property is sometimes called noninference or noninterference [24], [39]. In our current prototype implementation, secure variables are limited to integer and string types.

Figure 1 shows a data owner on the right, separated from a contract programmer on the left. The programmer runs the Sython interpreter that, except for handling secure data, is a

[1]A contraction of "Secure" and "Python"

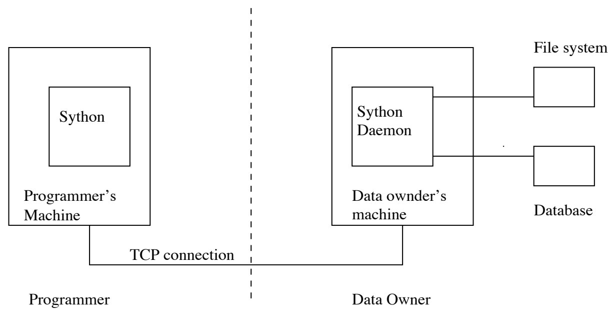

Fig. 1. Programmer vs. Data Owner

superset of the Python interpreter. In our prototype, the owner has three kinds of data: private data that resides in files, private data that resides in a database, and standard (regular) data that is accessible to the programmer. Sython accomplishes the storage of, and operations on, secure data by creating a new support program – the owner's system runs `sythond`, the Sython daemon that handles all requests from the Sython interpreter that runs on the programmer's machine. The two components, client and server, are connected using a standard TCP connection.

The Sython daemon that executes on the owner's machine is a background application that waits for commands from the interpreter that runs on the programmer's machine. These commands are for creating and manipulating secure variables. Non-secure or regular variables, on the other hand, live inside the interpreter on programmer's machine. When a request arrives at the server, `Sythond` then performs the requested operation (or reports it as invalid) and any results are returned to the Sython interpreter. Note that Brumley and Song [10] take a similar approach in their privilege separation design where privileged operations are performed in one process while non-privileged operations are performed in a separate process. Our focus instead is to provide this type of access within a programming language.

In addition to creating secure variables as directed by the programmer, `sythond` provides mechanisms for input and output of secure variables using the actual private values on the owner's machine. At this time, our prototype supports data that is input from files as well as data extracted from the SQLite [27] relational database. Results of queries run against the database can be stored in secure variables. `Sythond` can also receive input from a file and write output to a file. These files reside on the data owner's machine that runs `sythond` and are therefore private. Several new built-in Python (Sython) functions have been defined for use with secure variable input and output faculties. These new built-in functions are discussed in the next section.

Each instance of the Sython Interpreter has a private namespace within `sythond`. This allows multiple instances of the Sython Interpreter to share one instance of `sythond` concurrently and therefore, can be thought of as a client-server architecture, where the clients are Sython interpreter instances and `sythond` acts as the server.

IV. Sython Modifications to Python

Sython makes a small number of significant changes to the Python programming language and its libraries. Sython defines new syntax to visually separate secure from regular variables. In Sython all secure variables are denoted by a dollar-sign preceding the variable handle like so:

$$\$secVar = secVal \qquad (1)$$

The dollar-sign is only needed when initially declaring variables as secure, but we consider it good programming practice to use the dollar-sign whenever making an assignment.

```
1.    $x=syalloc('i')
2.    x+=1
3.    y=2
4.    x+=y
5.    y+=1
```

Fig. 2. Example Sython source code. Allocates a new secure integer `x` and adds one to it. `y` is a regular variable with the value `2`, which is added to `x`. Then `y` is incremented by one. Lines 1,2 and 4 are executed using secure variables stored in `sythond`.

```
...
$user = syinput()
$responses = syquery('select id from Users \
              where username=%s',$user)
$id = $responses[0][0]
$responses = syquery('select times_accessed \
              from Access where id=%i', $id)
$times_accessed = $responses[0][0] + 1
syquery('update Access set times_accessed=%i \
  where id=%i', $times_accessed, $id)
count+=1
...
```

Fig. 3. Example Sython source code with database access.

Figure 2 shows a simple (if useless) Sython program that illustrates the allocation of secure variables.

Notice that in Figure 2 the results of the function `syalloc()` are initially assigned to the secure variable. `syalloc()` is one of five new built-in functions. `syalloc()` requests that space be allocated in the current interpreter's namespace within `sythond`. In addition, `syalloc()` also sets an initial value which is randomly assigned from the set of possible values in the variable's domain of values. In the current implementation, Sython supports even, odd, positive, and negative as integer sets and uppercase, lowercase, punctuation, whitespace, and numerical digits as string sets. Sython also allows a specific range of values within the set from which the initial variable value is chosen. For example, an integer could be specified to be odd and in the range of 1 to 50. A possible initial value for that variable would be forty-five. Ranges are checked to make sure that they are not too narrow to enable unwanted inference of the probable actual value. For example, a range with fewer than ten elements could be considered too narrow in some applications.

Figure 3 shows sample code that accesses private data in databases tables. In this example, the user names are accessed privately from a file through a new function, `syinput()`); these names are run against the table `Users` through the function `syquery()` and used to update a value from a second table (`Access`). The only value that the programmer sees is the non-secure `count` variable. This example illustrates how secure variables are populated and how queries are run against private data.

The functions `syinput()` and `syoutput()` provide file input and output for secure variables. The function `syinput()` is one mechanism by which data owners initialize the secure variables that are later manipulated by the programmer. Similarly, programmers can write out contents of secure variables to a file using `syoutput()`. In some sense, the programmer's contract is to compute on the input and produce the desired output without being able to view the actual contents of secure variables. The system allows for any manipulation of non-secure variables as is customary in any programming language.

To enable input and output of secure Sython data with a database, `syquery()` is a built-in function that interacts with the SQLite database that is optionally part of `sythond`. `syquery()` stores secure variables to tables within the database as well as reads query results into secure variables. The query language is currently a stripped down version of SQL, and does not support numerical and some other types of queries.

Another new built-in function is `syval()`. `syval()` returns a string representation of a secure variable that is useful for debugging purposes. Because this representation is randomly created, it does not reveal the actual value to the programmer, but is useful for debugging. The string representation for a given secure variable will not change unless the value of the secure variable has changed. Moreover, two secure variables that have the same actual value may or may not have the same `syval()` value.

V. Implementation Details

This section details the implementation of our Sython prototype and might be of interest to readers familiar with Python internals. Sython is based on Python version 2.3.4. The modifications we made to Python were written in a combination of C and Python. Sython modifies the language's grammar to make use of the dollar-sign to denote secure variables. Because the dollar-sign is not used in Python, and because it is used in Perl, we felt it was a reasonable choice to include in Sython. Internally, a secure variable is implemented as a new built-in object type, similar to how other built-in types are implemented in Python. The secure variable type (known internally as `syobj`) supports many operations such as addition, subtraction, string concatenation and division. The implementations of these operations makes use of a Python package that handles communications with `sythond`.

The communications protocol between the Sython interpreter and `sythond` is implemented using standard TCP sockets. It is a custom, stateless protocol that was created for ease of development. Messages are sent from Sython to `sythond` using Python's object serialization facility. Each request is a Python Dictionary object containing some predefined command symbols. These symbols are `V`, `CC`, `cmd`, and `data`. `V` represents version and is currently an integer with the value 1. `CC` is short for client code; it uniquely maps the client to a specific namespace within `sythond`. `cmd` is the current command to be executed and is a string value. `data` carries any payload the command specifies. It is either empty (mapping to `None`) or is another dictionary which contains command specified key-value pairs. Figure 4 shows the messages sent for a specific set of Sython commands.

```
1.    $myInt=syalloc('i',(0,100),('ODD',))
⇒     {CC: None, V: 1, CMD:
      'NEW_SESSION', data: {}}
⇐     59747
⇒     {CC: 59747,V: 1, CMD: 'NEW_VAR',
      data: {vset: ('ODD',), vtype: 'i',
      vrange: (0, 100)}}
⇐     'sy0'
2.    x=123
3.    $myInt=myInt+x
⇒     {CC: 59747, V:1, CMD: 'OPmL',
      data: {arg2: 123, var1: 'sy0',
      op: '+'}}
⇐     'sy1'
⇒     {CC: 59747, V:1, CMD: 'FREE_REF',
      data: {var: 'sy0'}}
⇐     None
```

Fig. 4. Communications requests produced by Sython commands. The first command allocates a new secure integer which can be in the range of 0 to 100 and must be an odd value. This command produces a `NEW_SESSION` request (Sython only communicates with `sythond` when needed, as such a new session is started when the first secure variable is allocated) which returns a client code (`59747` in this session), followed by a `NEW_VAR` request. `sythond` returns `sy0` as the handle for this secure variable. The second command allocates a nonsecure integer, so no `sythond` requests are generated. The third command adds the nonsecure variable to the secure variable and stores it in the secure variable. This command produces two `sythond` requests. The first, the `OPmL` request, asks `sythond` to perform the addition operation on `sy0` and `123`. `sythond` returns a new handle for the result `sy1`. Next, because the result is stored in `myInt`, the original value of `myInt` is deallocated by the `FREE_REF` command, which returns `None`.

Within `sythond`, dictionary objects are used to map client codes to namespaces. Namespaces are also implemented as dictionary objects. Namespaces in `sythond` store needed information about the current state of secure variables. Each secure variable has a key that is unique within the namespace, of the form syN where N is an integer starting at 0 and increasing by one for each new variable. Namespaces also contain supporting information about each secure variable such as the `syval()` representation of a secure variable and type (currently integer or string).

Both the interpreter and `sythond` are themselves written in Python, which offers some advantages when performing operations on secure variables – all operations are carried out using the `eval()` function that Python provides. This function takes a string, evaluates it as a Python expression and returns any results as Python objects. Using a combination of Python's object serialization facility (known as `pickle`) and `eval()`, all operations involving secure variables, even those also involving regular Python objects are carried out by `sythond` on the server.

VI. Discussion and Applications

What makes Sython unique is what programmers are able to do when these tools are provided at the language level.

The access restrictions in Sython will allow for new types of distributed software development where organizations can hire the right developer for the task without worrying about data confidentiality. The onus upon the manager or overseer is to create the private data in files or in the database, and simply mark them as private.

What types of applications might Sython be best suited for? Many database application programmers are either inadvertently given access priveleges or are given a painstakingly transformed database with "junk" values for the purpose of testing. With Sython, the data owner would simply specify some variables as secure and describe how the initial values are populated from the database. Thus, the preparation on the part of the data owner is less burdensome. Even so, one might argue that database systems are already the best-equipped development system to handle privacy. Thus, we posit other types of applications where Sython might be most effective, for example:

- *Medical applications*. In medical applications that involve manipulation of measurement data such as electrocardiogram data or microarray data, such data are usually in files. Even if the patient name is hidden, the data itself may need privacy protection because it is part of a novel treatment or part of a study with very few patients. Also, the data itself has value in corporate espionage. Using Sython, programmers who build, say, a filtering application, can then manipulate such data without actually viewing the values.
- *Voting*. Voting, tallying and audits are examples of applications where data must be manipulated anonymously. Again, simply hiding human-identifying information may not be sufficient, and itself places a burden on the data owner. Using Sython, a programmer can be asked to develop applications that extract key statistics or to develop a well-formatted summary report.
- *Financial applications*. Similar to medical applications, the area of finance is rich with examples that can benefit from privacy-preserving programming. These include applications for data sets too small to provide statistical anonymity.

Note that any manipulation of secure or private data can also be achieved by constructing an elaborate library of remote objects, using for example Java's RMI package [26]. Thus, one might ask whether Sython provides any advantage over such an approach. We believe that there are several advantages:

- *Programmer convenience*. Instead of working with the particular conventions of an API, Sython programmers would simply declare and use variables. This lets the programmer avoid writing long expressions with the full pathlength containing the remote object name and method.
- *Simplification*. The burden of creating a remote object to safeguard access would fall upon each data owner for each application. This is both unnecessary and, furthermore, increases the risk that a mistake in implementing the remote object could inadvertently reveal data. Instead, in Sython, the language itself centralizes the checks and enforces review of operations.
- *Optimization*. By locating access control in the language, performance optimization efforts can be concentrated in the interpreter and server. For example, caching of input/output files on the server can help improve performance by avoiding unnecessary disk operations. Once such optimizations are incorporated into the server, they will impact all applications that use the system.
- *Standardization*. A refined and bug-free future version of Sython could gain acceptance and help set a minimal standard. Data owners would know what they are getting with Sython applications.

Sython is not a one-size-fits-all solution to data protection during development. Not all desired computations can be easily expressed using the operations Sython provides for use with secure data. Sython disallows operations that could be used to violate the noninterference properties of the system. In fact, during development several operations that were originally allowed had to be disabled because they could have been used to systematically discover the values of secure data. For example, one exploit involved the string multiplication operation where multiplying a string `s` (with length `l`) and a secure integer `i` would result in a string that is `l*i` long, and reveal the value of `i`. Because Sython allows developers to know the length of secure strings, allowing string multiplication between a string and a secure integer variable then taking the new string length and dividing out the original length produces the value of the secure integer. This was addressed by removing support for string multiplication involving secure integers from the prototype system. Support for the length operation on strings could have been removed instead, but would probably be too restrictive.

Another information flow leak that we discovered occurred as the result of an exception. When two secure variables were divided, a divide-by-zero exception was thrown if the divisor was zero. This result could be taken advantage of by the simple code snippet shown in Figure 5. Because of Sython's noninterference enforcement, some operations are not permitted or the results are randomized and therefore, some application programs may not be suited for use with Sython.

Also, Sython does not wholly eliminate the need for trusted developers. During the development phase, at least one trusted developer must act as manager to their non-trusted counterparts by creating the input/output specifications, developing program correctness tests, reviewing source code and administering the server on which `sythond` resides.

While Sython is Python specific, the Sython development model for privacy-preserving programming is not. Many other languages could well be modified for use with such a model. Python was chosen as the testbed language for the same reasons that Python is used generally: it allowed for rapid prototyping and ease of development. Other languages such as Java, Perl and C++ could be coupled with this sort of noninterference system.

```
$x=syalloc('i',None,('POSITIVE',))
$y=x \#save the initial value of x
count=0
try:
  while 1:
    y-=1
    count+=1
    $z=1/y
except sython.comm.Error:
  print 'the value of \$x is:',count
```

Fig. 5. A script to take advantage of a `ZeroDivisionError` exception to find the value of any (positive) secure integer. This problem was discovered and addressed during development, as discussed in Section VI. This script repeatedly subtracts one from a copy of the target secure variable and then uses it as the divisor in a division operation. A thrown exception points to the ZeroDivisionError and then the variable `count`, which had been incremented at the same rate that the target secure integer was decrementing, contains the initial value of the secure integer. In Sython, this script will now just loop infinitely as division by zero is mapped to 0.

VII. Summary and Future Work

This paper described Sython, an extension to the Python programming language aimed at enabling software development in the presence of strict data privacy requirements during development. Our framework, which we call *privacy-preserving programming*, is suited to a commonly-occuring mode of software development today in which a data owner contracts out specific programming tasks.

Sython is currently an open-source, exploratory reseach prototype; additional work needs to be done before it can be used for actual software development. At this time, we identify a few of these needed additions. Sython currently only supports integer and string types as secure variables, and therefore support for additional types is needed. In the current prototype, declaring function arguments to require secure variables is not yet supported. Secure variables can be passed as arguments and then tested at runtime by the developer, but the more preferable $def foo(\$bar, \$baz)$: syntax is not supported. Improvements could also be made to the way Sython and `sythond` communicate. As described in Section V the current protocol is stateless and does not support any strong authentication of requests via cryptographic primitives such as encryption and certificates.

Other possible areas of work include support for more databases and exploring the ability to graphically interact with trusted users while still ensuring noninterference for non-trusted users, this would allow trusted users to input information into secure variables via a user interface and display the contents of secure variables to the display. Another interesting future area of research is exploring whether features like an operating system or virtual machine layer could allow secure variables to be stored locally, without a daemon application running on a separate machine, while still ensuring noninterference. Our approach invites the use of formal models and theoretical rigor. For example, one might model a variety of security interventions and underlying mechanisms to allow for automated reasoning about a program's privacy guarantees. Similarly, by developing a formal semantics for operations involving secure variables, privacy properties might be rigorously established.

Acknowledgements

This work is partially supported by grant ITR-0325207 from the U.S. National Science Foundation.

References

[1] 40 million credit cards hacked. CNN news story, July 27, 2005. `http://money.cnn.com/2005/06/17/news/master_card/`

[2] The Health Insurance Portability and Accountability Act of 1996 (HIPAA). See `www.hipaa.org`.

[3] The EU Data Protection Directive. *EURIM Briefing No. 12*, July 1996.

[4] D.Beyleveld, D.Townend, S.Rouille-Mirza and J.Wright. The Data Protection Directive and Medical Research Across Europe. *Ashgate Pub.*, 2004.

[5] Taxpayer Browsing Protection Act. To prevent IRS employees from gratuitous snooping of confidential data.

[6] R.Agrawal and R.Srikant. Privacy-preserving data mining. *Proc. of ACM Sigmod*, May 2000.

[7] A.W.Appel and E.W.Felten. Proof-Carrying Authentication., *6th ACM Conference on Computer and Communications Security*, November 1999.

[8] B.Barak, O.Goldreich, R.Impagliazzo, S.Rudich, A.Sahai, S.Vadhan, and K.Yang. On the (im)possibility of obfuscating programs. *Proc. CRYPTO 2001*, August 2001.

[9] M.Bishop, Introduction to computer security. *Addison-Wesley*, 2004.

[10] D.Brumley and D.Song. Privtrans: automatic privilege separation. *USENIX Security Symposium*, 2004.

[11] H.Chang and M.J.Atallah. Protecting software code by guards. *ACM Workshop on Security and Privacy in Digital Rights Management*, Philadelphia, 2001.

[12] H.Chen and D.Wagner. MOPS: An infrastructure for examining security properties of software. *ACM CCS, 2002.*

[13] S.Cheng, P.Litva and A.Main. Trusting DRM software. *Workshop on Digital Rights Management for the Web*, January 2001, France.

[14] C.Collberg, C.Thomborson, and D.Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[15] C.Collberg, C.Thomborson and D.Low. Breaking abstractions and unstructuring data structures. *Proc. IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998.

[16] C. Collberg, and C. Thomborson. Watermarking, Tamper-proofing, Obfuscation: Tools for Software Protection. Technical report 2000-03, University of Arizona, 2000.

[17] C.Cowan. Software Security for Open-Source Systems. *IEEE Security and Privacy*, Jan/Feb 2003, pp. 38-43.

[18] C.Evans, `http://vsftpd.beasts.org/`

[19] C.J.Date. An Introduction to Database Systems, *Addison-Wesley*, 1999.

[20] U.Erlingson and F.B.Schneider. IRM enforcement of java stack inspection. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2000.

[21] D.F.Ferraiolo, D.R.Kuhn and R.Chandramouli. Role-based access control, *Artech House*, 2003.

[22] M. Fisher. Protecting binary executables. *Embedded Systems Programming*, Vol. 13, No. 2, February 2000.

[23] J.S.Foster, M.Fahndrich and A.Aiken. A theory of type qualifiers. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* Atlanta, Georgia, 1999.

[24] J.A.Goguen and J.Meseguer. Security policies and security models. *IEEE Symp. on Security and Privacy*, 1982, pp.11-20.

[25] A.Gordon and D.Smye. Typing a multilanguage intermediate code. *Conference Record of POPL 2001: 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* 248–260, 2001.

[26] W.Grosso. Java RMI. *O'Reilly Pub.*, 2002.

[27] R.Hipp, `http://www.sqlite.org`

[28] B.Horne, L.R.Matheson, C.Sheehan, and R.E.Tarjan. Dynamic self-checking techniques for improved tamper resistance. *ACM Workshop on Security and Privacy in Digital Rights Management*, November 2001.

[29] The Java programming language. `http://java.sun.com`.

[30] T.Jim, G.Morrisett, D.Grossman, M.Micks, J.Cheney and Y.Wang. Cyclone: a safe dialect of C. *Usenix*, June 2002, Monterrey, CA, pp.275-288.

[31] D.Kozen. Language-based security. *Proc. Conf. Math. Foundations of Computer Science*, September 1999, pp. 284-298.

[32] G.Morrisett, D.Walker, K.Crary and N.Glew. From system F to typed assembly language. *ACM Trans. Prog. Lang.*, Vol. 21, No. 3, pp.528-569, May 1999.

[33] A.Myers and B.Liskov. A decentralized model for information flow control. *Proc. Oper. Sys. Principles*, Oct 1997.

[34] A.Myers, Jflow: Practical mostly-static information flow control. *Proc. Symposium on Principles of Programming Languages*, 1999.

[35] G.Necula, S.McPeak and W.Weimer. CCured: typesafe retrofitting of legacy code. *Principles of Programming Languages*, 2002, pp. 128-139.

[36] G.Necula. Proof-Carrying Code, *Proceedings of POPL'97*.

[37] G.Van Rossum. `www.python.org`.

[38] S.Oaks. Java security, *O'Reilly*, 1998.

[39] A.Sabelfeld and A.Myers, "Language-Based Information-Flow Security", *IEEE Journal on Selected Areas In Communications*, Vol. 21, No. 1, January 2003

[40] J.Saltzer, Protection and the control of information. *Communications of the ACM*, Vol. 17, No. 7, July 1974.

[41] F.B.Schneider, G.Morrisett and R.Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, Lecture Notes in Computer Science, Vol. 2000, Springer-Verlag, pp. 86-101.

[42] U.Shankar, K. Talwar, J.S.Foster and D.Wagner. Detecting format string vulnerabilities with type qualifiers. *10th Usenix Security Symposium,* 2001.

[43] G.O'Shea. Security in operating systems, *Blackwell*, 1992.

[44] R.Simha, A.Choudhary, B.Narahari and J.Zambreno. An overview of security-driven compilation. *Workshop on New Horizons in Compilers*, Bangalore, India, December 2004.

[45] V.S.Verykios, E.Bertino, I.N.Fovino, L.P.Provenza, Y.Saygin and Y.Theodoridis. State-of-the-art in privacy preserving data mining. *SIGMOD Record*, 2004.

[46] J.Vijayan, Offshore outsourcing poses privacy perils. *ComputerWorld*, Feb 20, 2004.

[47] P.Wagle and C.Cowan. Stackguard: Simple stack smash protection for GCC. *Proc. of the GCC Developers Summit,* 243–256, 2003.

[48] D.Wagner, J.Foster, E.A.Brewer and A.Aiken. A first step towards autoamted detection of buffer overrun vulnerabilities. *Proc. Network and Distributed Systems Security,NDSS 2000*.

[49] J.Wyant. Establishing security requirements for more effective and scalable DRM solutions. *Workshop on Digital Rights Management for the Web*, January 2001.

[50] J.Zambreno, A.Choudhary, R.Simha, B.Narahari and N.Memon. SAFE-OPS: A Compiler/Architecture Approach to Embedded Software Security. *ACM Trans. Embedded Computing*, accepted.