A PROGRAM PARTITIONING TECHNIQUE FOR ENFORCEMENT OF
CONFIDENTIALITY POLICIES

BY
TEJAS R. KHATIWALA
B.E., South Gujarat University at Surat, 2003.

THESIS
Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2006

Chicago, Illinois
To my parents and their fathers.
ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. V.N. Venkatakrishnan, for his guidance and patience during the course of the thesis. It would not have been possible to reach this stage but for his support. I would like to thank my thesis committee members - Prof. Jon Solworth and Prof. A. Prasad Sistla - for their time and support.

I would specially like to thank my friend Raj for many engaging discussions and giving a helping hand with development which helped me refine our implementation.

TRK
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>APPROACH OVERVIEW</td>
</tr>
<tr>
<td>2.1</td>
<td>System overview</td>
</tr>
<tr>
<td>2.2</td>
<td>Security Policy Specifications</td>
</tr>
<tr>
<td>2.3</td>
<td>Analysis</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>PARTITIONING</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction and generation of private zone</td>
</tr>
<tr>
<td>3.2</td>
<td>Generation of public zone</td>
</tr>
<tr>
<td>3.3</td>
<td>Analysis for state exchange between zones</td>
</tr>
<tr>
<td>3.4</td>
<td>Generation of checks that makes use of runtime information.</td>
</tr>
<tr>
<td>3.5</td>
<td>Psuedo code for partition engine</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>SERIALIZATION</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Basic idea of serialization</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Serialization and data types</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Pointers</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Arrays</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Structures</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Analysis of recursive structures</td>
</tr>
<tr>
<td>4.1.7</td>
<td>The Messaging Protocol</td>
</tr>
<tr>
<td>4.1.8</td>
<td>The hash-table</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Code generation</td>
</tr>
<tr>
<td>4.2</td>
<td>Limitations</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>EVALUATION</td>
</tr>
<tr>
<td>5.1</td>
<td>Policy enforcement evaluation</td>
</tr>
<tr>
<td>5.1.1</td>
<td><em>Linux-Monitor</em></td>
</tr>
<tr>
<td>5.1.2</td>
<td><em>Htpasswd</em></td>
</tr>
<tr>
<td>5.1.3</td>
<td><em>Mediachat</em></td>
</tr>
<tr>
<td>5.1.4</td>
<td><em>chfn, chsh, passwd</em></td>
</tr>
<tr>
<td>5.2</td>
<td>Evaluation</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Micro benchmark on the domain transfer operation</td>
</tr>
<tr>
<td>5.3</td>
<td>Overall performance measurements</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Performance Improvements</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2 Additional memory overheads</td>
<td>44</td>
</tr>
<tr>
<td>5.3.3 Program sizes</td>
<td>44</td>
</tr>
<tr>
<td>5.4 Security analysis of the approach</td>
<td>45</td>
</tr>
<tr>
<td>6 RELATED WORK</td>
<td>47</td>
</tr>
<tr>
<td>6.1 Sandboxing</td>
<td>47</td>
</tr>
<tr>
<td>6.2 Static analysis</td>
<td>47</td>
</tr>
<tr>
<td>6.3 Information flow analysis</td>
<td>47</td>
</tr>
<tr>
<td>6.4 Taint based approaches</td>
<td>48</td>
</tr>
<tr>
<td>6.5 Program partitioning approaches</td>
<td>48</td>
</tr>
<tr>
<td>7 FUTURE WORK</td>
<td>51</td>
</tr>
<tr>
<td>7.1 Function-Pointers</td>
<td>51</td>
</tr>
<tr>
<td>7.2 Optimization over size of private zone</td>
<td>52</td>
</tr>
<tr>
<td>7.3 Better approximation for variable sizes</td>
<td>53</td>
</tr>
<tr>
<td>7.4 Maintaining reference change to array index</td>
<td>54</td>
</tr>
<tr>
<td>8 CONCLUSION</td>
<td>55</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>56</td>
</tr>
<tr>
<td>VITA</td>
<td>59</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>TABLES MAINTAINED BY PUBLIC ZONE AND PRIVATE ZONE</td>
</tr>
<tr>
<td>II</td>
<td>TABLES MAINTAINED BY PUBLIC ZONE AND PRIVATE ZONE FOR FUNCTION POINTERS</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>17</td>
<td>43</td>
</tr>
<tr>
<td>18</td>
<td>45</td>
</tr>
<tr>
<td>19</td>
<td>52</td>
</tr>
</tbody>
</table>

viii
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Better approximation for variable sizes</td>
<td>53</td>
</tr>
<tr>
<td>21</td>
<td>Maintaining reference to an array index</td>
<td>54</td>
</tr>
</tbody>
</table>
LIST OF ABBREVIATIONS

CIL  C Intermediate Language
SUMMARY

When an application reads private / sensitive information and subsequently communicates on an output channel such as a public file or a network connection, how can we ensure that the data written is free of private information? This thesis address this question in a practical setting through the use of a technique that we call “data sandboxing”. Essentially, data sandboxing is implemented using the popular technique of system call interposition to mediate output channels used by a program. To distinguish between private and public data, the program is partitioned into two: one that contains all the instructions that handle sensitive data and the other containing the rest of the instructions. This partitioning is performed based on techniques from program slicing. When run together, these two programs collectively replace the original program. To address confidentiality, these programs are sandboxed with different system call interposition based policies. This thesis discusses the design and implementation of a tool that enforces confidentiality policies on C programs using this technique. It also reports our experiences in using our tool over several programs that handle confidential data.
CHAPTER 1

INTRODUCTION

Sandboxing (or run-time monitoring) is a powerful and practical technique to protect an application that receives input from external sources. Often, these sources are not trustworthy and may send input that may victimize these programs. By monitoring the code for adherence to a specific policy, sandboxing prevents these attacks from being successful.

Several forms of sensitive information are handled by these programs during their execution. When users initiate authorization routines in such program for requesting various services, the request is usually accompanied by authentication data such as passwords and cryptographic key passphrases. In addition, the request may also include other sensitive information such as as credit card numbers. Such sensitive information is typically used to ensure that service requests are granted only to users with the appropriate privileges. Therefore, protection of such information during program execution is an important problem. Needless to say, such sensitive information are attractive targets for local/remote attackers, who look for ways to obtain services from the system. Once an attacker obtains such information, it is straightforward to masquerade as a user, and obtain further privileges on the system. Furthermore, the same piece of sensitive information (such as a common NIS password) could be used to obtain access to other hosts on the network. Combined with the fact that many users typically tend to use the same passwords/phrases for several hosts/websites that they have accounts in, disclosure of such information may lead to compromises of other hosts as well. Hence, it is critical that such information be protected from any unauthorized disclosure.
Such authentication and authorization information are critical pieces of information that distinguish a particular user from an attacker (trying to masquerade as the user). The security of the system is therefore crucially dependent on the secrecy of storing, retrieving and processing this information.

Sensitive information is usually protected by encryption when stored on external disk storage and is directly inaccessible to anyone who does not have authorized access. However, typically sensitive information enters the program as plaintext several interactive programs in a Unix system, and is subsequently encrypted in the program. The most notable of these programs are the `su` and `passwd` programs, but several others make use of password information as well such as the `openssh` suit of tools, web browsers (that store site-specific information encrypted with a master password), email client (that communicate with an IMAP server), wireless network managers and most servers that perform some form of password authentication and so on. In other cases, it is memorized by the user and provided to the system when requesting the service. Hence it is crucial that sensitive information related to authentication and authorization be adequately protected.

Sensitive information could leak from programs in several possible ways. From a practical perspective, an approach for protecting sensitive data handled by these programs requires defending against common methods used by attackers in obtaining sensitive information from systems programs. Some of the most common techniques used by attackers are summarized below:

- *Use memory corruption vulnerabilities.* Also, programs in type-unsafe languages such as C may have other vulnerabilities such as buffer overflows. A buffer overflow in the program can result

---

1 We note that such information can be obtained from naive users through spoofing, which is an important and difficult problem that remains to be addressed satisfactorily.
in attacker-induced code being executed. If the lifetime of sensitive information is large, attacker-induced code can easily read sensitive information from program, and write them to a location (a file or the network). This can subsequently be read by the attacker.

- *Analyze debugging information.* Another important threat comes from core files and crash reports. Such information contains all the data read and processed by programs. During the middle of application execution, a crash results in the entire application data, (contents of data and stack sections) being written to core files, which can be used for the purposes of debugging. Sensitive information such as passwords and credit card numbers can leak through such crash information. An attacker who has access to such core files, can recover sensitive information through a direct forensic analysis aided with the source code information of the program.

This strongly suggests the need for an approach that addresses protection of sensitive information in systems code written in languages such as the C programming language. We note that information flow analysis techniques are mechanisms that required to analyze the flow of information within the program, amongst the variables in a program. However, from a practical perspective, what matters for the end-user is often an assurance that the data does not become available to other users / outsiders on publicly readable channels such as files and network connections.

The discussion about information flow analysis techniques contrasted with sandboxing techniques that was given above suggest an important point. An approach that compartmentalizes sensitive information (and the code that processes it) from the rest of the program will minimize the risks to sensitive information. In this paper, we describe a new approach based on observation sketched above for protection of sensitive data handled by programs.
The two most common methods of implementing sandboxing are *in-line reference monitoring* (where code that implements the monitor is embedded in the same application) and *interface interposition* (where the monitor code is implemented to run at a specific interface boundary). Inline reference monitoring can technically enforce a richer set of policies when compared to interface interposition, as policies for inline monitors can specify constraints about internal variables of a program. An example of such a policy constraint is “the variable `numfiles` representing the number of files currently opened by a program never becomes more than 50”.

However, for C programs, system call interposition, an instance of the interface interposition approach, is the preferred approach for the following reason:

- **Non-bypassability** Programs that receive untrusted input can result running arbitrary code. Due to the absence of strong memory protection in C, such arbitrary code can bypass the checks placed in them in-line reference monitor. Given the two facts that the operating-system kernel / application boundary is safe, and security-related actions of programs must be ultimately effected via system calls, system call interposition does not usually suffer from the non-bypassability problem when implemented correctly.

System call interposition has therefore enjoyed a lot of attention as a practical technique in several earlier works (1; 2; 3; 4; 5; 6). Mechanisms for implementing policies based on system calls is also a well-studied topic. For instance, the policy listed in the above example can be tracked using a finite-automata based monitor that tracks whether “the number of active open calls made by the program is less than 50”.
The simplicity of policy enforcement in system call interposition comes at a cost. Monitors written for this approach cannot reason about the internal data-flow in a program, since they operate exclusively at the application/OS kernel boundary. Consequently, they tend to be coarse-grained and cannot enforce policies such as confidentiality of private data whose enforcement depends on the internal data flow in the program.

To illustrate this further, let us consider a confidentiality policy for a program that plays music files. It is not unusual to find that such programs periodically connect to an external server, possibly to obtain commercial information or update information. The user considers her music files and preferences sensitive and therefore these are not to be sent to the server. To prevent sensitive information from escaping the program, the sandboxing policy over system calls will need to prevent any write operations to public output channels such as files and network connections. This policy will definitely prevent information from leaking, but is very likely to trigger a loss of functionality with respect to these output operations, even if they involve non-sensitive data. This is due to the fact that many output operations are not related to processing sensitive information. These tasks may involve reading and writing (non-sensitive data) to public files or the network. Since the system call approach does not have internal data flow information between any two system calls, it cannot distinguish between sensitive and non-sensitive information written to output channels.

This thesis is organized as follows: Chapter 2 gives a technical overview of the approach and the various components of our system including analysis. The results of the analysis are used in the partitioning procedure described in Chapter 3. Chapter 4 describes serialization technique that is used during partitioning. Chapter 5 describes our experience in using the tool for various free/open source
utilities. Chapter 6 discusses related work. In chapter 7 we talk about future work for this thesis. In chapter 8 we conclude.
Our approach involves splitting the original program into two partitions. We call these partitions public zone and private zone. The runtime view of these new programs is shown in Figure 1. The private zone is the set of instructions of the original program that operate on potentially sensitive data. The public zone consists of all other instructions. The original program execution is substituted with the execution of these two programs. Whenever the public zone needs to read or process sensitive data, it initiates a domain transfer operation, which involves some IPC operations, resulting in a logical transfer of control to the private zone. These communication operations do not involve sending or receiving sensitive data. The private zone proceeds to execute code that processes sensitive input. Once operations on sensitive objects are done, or when non-sensitive operations need to be performed, a domain transfer is performed again to the public zone. At any further point in the program, if access to sensitive information is required, another pair of domain transfers happen.

As noted in the figure, the public zone is disallowed from reading sensitive input, while the private zone is disallowed from writing to public output channels. Therefore, the private zone is the only component that receives, processes and stores sensitive data handled by the program. It thereby retains sensitive data completely within its address space.

This approach, called data sandboxing, combines the relative merits of inline reference monitoring (ability to incorporate data-flow information in policies) with the practicality of system call interposition (i.e., non-bypassability and easier policy development). The high level idea used in the approach is
similar to program slicing (7). A slice consists of instructions of a program that are affected by a set of variables from some program point, known as the slicing criterion. (We note that our approach always proceeds in the forward direction, starting from the criterion, and our notion of slices is slightly different from the standard notion, as explained in the next section). We identify instructions of the program that act on sensitive information. Based on this analysis, we partition the original program to meet the requirements of the confidentiality policy.
The application of this partitioning technique to the enforcement of a broad class of information flow confidentiality policies is discussed in this paper. These policies prevent private information from leaking from the program. Our technical presentation is focused on enforcement of these policies. Enforcement of other policies that gain precision by using data-flow information in programs is very much possible using our approach. However, we do not discuss them further in this paper. In addition, confidentiality policies also include lattice based multi-level policies (8), but our focus here is on dealing with only two levels: “sensitive” (high) or “public” (low). We also do not consider the effects of covert channels such as implicit flows, timing or storage channels (9). As remarked in (10), additional research in this field is needed to make these techniques applicable to a wide range of programs.

While partitioning can be done manually for each program, it is not a viable option, considering the sake of both correctness and usability. Therefore, we have created a tool that partitions C programs in an automatic manner. Using our approach, users who are programmers can freely modify programs.
running in their systems in order to guarantee confidentiality. End-users can freely benefit from these changes.

Creating separate programs for security reasons may seem to be a heavyweight solution, but is not new to the security community. The idea of privilege separation is based on creating separate programs to provide security assurances, as illustrated by the design of qmail (11) (a very widely used mail transfer agent), and in partitioning of programs such as OpenSSH (12; 13). These projects also illustrate that security assurances outweigh any concerns due to overheads imposed by the partitioning solution. Our approach uses such partitioning techniques for enforcement of confidentiality policies.

Enforcement of policies such as information flow integrity (a policy that prevents untrusted information from flowing to trusted information), is straightforward using techniques from our approach, since these policies have a dual nature with confidentiality policies.

Performing this slicing and program creation operation serves the important purpose of guiding the enforcement mechanism with dataflow information of the relevant variables. So, when the policy is enforced on the new program just created, the enforcement strategy has the additional precision needed to accommodate the data flows in the program.

Our approach works through transforming the source code of the programs that the user wishes to be strengthened with such policies and is designed with the following usability goals in mind:

1. Operating system transparency Our approach is intended to work without any changes to the operating system kernel. An approach that requires changes to the kernel would dissuade users from adopting to it.
2. **Strong compatibility** Any changes made by our approach to programs should be compatible with existing software and libraries. Additionally, it should not require specially modified libraries or relinking. This is to ensure that the approach works without having to make use of redundant libraries or having the risk of making some existing programs incompatible. More importantly, it should not cause failures in previously working applications.

3. **Minimal burden on the user of the tool** Our approach is designed to make minimal burden on the programmer employing our tool. The programmer using our tool must not be required to undergo any effort apart from providing the tool with some initial suggestions.

4. **End-user transparency** The mechanisms employed by the approach should be completely transparent to the end-user. The end user also must not be able to visibly sense the effect of the changes made by the approach. This requires us to ensure *semantic equivalence* between the original and the transformed programs.

The approach described in this paper satisfies all these objectives. Goal 1) and 4) ensure that the approach is sound and easily adaptable by end users, and and goals 2) and 3) ensure that the programmer to modify programs is not under any significant burden. Taken together, these goals are important requirements for an approach to be usable, and the design and trade off decisions we have made in the approach were done keeping these goals in mind. In essence, our approach can be seen as an attempt to bridge the gap between practical approaches such as sandboxing to the realm of policy enforcement for confidentiality policies.
2.1 **System overview**

Figure 2 shows the basic architecture of our approach. A security policy is provided along with the program to an *analyzer*. The analyzer performs data flow analysis and deduces a list of variables (for every program point) that may potentially contain sensitive values. It then performs a second analysis where it identifies the set of sensitive instruction regions in the program. These regions are then taken by the *partitioner* to create the public zone and private zone with appropriate transfer routines that enable the domain transfer operation between these programs. The resulting programs are confined using system call interposition policies at runtime. We describe these operations in the following sections.

2.2 **Security Policy Specifications**

In our approach, security policy specifications specify the confidentiality requirements on the original program. There are two parts to a policy specification: a *runtime* part that is used to confine the program during its execution, and a *static* part that is used by our tool to analyze and partition the program.

*Runtime confinement policy* These are policy specifications over the alphabet of system calls that are enforced on the application when it is executed. Separate policies are written for the public zone and the private zone respectively to meet our confinement objectives. Specifying and enforcing such policies is fairly standard, as illustrated in several past works such as (1) and (14). We do not discuss them any further in this paper.

*Policy specification for static analysis* Static policy specifications are used by our analyzer to infer the flow of sensitive information in the program. These are written based on function prototypes. Let us say `int f(a, b)` is a function, where the parameters `a` and `b` are passed by value. Then the policy
specification for \( f \) may be stated as follows: \( f(\text{high, -}) \rightarrow \text{high} \) and \( f(\text{low, -}) \rightarrow \text{low} \). This means that the return value of \( f \) is sensitive if and only if \( a \) is sensitive. Similar specifications are provided for values provided by reference as they act as outputs. Such input output specifications are used in the following cases:

- **Sensitive input routine specifications** User input enters a program using a function call or a system call that reads from an input channel. For instance, in the `su` program, the function `getpass` reads user input. For this function, while the input is not sensitive, the output is. The analyzer requires these inputs for propagation of sensitive values further in the program. These are specified using the same notation that is used for system calls that is described below.

- **System call specifications** System calls are not analyzable, and hence our approach requires us to specify the input/output semantics of system calls. For instance, a `int read(int fd, void *buf, size_t count)` system call that has the following policy specification: `low read(high, low/high, low)`. It suggests that if the first argument (the file descriptor) is a sensitive value, then the second argument `buf` (passed by reference) points to a sensitive (high) value at the end of the read operation. Similar specifications are added for external library calls that are not analyzed by our tool. Note that we do not require specifications for all system or library calls, but for only those that are used to process sensitive information.

Occasionally, the policy may need to refer to the input program line numbers to annotate the sensitive input processing routines. A typical example is an `open` system call whose file name argument is not statically available. In this case, the programmer needs to provide additional location information of the call in the source file as part of the policy.
We use the above toy example as a running illustration of the ideas used in this paper. It is a simple authentication routine of a program that accepts a 32 bit PIN number as input, compares it to the stored PIN and writes the result of authentication to a public log file.

The policy specifies that the input function `getInput` returns sensitive information, and `SECRET-FILE` is sensitive (high), and `LOG-FILE` is public (low). A correct enforcement of this policy will prevent the PIN information from going to the log file.

### 2.3 Analysis

The objective of the analysis engine is to identify instructions of the program that handle sensitive information. This information is given to the partition engine that partitions the program into two. As input, analysis engine is given a policy specification for various context-functions and variables that are sensitive in the program. Based on this information, the engine propagate the sensitive values across the program and identify all potential variables that may receive sensitive information. It then identifies the set of instructions that act on these variables.
In this work, we have used static analysis engine developed by Raj Swaminathan (15). In general, any static analysis engine that provides with information required by the partition-engine can be used. At the end of analysis the partition engine expects following information from the analysis engine:

1. For given context function(s), identify and taint all sensitive instructions and associated variables.
2. Information about sensitivity of each variable used in context functions(s) at each instruction.

The partition-engine makes use of this information and generates public zone and private zone as described in next section.
CHAPTER 3

PARTITIONING

3.1 Introduction and generation of private zone

The goal of the partitioning routine is to split the program into two working programs, the public zone and private zone. The routine takes as input the set of sensitive instructions $S$ within the context function identified by the analysis routine. Note that this set by itself does not represent the entire set of instructions that are called in the private zone program (though the analysis accounts for the effects of such calls). It then performs static analysis to create the private zone program. This is done in 4 steps:

1. **Transitive closure of function calls** the routine looks for function calls made in $S$, the reason being, a function call instruction may be present in the private zone which would require the function’s definition to be inserted as well. This step needs to be recursive, as the called function can further call other functions that are not tagged. It thus computes the transitive closure of call relationships starting from a context function.

2. **Computing definition of global vars** In this step, the partition routine goes through all the statements of each function in the list generated from the previous step, to detect the presence of any global variable used. The objective of this step is to include definition/initialization of any global variable that is referred to by sensitive statements.

3. **Block Identification** Since the entry of sensitive information into the program, it’s sensitivity propagates through many instructions spreaded across the context-function and most likely need not
be sequential; that is, it is intervened by instructions that are not tainted during the analysis. Conceptually, the part of context-function that holds lifetime of the sensitive information should be moved to a private zone program so that it can be protected. However, as explained earlier such part would contain many unnecessary instructions that is not tainted and would significantly impact private zone’s size as it would grow unnecessarily due to untainted instructions and their transitive-closures. We have instead chosen to identify regions of instructions that exclusively handle sensitive data.

In this step, the partition-engine makes use of information provided by analysis-engine to identify sensitive code regions or blocks. A sensitive code region comprises of one or more sequential marked instruction(s) / statement(s). If control enters the first instruction of the region, it exits only through the last instruction. There can be several such regions. When the partition-engine encounters the first marked instruction, it begins a new block. Every consecutive marked instruction is added to the block. The first non-marked instruction encountered marks the completion of the current sensitive code block. Any intervening unmarked instruction would therefore result in at least two blocks. A branch is usually considered the start of a new block unless the entire branch statement itself is marked.

For our example, two blocks are created. The first block contains instructions in lines 2, 3 and 4 and the second one contains line 6, 7 and 8. Therefore, the operation that opens the (public) LOG-FILE will be performed in the public zone.
Note that, generation of code responsible for sharing program environment between the two programs is done simultaneously during this step. Separate set of instructions are generated for each block and are inserted at the start and end of block.

4. **Generating main() of private zone program** Once the set of sensitive blocks is analyzed, the modules (in public zone) corresponding to the private zone needs to be accommodated in private zone. One intuitive way to do this is to create wrapper functions that enclose the sensitive instruction regions. Whenever a domain transfer is initiated by the public zone, these functions can be invoked by passing them all the non-sensitive values from the public zone.

A problem arises with such an approach concerning how sensitive variables are shared between these wrapper functions. Again, there are two options in this case: a) have them in a scope that is global to all these functions or b) restore these values as a first step in every wrapper function, and save them on exit. However, the first option has a disadvantage of making the sensitive variables accessible to many other functions that don’t require them. The second option has an obvious performance penalty. Therefore, the approach we have taken is to create blocks within the function itself. The sensitive variables that are shared across blocks can now be at the function scope. We use the facility offered by the CIL (16) tool that lifts all local declarations (including block-level) to the function scope level. The private zone’s main() is thus constructed with several blocks that are present in the context function, along with the corresponding local and global variables required by these blocks.
3.2 **Generation of public zone**

The public zone program includes all the statements in the original program, with the exception of the sensitive blocks. Generating program same as original program but excluding sensitive blocks is done in one single step. The partition-engine scans through context-function(s); ignores sensitive instructions and replaces it with code responsible for sharing program environment between the two programs at that point. Also, this program is created with instructions to fork the private zone program and set up a communication channel. Note that, such instructions must be inserted before any sensitive information enters the original program. In the public zone, these instructions must precede any node where a domain transfer operation is initiated to the private zone. This can be achieved by placing these operations in a *dominator* node. The first instruction in the function is clearly a dominator where we have placed these instructions. Similarly the operation for waiting for the private zone needs to be done on a post-dominator node. For functions with multiple return program points, the one-return transformation (16) is a way of getting the post-dominator point. The new public zone program now replaces the original program executable.

3.3 **Analysis for state exchange between zones**

While blocks provide the needed abstraction, it is also necessary to maintain program state across the public zone and private zone. The partition-engine makes use of sensitivity information of each variable at each instruction provided by the analyzer to keep track of the variables that are used in the block. This information is necessary to generate code that provides the private zone with these runtime values. Thus, if $V_B$ is set of variables that are used in block $B$ and $V_{\text{start}}$ is the set sensitive variables
at the start of the block, then the set difference of $V_B$ and $V_{start}$ is the set of values that needs to be communicated from public zone to private zone at runtime.

When a block finishes execution, the program state has to be communicated back to the public zone. By definition of the private zone, note that no sensitive values are needed by the public zone. Hence, only values of updated non-sensitive values need to be returned back to the public zone. If $V_{end}$ is set of sensitive variables at the end of a block, then the set difference of $V_B$ and $V_{end}$ is the set of values that need to be returned to the parent.

3.4 **Generation of checks that makes use of runtime information.**

Whenever the private zone is not processing sensitive information, it blocks on the communication channel for the public zone to initiate the domain transfer operation. It needs to know the right block to execute when it receives a message. However, this information is not statically available, as the parent can initiate a domain transfer from several possible locations, hence we cannot generate static target labels for the private zone. To solve this problem, we generate code that (at runtime) determines the target block to be executed. The public zone supplies this information at runtime, as it knows the block to be executed. The private zone on receiving this message finds the right block to execute. Lines 5 and 6 in the transformed example show the generated code.
```c
1. int pin, cpin, flag;
2. START:
3. msg = read_msg_from_parent();
4. if (msg == TERM) exit(0);
5. else if(msg == BLK1) goto Block1;
6. else if(msg == BLK2) goto Block2;
7. Block1:
8. read_n_demarshal(cpin, pin);
9. pin = getinput();
10. cpin = crypt(pin);
11. fd1 = open(SECRET-FILE, ...);
12. goto START;
13. Block2:
14. read_n_demarshal(flag);
15. stored-pin = read-from-file(fd1, ...);
16. if (cpin == stored-pin) flag = 1;
17. else flag = 0;
18. marshal_n_write(flag);
19. goto START;
```

Figure 4: Transformed private zone

### 3.5 Pseudo code for partition engine

Below is the pseudo code that summarizes the steps described in the section:

Generate private zone:

1. **Find transitive-closure**

   (a) In each context function, mark all tainted instructions for any function call or any reference to function;

   (b) Recursively scan definition of each marked function from above step and mark any function call or any reference to function;
(c) Scan each marked function definition and mark any global variable used and its type;

2. Identify and build blocks

In each context function,

(a) Start a block with occurrence of first tainted instruction;
(b) End the block with last continuous instruction since start of block; For statement containing blocks within them such as if, switch, while if whole statement is not tainted then instruction preceding such statement is marked end of block;
(c) Save each variable used in the block in a buffer $S_{locals}$ (if it is not already there);
(d) Generate de-serialization code for each non-sensitive variable in $S_{locals}$ and insert at the start of the block;
(e) Generate serialization code for each non-sensitive variable in $S_{locals}$ and insert at the end of the block;
(f) Uniquely label the block;

Repeat above steps for each block.

3. Build main

(a) For each variable in $S_{locals}$ include its declaration at function scope;
(b) For each local variable of type char * in $S_{locals}$ for that context function, initialize it to NULL; that is any variable of type char * in $S_{locals}$ that was passed as parameter in context-function will not be initialized;
(c) Build control-flow statement that would transfer control to associated labeled block as instructed from public zone;

(d) Append blocks created from previous step to the control-flow statement;

Generate public zone:

In each context function,

1. Start a block with occurrence of first tainted instruction;

2. End the block with last continuous instruction since start of block; For statement containing blocks within them such as if, switch, while if whole statement is not tainted then instruction preceding such statement is marked end of block;

3. Generate serialization code for each non-sensitive variable used in the block;

4. Generate control flow statement and append it to serialization code;

5. Generate de-serialization code for each non-sensitive variable used in th block;

6. Append de-serialization code to above code and replace the block with this combined code;
CHAPTER 4

SERIALIZATION

In order for execution to be transparent, the domain transfers between public zone and private zone need to share the program environment at the transfer point. However, copying the entire program environment is an inefficient operation, as we need to only transport a subset of the environment of the variables that are relevant to the private zone at that point. Our analysis module provides this information to the partitioning module. The exchange of program state information between the public zone and private zone is done by serializing (also known as Marshalling) the program state into a byte array at that program point, and transferring the result to the private zone. In the private zone, this byte array is de-serialized to reconstruct the information about the environment. We have achieved automatic serialization of a program’s state by generating stub modules for each data type that is exchanged during the domain transfer.

4.1 Implementation

4.1.1 Basic idea of serialization

In our scheme, any variable of any type is serialized in two steps:

- Copy its address to byte array(4 bytes)
- Copy its value to byte array(size of variable)

For example, to serialize an integer will require 8 bytes, a character will require 5 bytes, etc. Figure 5 shows the code to serialize an integer \( i \) and character \( c \).
u_int8_t buffer[4096], *p;
int i, addr;
char c;

memset(buffer, 0, 4096);
p = &buffer[0];

addr = (int)&i;
add_to_address_table(addr); // public zone maintains addresses of
memcpy(p, &addr, sizeof(int)); // variables it sends to private zone
p += sizeof(int);
mempcpy(p, &i, sizeof(i));
p += sizeof(i);

addr = (int)&c;
add_to_address_table(addr); // public zone maintains addresses of
memcpy(p, &addr, sizeof(int)); // variables it sends to private zone
p += sizeof(int);
mempcpy(p, &c, sizeof(c));
p += sizeof(c);

Figure 5: Code to serialize int i and char c

De-serialization is done in same sequence:
u_int8_t buffer[4096], *p;
int i, addr;
char c;

memset(buffer, 0, 4096);
read_from_other_process(buffer);
p = &buffer[0];
memcpy(&addr, (void*)p, sizeof(int));
p += sizeof(int);
add_to_hash((int*)&i, addr); // mapping of i's address in private
memcpy(&i, (void*)p, sizeof(i)); // zone to that in public zone
p += sizeof(i);

memcpy(&addr, (void*)p, sizeof(int));
p += sizeof(int);
add_to_hash((int*)&c, addr); // mapping of c's address in private
memcpy(&c, (void*)p, sizeof(c)); // zone to that in public zone
p += sizeof(c);

Figure 6: Code to de-serialize int i and char c

- Read 4 bytes from byte array and store them in a hash table (that maps address of that variable in public zone to that in private zone. Please see discussion below for more information.)

- Read number of bytes equal to size of that variable into the variable.

Figure 6 the code to de-serialize an integer i and character c in a different process.

Note that the sequence in which variables are serialized and de-serialized are same in both public zone and private zone.

Address information of the variable is accommodated in byte array due to the following fact: although change in value of the variable can be exchanged while preserving semantics of its usage, change
in its reference cannot be exchanged since address of the variable in one process doesn’t hold validity in another process. Consider the code snippet show in Figure 7.

If line 5, 6, and 7 are tainted sensitive (and hence moved to other process( private zone)), it will require to serialize \( y \) and \( z \) to communicate environment. If only value of \( z \) was passed then after domain transfer back to original program( public zone), \( y \) will lose the reference information about \( z \)(although it’s value will be equal to that of \( z \)) and \( z \) will be printed as 2 as opposed to 3 and value of \( x \) will become 2 as opposed to 1.

### 4.1.2 Serialization and data types

Based on scheme described above, variables of basic types such as `int`, `char`, `float` etc. can be easily serialized into byte array due to their known sizes. However, more complex data types such as structures and pointer variables cannot be serialized as there is no straight forward way to determine their sizes. Although structure containing only basic data types can be serialized using our basic scheme, in practice structures contain various pointer variables. In other words, pointer variables and variables
of type structures with pointer variables as their members require special treatment for serialization and will involve more steps to serialize and de-serialize than mentioned above.

### 4.1.3 Pointers

During domain-transfer from public zone to private zone, any non-pointer variables are restored (de-serialized) in private zone by their values. Although their address information is maintained to preserve its state in public zone by any change in reference in private zone, they are not required to be allocated memory in private zone. This is where a pointer variable is different and is treated specially as it is required to be allocated memory. Subsequently, even though a pointer variable that is used in private zone might be allocated memory in public zone, it is conservative to assume that such instructions will not be tainted during analysis and hence will not be moved to the private zone. Thus, de-serializing pointer variable in private zone requires to allocate memory for it and size of information needs to be communicated during serialization. Therefore, serializing a pointer variable involves copying its size followed by its address and its value into byte array while de-serializing involves reading size followed by reading address and allocating memory of size number of bytes and reading value into the variable.

Figure 8 shows the code serialize and Figure 9 shows de-serialize c of type char *.

Note that, this procedure is applicable only to variable with single pointers and only while domain transfer from public zone to private zone and not the vice-versa. Hence, during domain transfer from private zone to public zone the marked ** step in de-serialization code is not performed.
```c
#define NUL 0x01

u_int8_t buffer[4096], *p;
char *c;
int l;
c = (char *)malloc(sizeof('password') + 1);
strcpy(c, "password");

memset(buffer, 0, 4096);
p = &buffer[0];
if(c == NULL) {
    l = NUL;
    memcpy(p, &l, sizeof(int));
p += l;
}
else {
    l = strlen(c);
    memcpy(p, &l, sizeof(int));
p += l;
    addr = (int)c;
    add_to_address_table(addr);
    memcpy(p, &addr, sizeof(int));
p += sizeof(int);
    memcpy(p, (void *)c, l);
p += l;
}
```

Figure 8: Code to serialize char *c
u_int8_t buffer[4096], *p;
char *c;
int l, addr;

memset(buffer, 0, 4096);
read_from_other_process(buffer);
p= &buffer[0];

memcpy(&l, (void*)p, sizeof(int));
p += sizeof(int);
if (l == NULL) {
    c = NULL;
} else {
    memcpy(&addr, (void*)p, sizeof(int));
    add_to_hash(c, addr);
    p += sizeof(int);
    c = (char*)malloc(l+1); **
    memcpy(c, (void*)p, l);
    c[l+1] = 0;
    p += l;
}

Figure 9: Code to de-serialize char *c
4.1.4 Arrays

Arrays of basic data types are serialized and de-serialized in same way as pointers; that is, size information is shared during serialization. However, arrays are not allocated in either private zone or public zone during de-serialization.

4.1.5 Structures

In simplest form, a structure variable can be conceived as set of variables of various data types and hence can be serialized and de-serialized by serializing and de-serializing each member individually. Note that, an optimization can be achieved on serialization of structures containing variables of only basic data types by directly copying it into byte array without copying each member individually(size of such structure can be obtained by sizeof operator) and likewise for de-serialization.

In practice, various complex data types such as linked-lists, circular linked-lists, recursive linked-list etc. are used. In C language, these data structures are basically made up of structures having one or more pointer variables of self type, and definition of the data type depends upon their semantic association. To describe structures containing variables of self type, we have developed a messaging protocol. We will describe this protocol after brief analysis.

4.1.6 Analysis of recursive structures

Intuitively, recursive structures can be flattened into byte array in a recursive fashion in which member variables of each node are serialized depending on their type. For example, consider a structure struct list shown in Figure 10.

For sake of simplicity, assume that the program makes use of a variable of type struct list to maintain a singly linked-list of 3 nodes. A recursive module would recursively copy count of all 3 nodes(using
```c
struct list {
    int count;
    struct list *next;
};
```

Figure 10: Recursive structure: struct list

```c
struct list1 {
    int count;
    struct list *next;
    char name[10];
};
```

Figure 11: Recursive structure: struct list1

The serializing technique discussed above to serialize an `int` and stops when it encounters a NULL pointer. To do this, the module will require to allocate a new node in private zone each time it encounters `next` and copy `count` to `count` of new node.

Note that, since serialization is done in an order of member variable’s listing in the structure, such recursive serialization is done in a “dept-first” manner. For example, in singly linked list with 3 nodes of type `struct list1`(as shown in Figure 11).

The module will serialize `count` of first node followed by `count` of second and third node. Once list is exhausted, it will serialize name of third node followed by `name` second and first node. Instance of byte array, in this case, can be depicted as shown in Figure 12.
Here numerical are designating each node in the list and node* are designating address of that node in the public zone. count* and name* denotes serialization of that variable including its address and size information.

Now consider that a program makes use of a variable of type struct list to maintain a circular linked-list of 3 nodes. In this case, the recursive module will never return as the list will never exhaust due to absence of NULL reference. In this case, a condition can be introduced in which address of each node that is serialized is checked if it was serialized earlier by looking into address-table and if a repeating address is encountered then module can return. This is a sufficient condition to serialize a linked list having linked list of a different type as their member variable.

4.1.7 The Messaging Protocol

Based on analysis above, we can derive 3 observations about data types based on recursive structures:

Any node of such data type would point to either

1. a NULL node or
2. a new node of same type that has not been traversed or
3. a node of same type that has been already traversed
To serialize a variable of data type based on recursive structure, it is required to include above information while copying each node into byte array. Based on this information, the de-serializing module can reconstruct same variable in private zone. This information is accommodated by adding an integer specifying type of next node in the list. For example, for NULL node an integer, say 0x01, is interpreted as NULL and associated action in de-serializing module would be `next = NULL;`; for new node an associated action would be to allocate memory for new node and as described earlier a repeating node generally means returning from the module. Such messaging is also essential to communicate values which cannot be referred to, such as null pointers.

Below are instances of byte array containing such messaging for a singly linked-list and a circular linked-list of type `struct list` with 2 nodes:
Here, node* denotes address of that node and count* denotes address and size information required of that variable; PTR, NUL and PEX are macro defined to some integer value.

```c
#define NUL 0x01 // a null reference
#define PTR 0x02 // a new node
#define PEX 0x03 // an already existing node
```

Here is the code to serialize and de-serialize a list of type struct list with 3 nodes:
u_int8_t buffer[4096], *p;
long int addr, msg;

void rec_ser_struct_list(struct list *node) {
    if (node == NULL) {
        msg = NULL;
        memcpy(p, &msg, sizeof(int));
        p += sizeof(int);
        return;
    }
    addr = *((int *)&node);
    if (find_address_table(addr)) {
        msg = PEX;
        memcpy(p, &msg, sizeof(int));
        p += sizeof(int);
        memcpy(p, &addr, sizeof(int));
        p += sizeof(int);
        return;
    }
    add_to_address_table(addr);
    msg = PTR;
    memcpy(p, &msg, sizeof(int));
    p += sizeof(int);
    memcpy(p, &addr, sizeof(int));
    p += sizeof(int);
    memcpy(p, &node->count, sizeof(node->count));
    p += sizeof(node->count);
    rec_ser_struct_list(node->next);
    return;
}

int ser_struct_list(struct list *first) {
    memset(buffer, 0, 4096);
    p = &buffer[0];

    if (first == NULL) {
        msg = NULL;
        memcpy(p, &msg, sizeof(int));
        p += sizeof(int);
        return (p - buffer);
    }
    addr = (int)(first->count); // this address is same as address of first node
    add_to_address_table(addr);
    memcpy(p, &addr, sizeof(int));
    p += sizeof(int);
    memcpy(p, &first->count, sizeof(first->count));
    rec_ser_struct_list(first->next);
    return (p - buffer);
}

Figure 15: Code to serialize (struct list *)
4.1.8 The hash-table

As described earlier, during de-serialization process in private zone, a hash-table is maintained to map address of a variable in public zone to that in private zone. Public zone, during serialization, also maintains table of address of each variable that is sent to the private zone. When a non-sensitive variable is sent back from private zone to public zone, private zone consults hash table and uses variable’s address in public zone during serialization process. Hence, in public zone, any reference modification by private zone is maintained after domain transfer. For example, addresses of two variables `char c` and `int i` are maintained in public zone and private zone as shown in Table I during any domain transfer:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Address in public zone</th>
<th>Address in private zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0xF1</td>
<td>0xB1</td>
</tr>
<tr>
<td>i</td>
<td>0xF2</td>
<td>0xB2</td>
</tr>
</tbody>
</table>

Address-table maintained by public zone
Hash-table maintained by private zone

**TABLE I: TABLES MAINTAINED BY PUBLIC ZONE AND PRIVATE ZONE**

While domain transfer from private zone to public zone, private zone consults its hash-table for address information of `c`, embodies 0xF1 (and not 0xB1) and its value in private zone.
4.1.9 Code generation

Generating code for all the steps required to serialize/de-serialize a variable is somewhat cumbersome. Also, any change in design decision would result in significant changes in the generating code. Hence, such code for basic data types can be manually written and can be linked to the public zone and private zone. In our design, we have employed C macros for code to serialize/de-serialize basic data types. These macros are used, in turn, to define macros of more complex data types such as pointers. Although such macros for a variable of structure type cannot be manually written as it depends on definition of structure and hence code for such variable is required to be generated, using macros for various conditions and basic data-types significantly reduces the size of the code that is required to be generated for structures. We generate code for structure variables and link them with public zone and private zone. Code for rest of the variables are retrofitted in the program in form of macros. Although, it doesn’t require to normalize these macros to be able to compile both the programs, CIL has facility to do so. One advantage of our approach is that any change in design can be easily accommodated by simply changing macro definition thus providing the flexibility in the design.

4.2 Limitations

While we believe our approach is general enough to be applicable to a large class of programs, our implementation has two main limitations. Both of them relate to how pointers and aliasing are used in the target program. The first limitation concerns the use of aliasing. Currently the analysis engine treats pointers as data references, treating any operations that directly make use of pointers as references to values. They will be correctly marked as sensitive. However, the presence of aliasing (where an object is referred to both by its original name and its alias) may invalidate this assumption. Whenever the
analysis encounters the possibility of aliasing, it prompts the user for manual verification. In this case, we had to perform manual verification of the code generated for the use of aliasing. Our assumption is valid for the programs we have tested our tool with. In these examples, pointers were indeed used as references and hence our code generation approach was correct. Integration of points-to-alias analysis algorithms in our approach will ensure that we correctly identify all sensitive instructions and avoid such manual verification.

The second assumption concerns the size of objects referred by pointers. While serializing data to communicate the public zone state to the private zone, there are some situations where the sizes of objects pointed to by pointer data types is not known. In general, tracking pointer sizes is hard. One solution is to have fat pointers that record additional information, such as their sizes. Another viable solution is to have a shared memory region between the public zone and private zone that contains public data, and taking additional measures to protect sensitive data in the private zone. This will be explored as part of our future work. For now, the code generation uses some simple heuristics that make certain guesses about the nature of the variable being pointed to. For example, to serialize a variable of type pointer to char, it relies on strlen() function for the size of the variable. Another current option we had at hand was to have the user of the tool explicitly specify the size of the pointed-to variables. We preferred the first option even though it is a potentially unsound heuristic, as it seems to be a good tradeoff to the complexity of specifying individual object sizes. We adopted this approach, and followed up by manual verification for verifying pointer sizes. Since the sizes of code blocks was small (see Figure 18), this was easily possible.
```c
u_int8_t buffer[4096], *p;
long int addr, msg;
struct list * rec_deser_struct_list() {
    int t_addr;
    struct list *node;
    memcpy(&msg, (void *)p, sizeof(int));
    p += sizeof(int);
    if(msg == NULL) {
        node = NULL;
        return node;
    }
    if(msg == PEX) {
        memcpy(&addr, (void *)p, sizeof(int));
        p += sizeof(int);
        t_addr = find_hash_table(addr);
        if(t_addr == -1) {
            printf("ERROR: missing address\n");
            exit(0);
        }
        node = (struct list *)t_addr;
        return node;
    }
    memcpy(&addr, (void *)p, sizeof(int));
    p += sizeof(int);
    node = (struct list *)malloc(sizeof(struct list));
    t_addr = *((int *)node);
    add_to_hash(t_addr, addr);
    memcpy(&(node->count), (void *)p, sizeof(node->count));
    p += sizeof(node->count);
    node->next = rec_deser_struct_list();
    return node;
}
int deser_struct_list(struct list *l) {
    memset(buffer, 0, 4096);
    read_from_other_process(buffer);
    p = &buffer[0];
    memcpy(&msg, (void *)p, sizeof(int));
    p += sizeof(int);
    if(msg == NULL) {
        l = NULL;
        return (p - buffer);
    }
    memcpy(&addr, (void *), sizeof(int));
    p += sizeof(int);
    add_to_hash((int *)&l->count, addr);
    memcpy(&(l->count), (void *)p, sizeof(l->count));
    p += sizeof(l->count);
    l->next = rec_deser_struct_list();
    return (p - buffer);
}
```

Figure 16: Code to de-serialize (struct list * )
CHAPTER 5

EVALUATION

In this section, we describe the results with our prototype implementation. Our implementation uses the CIL (16) framework. We have chosen representative examples from several classes of open source programs: simple authentication programs, system monitoring and analysis programs, music related utilities that handle sensitive information. Our evaluation section is divided into three parts 1) policy enforcement evaluation, where we present the effectiveness of policies in preventing information flow 2) performance evaluation and 3) security analysis of the approach.

5.1 Policy enforcement evaluation

5.1.1 Linux-Monitor

Linux-Monitor is a utility that polls system resources such as processes and disk partitions at specific intervals of time. It then proceeds to log this information into system logs or at a remote-server running on port 8881 (syslogd substitute for remote logging). Here we enforce a policy that prevents linux-monitor from logging to a remote connection when it reads from system resources considered sensitive by the system administrator. The policy enforcement successfully prevented this information from being communicated.

5.1.2 Htpasswd

Htpasswd is used to create and update flat-files, that store user names and password for basic authentication of HTTP users. It calls the crypt routine and then writes encrypted passwords to a user
specified file. Our tool separates the code handling password information and enforces a policy that prevents this information from being written to any other file.

5.1.3 **Mediachat**

*Mediachat* is our own implementation that simulates a media player that allows users to connect to a chat server while reading and playing music files from their local disk. It is an example of an application that operates on sensitive (music preferences) and non-sensitive data (network chat) at the same time. The policy is to prevent the list of music files from being communicated on the network to the server. A typical sandboxing policy would have prevented the chat application from correctly functioning. Our data sandboxing approach allowed the chat to proceed while preventing any private music information from being sent to the network.

5.1.4 **chfn, chsh, passwd**

These are standard Unix administrative utilities. Our tools successfully partitioned the authentication code in these programs. We sandboxed the private zone programs to ensure that sensitive information is only written to the system password files (in the case of passwd, chfn and chsh). The sandbox for the public zone programs need a setuid wrapper, as these are setuid programs and the ptrace mechanism we use for user-level system call interposition does not cross setuid boundaries. A kernel interception mechanism can handle this situation.

5.2 **Evaluation**

The performance of the system was analyzed both for the overall performance and for the domain transfer operation using micro benchmarks. We ran experiments on a Intel P4 3.4 GHz processor with 2 GB of RAM running linux kernel version 2.6.9.
5.2.1 **Micro benchmark on the domain transfer operation**

We measured the base overhead for a domain transfer between public zone and private zone. This overhead results from setting up a communication channel and performing serialization. Of the programs we tested, the linux-mon program had the worst case domain transfer overhead. We monitored five different resources and the time to serialize shared data was 2.86 milliseconds. The performance penalty for domain transfer operation (cost of communication setup + serialization and deserialization as opposed to simply executing instructions in the original program) was a factor of 6.16.

5.3 **Overall performance measurements**

We measured the overall performance (public zone + private zone, combined user + system time, without system call interposition) of the transformed programs compared to the original programs, and the results are shown in Figure 17. Linux monitor is a continuously running server program that keeps reading sensitive files and so we could not use this form of comparison. From the results, we note
that the worst overhead is about 231.5\% for htpasswd, and is primarily due to the new address space
creation. System call interposition techniques at the kernel level would add about 10-15\% overheads.

5.3.1 Performance Improvements

Our implementation is currently not optimized for performance. The major factors contributing
to the overheads are the domain transfer operation and the associated copying of state. Several op-
timizations are possible. The current implementation uses address spaces for memory isolation. An
intra-address space protection mechanism such as software fault isolation (17) can result in much lower
overheads. Also compiler optimizations such as code motion can be used to produce fewer state trans-
fers, and live variable analysis may result in lowering copying overheads.

5.3.2 Additional memory overheads

Since our approach creates a new private zone process for every program, there is an additional
memory overhead due to the process creation. We chose to measure memory overhead for to the
htpasswd program, as the private zone process in this case has the worst case memory consumption
both for code as well as data. We inserted a break point in the code of the private zone, and measured
the resident memory size of the process. The additional memory required was 2.5 Megabytes. Other
programs listed above tested will consume significantly less memory than htpasswd.

5.3.3 Program sizes

Figure 18 gives a table of the analyzed program sizes in LOC, the number of marked (sensitive)
instructions in LOC, the number of blocks and the average block sizes respectively. The actual sizes of
the programs are much larger, the LOC was measured without counting the code from libraries. Also
the sizes of the private zone programs are the sizes of these marked instructions plus a linked library of
Figure 18: Program size information

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Orig. (# LOC)</th>
<th>Marked ins (# LOC)</th>
<th># of Blocks</th>
<th>Avg. # LOC/block</th>
</tr>
</thead>
<tbody>
<tr>
<td>passwd</td>
<td>2333</td>
<td>193 (8%)</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>htpasswd</td>
<td>984</td>
<td>240 (24%)</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>chfn</td>
<td>1238</td>
<td>126 (10%)</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>chsh</td>
<td>1138</td>
<td>112 (10%)</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Mediachat</td>
<td>335</td>
<td>92 (27%)</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

around 200 lines of C code that performs serialization. We observe that the percentage of instructions that handle sensitive data in these programs ranges from 8% to 27%. The highest is for Mediachat, which performs a significant fraction of its code in handling sensitive (music) data.

5.4 Security analysis of the approach

Our approach also prevents damage from attacks that subvert the program to gain access to confidential data. In this case, the (minimal) code that runs in the private zone only reads and processes sensitive information. The public zone if subverted, thus cannot be directed to read sensitive information. Secondly, the code that runs in the private zone does not share sensitive data with the public zone. Also, the communication channel between the public zone and private zone is a pipe, which is only available to these two processes and hence cannot be observed by any local processes acting on behalf of the attacker. It is possible that the private zone can have code (derived from the original program) to have bugs that can be exploited. In this case, we first note that our approach does not do any worse than before, as the original program still can be exploited. Secondly, since the private zone runs the minimal piece of code required to run sensitive information, as opposed to the public zone which runs
code that does the bulk of the work of the program, chances of such bugs being present in the private zone is reduced. Thus the code that reads confidential data is separate from the remaining processing code, and this mitigates security risks of losing confidential information.
CHAPTER 6

RELATED WORK

6.1 Sandboxing

Sandboxing based approaches (1; 2; 3; 4; 5) involve observing a program’s behavior and blocking actions that may compromise the system’s security. This is achieved in most of these works through system call interposition. This approach is transparent to the program, and policy enforcement comes without needing to modify programs. However, the monitor cannot distinguish between sensitive and public data that is written in an output channel. Consequently, sandboxing approaches do not allow program actions that write to public output channels, and could therefore be limiting application behavior in many situations. In this work, we have provided an approach that addresses this problem.

6.2 Static analysis

Several static analysis based approaches (18; 19; 20; 21) have been successfully used for finding bugs in C programs. However, static analysis alone cannot be used reliably in policy enforcement for C programs, as programs can be compromised due to lack of memory safety. Though our approach uses static analysis, this is merely done to partition a program. Our enforcement technique is based on monitoring at the application/kernel boundary, and hence is not bypassable.

6.3 Information flow analysis

There is a long history of work in information flow analysis of program. Work on information flow started with the work of Bell and LaPadula (8) in the context of processes in an operating system. More
recent work (22; 23) brought this work in terms of data flow in programs. Most of these approaches rely on type safety for enforcement of policies. In a language such as C which is not type safe, direct application of such policies will not result in reliable enforcement. In such systems, confidential data continues to remain in the program’s memory once a program is victimized, and is readily available without any protection. The goal of this work was to minimize the window of disclosure of confidential information. Though we use static analysis for inferring possible data flows, our enforcement uses the combination of address space separation of private data and sandboxing for reliable enforcement of such policies.

6.4 Taint based approaches

Taint based approaches (24; 25) have been recently used in detection of data-oriented attacks such as injection attacks. Taint-based approaches do have the scope for providing finer granularity to assist sandboxing. In all the above works, taint based enforcement has been used for protecting system integrity, i.e., ensuring that data from untrusted sources do not compromise trusted destinations such as shell commands. In this work, we presented an approach to solve the dual problem of confidentiality, where input from trusted sources do not reach untrusted (public) output channels.

6.5 Program partitioning approaches

A program transformation technique Jif-split was described in (26) in the context of distributed programs running on untrusted hosts. Programs written in a language called Jif are annotated with security types, and the system automatically splits the program to match the enforced security policy described through these types. There are several differences with respect to our approach: First, their approach is in enforcement of distributed programs such as web services in the context of untrusted
hosts. Our approach is for protecting the confidentiality of inputs in trusted hosts that may receive untrusted inputs. Secondly, their approach requires writing programs in a special type-safe language for Jif to accomplish the partitioning, and our approach is focused on retrofitting C programs that handle confidential information for protection of sensitive data.

Privilege separation is another idea that uses separation of program on the basis of privileges required to minimize security risks. It was used in the design of programs such as qmail (11), and later in retrofitting of programs such as OpenSSH manually (12) and automatically (13). While the goal of these approaches is to separate code running with special privileges, our approach is focused on using using a similar approach to make private data inaccessible to the parts of the program that do not require it. Privtrans (13) was the first work that applied an automated technique for privilege separation in the realm of C programs. We follow a similar partitioning approach that partitions programs based on process-level separation. However, there are two main differences. Their approach to partitioning unprivileged and privileged code is based on a delegation / authorization model, where the unprivileged code requests the privileged code to perform certain operations. This is suitable for privileged operations such as setuid, where explicit operating system permissions are required to perform these operations. However, it may not be suitable for operations that handle private data, as the unprivileged program can be compromised to perform such an operation (say reading a sensitive file) by itself without needing special permissions or having to request the privileged program to perform that operation. (For instance, this can be done after a compromise through a return-into-libc attack.) In our approach, such an operation is prevented through the policy enforcement by system call interposition. Secondly, they assume the procedure level abstraction as the boundary for partitioning. This method of partitioning is unsuitable for our approach
that intends to minimize the instructions that can access sensitive data, and thereby making it inaccessible to portions of the program that do not require it. Doing this requires us to insert domain transfer code in the body of functions and not necessarily at function boundaries.
CHAPTER 7

FUTURE WORK

Although our implementation have partitioned several programs, in order for it to be able to handle more general class of programs - specially legacy applications - some work is required to be done:

7.1 Function-Pointers

In our implementation, we can not communicate function pointers during domain transfer. One solution to this problem could be to identify the instructions when function pointers are assigned values and move them to private zone. Although this solution can be applicable to very basic programs, it will not work if such instructions are encompassed in control loops and/or statements.

Other solution could be to maintain two separate tables that map function name to its address in both the programs. Function pointers can be communicated by communicating their name and assigning associated address (in that program) to it. For example, the tables for two function definitions (void foo()) and (void foobar()) can be implemented as below:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>“foo”</td>
<td>0xE1</td>
</tr>
<tr>
<td>“foobar”</td>
<td>0xE2</td>
</tr>
</tbody>
</table>

Function-address-table maintained by public zone

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>“foo”</td>
<td>0xC1</td>
</tr>
<tr>
<td>“foobar”</td>
<td>0xC2</td>
</tr>
</tbody>
</table>

Function-address-table maintained by private zone

TABLE II: TABLES MAINTAINED BY PUBLIC ZONE AND PRIVATE ZONE FOR FUNCTION POINTERS
void (*funcPtr)();
if (expression) {
    funcPtr = &foo;
}
else {
    funcPtr = &foobar;
}

Figure 19: Communicating function pointers

Now consider scenario in Figure 19. While communicating funcPtr from public zone to private zone (or vice-versa), public zone looks into its function-address-table to fetch string associated with value of funcPtr and send the string, say “foo”, to the private zone. On receiving “foo”, private zone looks into its function-address-table for “foo” and assigns 0xC1 to funcPtr. These tables can be initialized with function addresses in the beginning of each programs and made available to all the functions in that program. Although this is a viable solution, there are some security implication having global function declarations.

7.2 Optimization over size of private zone

During partitioning, multiple function definitions are moved to private zone as transitive closures. Even though not all the instructions in these functions are sensitive, no domain-transfer is occurred from private zone to public zone during execution of non-sensitive instructions in these functions. This is because, in current design only public zone can request domain transfer to private zone and not vice-versa as the partitioning engine does not take into account any functions other than context-functions. Improvement in this design such that domain transfer can occur in all the functions that are moved to
private zone can greatly reduce the size of private zone. Although such domain transfer can incur an additional overhead of serialization, an optimization can be achieved by code motion.

### 7.3 Better approximation for variable sizes

Currently we are relying on `strlen()` function for size of a `char *`). Although in all the examples that we tried, this tend to work, it has some security implications. For example consider the code show in Figure 20.

If line 4 is tainted as sensitive(and hence moved to private zone) then during domain transfer, `c` in private zone would be allocated only 6 bytes (`strlen(``hello'')+1`) as part of restoring its value(referring to serialization process as explained above). Concatenating “ world” would overflow the heap which is not a desired situation.

In general, this is a problem for only those character pointer references that are memory allocated. Pointers without memory allocation generally used as reference pointer(that is, they are not used in operations such as `strcat()`) and are not a problem. One probable solution to this problem is to keep track of information - if a variable of `char *`) is allocated memory. Such information can be an approximation during analysis phase and, in general, hard to achieve for C programs.
7.4 Maintaining reference change to array index

Current implementation of serialization cannot keep track of reference to a specific index of array. For example, in Figure 21 since only address of \( a[0] \) is maintained in hash-table in private zone, reference to \( i \) cannot be maintained during domain-transfer. This problem can be easily fixed by adding address of all the array indices in the hash-table in private zone and in address-table in public zone.
CHAPTER 8

CONCLUSION

In this thesis, we presented an approach called data sandboxing for protecting confidential information from unauthorized disclosure. Our approach works by partitioning program instructions into two parts, the public zone and private zone. These two parts are isolated from each other through address space separation, thus making the sensitive data in private zone inaccessible to the public zone part of the program. Our approach has potential to prevent sensitive information from disclosure in the event of a compromise. It also overcomes the disadvantages of pure sandboxing approaches which prevent external communication in order to enforce confidentiality. We illustrated the use of our approach by applying it to several examples.
CITED LITERATURE


15. Swaminathan, R.: Static information flow analysis tool for identifying sensitive program regions.


## VITA

<table>
<thead>
<tr>
<th>NAME</th>
<th>Tejas Khatiwala</th>
</tr>
</thead>
<tbody>
<tr>
<td>HONORS</td>
<td>“Graduate Student Leadership Award”, Department of Computer Science, University of Illinois at Chicago, Chicago, 2006-2007.</td>
</tr>
</tbody>
</table>