PtrSplit: Supporting General Pointers in Automatic Program Partitioning

Shen Liu
The Pennsylvania State University
University Park, PA
sxl463@cse.psu.edu

Gang Tan
The Pennsylvania State University
University Park, PA
gtan@cse.psu.edu

Trent Jaeger
The Pennsylvania State University
University Park, PA
tjaeger@cse.psu.edu

ABSTRACT

Partitioning a security-sensitive application into least-privileged components and putting each into a separate protection domain have long been a goal of security practitioners and researchers. However, a stumbling block to automatically partitioning C/C++ applications is the presence of pointers in these applications. Pointers make calculating data dependence, a key step in program partitioning, difficult and hard to scale; furthermore, C/C++ pointers do not carry bounds information, making it impossible to automatically marshall and unmarshall pointer data when they are sent across the boundary of partitions. In this paper, we propose a set of techniques for supporting general pointers in automatic program partitioning. Our system, called PtrSplit, constructs a Program Dependence Graph (PDG) for tracking data and control dependencies in the input program and employs a parameter-tree approach for representing data of pointer types; this approach is modular and avoids global pointer analysis. Furthermore, it performs selective pointer bounds tracking to enable automatic marshalling/unmarshalling of pointer data, even when there is circularity and arbitrary aliasing. As a result, PtrSplit can automatically generate executable partitions for C applications that contain arbitrary pointers.

KEYWORDS

Automatic program partitioning; bounds tracking; data marshalling

1 INTRODUCTION

Following the principle of least privilege, privilege separation in software refers to separating a software application into multiple partitions, each with its own set of privileges. Partitions are isolated so that the compromise of one partition does not directly lead to the compromise of other partitions. Function calls between partitions are realized by Remote-Procedure Calls (RPCs); data for an RPC are marshalled and sent to the callee, which unmarshalls the data, performs its computation, and sends the result back to the caller.

Privilege separating programs in low-level, type-unsafe languages such as C/C++ is especially beneficial to security because these programs are prone to attacks (e.g., attacks enabled by memory vulnerabilities). For instance, OpenSSH was refactored by Provos *et al.* to have unprivileged monitor processes for handling user connections and one privileged server process [28]. Another example is the microkernel operating-system design, in which a minimum amount of code is kept in the kernel and most OS functionalities are pushed outside. Yet another example is Google's Chromium browser, which isolates each tab into a sandboxed process [1, 29].

These manual restructuring efforts have significantly improved the security of the relevant software; however, they are labor intensive and sometimes error-prone.

Several systems [3, 5, 21, 32] have been proposed to apply program analysis to separate C/C++ applications automatically into partitions, from a small number of user annotations about sensitive data. These systems demonstrate automatic program partitioning can be practical. However, one major limitation of these systems is that they lack good support for pointer data, which are prevalent in C/C++ applications. In particular,

- C-style pointers do not carry bounds information; when a pointer needs to be sent across the partition boundary in an RPC call, marshalling does not know the size of the underlying buffer and consequently cannot marshall the buffer automatically. Some systems adopt heuristics when marshalling pointer data (e.g., a "`char *`" pointer is assumed to point to a null-terminated string); however, programmers are often required to write marshalling and unmarshalling code manually for pointer data, especially for pointers that point to dynamically allocated buffers. Some systems avoid the problem by restricting the partitioning algorithm to not create partitions that require pointer passing; this design, however, limits the flexibility of where partitions can be created.
- A program-partitioning algorithm needs to reason about dependence in a program to decide where to split. When the program has pointers, a global pointer analysis is typically required to understand aliasing and how data flow in memory. However, global pointer analysis is often complex and does not scale to large programs.

In this paper, we propose a series of techniques that enable the support of pointers in automatic program partitioning. These techniques are implemented and evaluated in a system called PtrSplit. Its major techniques and contributions are as follows:

- Taking source code as input, PtrSplit constructs a static Program-Dependence Graph (PDG) for the program. A feature in PtrSplit's PDG that distinguishes it from previous

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134066

PDGs for imperative programs is a technique called *parameter trees*. It provides a modular way of constructing the PDG for a program with pointers; as a result, only an intraprocedural pointer analysis is needed, instead of a global pointer analysis. Our tree representation generalizes the object-tree approach in prior work [19], which discussed a tree representation for objects in object-oriented languages and did not cover pointers at the language level; our system uses the tree representation for representing pointers in imperative languages and deals with circular data structures resulting from pointers.
Based on the PDG, PtrSplit performs a standard reachability-based program-partitioning algorithm that separates the program into a partition that accesses sensitive data and a partition for the rest of the code.

- To marshall pointers, PtrSplit instruments the program so that pointers carry bounds information. However, prior work shows that full pointer bounds tracking incurs significant performance overhead. PtrSplit makes the critical observation that program partitioning does not need full pointer tracking—it is sufficient to track the bounds of pointers that cross the partitioning boundary. Therefore, given an arbitrary partitioning of the program, PtrSplit computes a set of pointers that require bounds information and instruments the program to track the bounds of only those pointers. We call this *selective pointer bounds tracking*.
- PtrSplit generates code that performs marshalling and unmarshalling for data sent over an RPC call. This is automatic even for pointer data because all pointers that cross the partition boundary carry bounds information. We describe a type-based algorithm for performing deep copies of pointer data, which can cope with the situation of circular data structures and arbitrary aliasing, without user involvement. For instance, PtrSplit allows one partition to send a circular linked list to a second partition.

The prototype of PtrSplit is implemented inside LLVM; our preliminary evaluation on security-sensitive benchmarks and compute-intensive benchmarks suggests the system is already practical for C applications with pointers and can produce executable partitions with a modest amount of performance overhead.

2 RELATED WORK

Several tools have been proposed to assist programmers in program partitioning. Privman [16] is a library for helping programmers manually partition their applications to control access to privileged system calls. Wedge [3] provides a dynamic profiling tool for partitioning assistance. It collects statistics about how a program uses memory to help programmers draw partition boundaries; however, programmers still need to perform manual code changes and partitioning. Trellis [22] infers access policies on code and data in multi-user applications from user annotations and enforces the policies through a modified OS.

Automatic program partitioning employs program analysis and separates a program into multiple partitions, with minimum user involvement. Privtrans [5] performs static analysis to automatically partition a C application into a privileged master process with sensitive information and an unprivileged slave process. ProgramCutter [32] collects a dynamic dependence graph via profiling and performs graph partitioning to produce partitions that balance performance and security using a multi-terminal minimal cut algorithm. SeCage [21] employs hybrid static/dynamic analysis to compute a set of functions that can access secrets and isolates the sensitive partition via hardware virtualization support. Jif/split [33, 35] automatically partitions a Java source program based on security-label annotations and a description of trust relationships between protection domains. Swift [7] generalizes Jif/split for the development of web applications by exploring general algorithms for improving both security and performance. With the emergence of Trusted Execution Environments (TEEs), there have also been program-partitioning frameworks that target Intel's SGX or ARM's TrustZone. For instance, Rubinov *et al.* [30] proposed a static-analysis framework that partitions an Android application into one component that runs in TrustZone's secure world and one that runs in TrustZone's normal world. A similar system called Glamdring [20] targets Intel's SGX.

All the aforementioned automatic program partitioning frameworks, either partition programs in languages that do not have explicit pointers (e.g., Java) or require programmers to manually write marshalling and unmarshalling code for pointer data [5, 32]; furthermore, data dependence computed by these frameworks that partition C/C++ application are incomplete in the presence of pointers and can lead to incorrect partitioning results. In contrast, PtrSplit uses a PDG representation that soundly represents pointer data as parameter trees, and tracks pointer bounds for automatic marshalling and unmarshalling without user involvement.

PtrSplit partitions programs so that sensitive data within the sensitive partition cannot be directly or indirectly accessed by the insensitive partition. It is a form of controlling the flow of sensitive information. Information flow can be controlled in other mechanisms, through dynamic information-flow flow tracking as in systems such as Asbestos [10], HiStar [34], and Flume [17], or a capability model as in Capsicum [31], or via a static language mechanism such as Jif [23].

3 SYSTEM OVERVIEW

Fig. 1 presents PtrSplit's workflow. It takes the source code of a single threaded C application as input; the code has been annotated by the programmer with information about sensitive and declassified data. Sensitive data can be either confidential data (e.g., keys) or data from an untrusted source (i.e., tainted data such as user input).

The source code is converted to an LLVM IR program by LLVM's front end. PtrSplit then constructs the PDG for the IR program. A PDG-based algorithm then computes two *raw partitions*: one sensitive raw partition that can access sensitive data and one insensitive raw partition with the rest of the code. However, raw partitions cannot run directly because after partitioning some function calls become Remote-Procedure Calls (RPCs) and it is necessary to add RPC wrapper code for data marshalling and unmarshalling. In PtrSplit, each partition is loaded into a separate process, so RPC wrapper code must be added for inter-process communication.

Based on raw partitions, PtrSplit performs selective pointer bounds tracking, which tracks bounds information for pointers

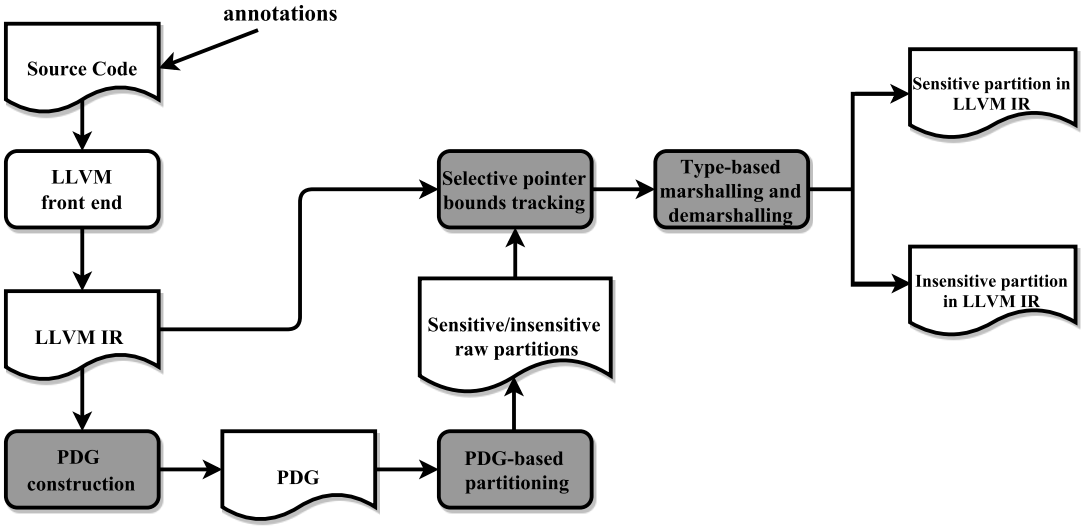

Figure 1: The workflow of our automatic program-partitioning framework (gray components belong to PtrSplit).

whose values can potentially cross the partitioning boundary. Bounds information for pointers is then used by a type-based method, which generates RPC wrappers that perform data marshalling and unmarshalling for inter-process RPC calls. In the end, PtrSplit generates one executable partition with all sensitive code, data, and RPC wrappers, and also an executable partition with insensitive code, data, and RPC wrappers.

A running example. We will illustrate the main points of PtrSplit by a toy example in Fig. 2. The example takes a username and a text input from the user, greets the user by the `greeter` function, initializes a key, and encrypts the text by xor-ing it with the key. The global `key` is the sensitive data that needs protection; therefore, it is marked sensitive using a C attribute. Note the program has a format-string vulnerability at line 6 in `greeter`, which could allow an attacker to take over the program and learn the key.

Intuitively, a partitioning framework should put the `greeter` function into the insensitive partition since no sensitive data can flow to it. Other functions, including `initkey`, `encrypt`, and `main` should be in the sensitive partition since `key` may be accessed by them directly or indirectly. This partitioning would isolate the format-string error in `greeter` into the insensitive partition and prevent the attacker from learning the key. Similar to other partitioning frameworks, PtrSplit also supports declassification. If `ciphertext` is annotated as declassified data, `main` can also stay in the insensitive partition even though it accesses `ciphertext`; in this way, vulnerabilities in `main` are isolated.

4 PDG AND PARTITIONING

Program partitioning requires analyzing dependence in an input program carefully and adjusting the program to a distributed programming style. A key step in PtrSplit is to construct for the program a graph representation of dependencies, called the Program Dependence Graph (PDG [11]); two follow-up steps in PtrSplit including program partitioning and selective pointer bounds tracking are performed on the PDG, as we will discuss.

Conceptually, a PDG represents a program's data and control dependence in a single graph and can facilitate static analysis including program slicing and automatic parallelization. There are many systems that construct PDGs for programs in different languages and with different precision. A distinguishing feature of our PDG construction is its approach of *parameter trees* for representing composite data (e.g., pointers) that are passed during function calls and returns. We will start explaining nodes and edges that are common in a PDG representation in Sec. 4.1, and discuss the parameter-tree approach in Sec. 4.2. In this discussion, we will use examples in C for readability, even though PtrSplit constructs PDGs for LLVM IR programs; the IR-level PDG construction will be explained in Sec. 4.3. Finally, we present a standard PDG-based partitioning algorithm in Sec. 4.4.

4.1 Regular Nodes and Edges in PDGs

Every instruction in the program is represented as an *instruction node* in a PDG. For edges, there are *data/control dependence edges* and *call edges*.

In general, an instruction node $n1$ is data dependent on instruction node $n2$ if $n1$ uses some data produced by $n2$. Our PDGs have two kinds of data-dependence edges:

(1) there is a *def-use dependence* if $n1$ uses a variable x that is defined in $n2$; an edge from $n2$ to $n1$ with label x is added.[1]
(2) there is a *RAW (Read-After-Write) dependence* if $n1$ reads memory that was written by $n2$ and an edge from $n2$ to $n1$ is added with label *id*, assuming *id* points to the memory in question.

An example of def-use dependence is as follows. Variable `x` is defined in "`x = 1`" and later used in the assignment to `y`.

```
x = 1;
... // x not modified
y = x + x;
```

[1]The edge direction reflects the dataflow direction, instead of the direction of dependence; this makes algorithms on PDGs easier to state.

```
1  char __attribute__((annotate("sensitive"))) *key
       ;
2  char *ciphertext;
3  unsigned int i;
4
5  void greeter (char *str) {
6    printf(str); printf(",␣welcome!\n"); }
7
8  void initkey (int sz) {
9    key = (char *) (malloc (sz));
10   // init the key randomly; code omitted
11   for (i=0; i<sz; i++) key[i]= ...;
12 }
13
14 void encrypt (char *plaintext, int sz) {
15   ciphertext = (char *) (malloc (sz));
16   for (i=0; i<sz; i++)
17     ciphertext[i]=plaintext[i] ^ key[i];
18 }
19
20 void main (){
21   char username[20], text[1024];
22
23   printf("Enter␣username:␣");
24   scanf("%19s",username);
25   greeter(username);
26   printf("Enter␣plaintext:␣");
27   scanf("%1023s",text);
28
29   initkey(strlen(text));
30   encrypt(text, strlen(text));
31   printf("Cipher␣text:␣");
32   for (i=0; i<strlen(text); i++)
33     printf("%x␣",ciphertext[i]);
34 }
```

Figure 2: A toy C program that encrypts a plaintext.

An example of RAW dependence is as follows. Memory location pointed to by p is written in instruction "*p = 1" and read in the assignment to y.

```
*p = 1;
... // memory pointed to by p not modified
y = *p;
```

For control dependence, an instruction node $n1$ is control dependent on $n2$ if, intuitively, there are two edges out of $n2$ and taking one edge results in the execution of $n1$, while taking the other edge results in the case of not executing $n1$. The formal definition of control dependence can be found in [11].

Call edges connect call sites with the entries of possible callee functions. For an indirect call (a call through a variable, e.g.), it may be connected with multiple possible callee functions. We adopt static, type-based matching [26] so that an indirect call via a function pointer can target any function whose type is compatible with the function pointer's type. For this method to be valid, some preprocessing of source code is required [26] (e.g., to eliminate type casts that involve function-pointer types by adding function wrappers).

4.2 Parameter Trees

The motivation for parameter trees is to simplify the computation of inter-procedural data dependence and obtain a modular PDG-construction approach. To illustrate this, let us revisit the example in Fig. 2. Notice on line 30, there is a function call to `encrypt` and `text` is passed; inside `main` the `text` buffer is written by `scanf` and inside `encrypt` the passed buffer is read. Therefore, there is a RAW dependence between the `scanf` call instruction in `main` and the instruction in `encrypt` that reads the buffer.

This kind of dependence is inter-procedural. The proper calculation of such dependence would require a whole program analysis such as a global pointer analysis, which needs access to all code, is complex, and often does not scale. Furthermore, the resulting PDG may suffer from edge blow ups: suppose the caller has n instructions that can write to a buffer and all n writes can affect the result of m reads in the callee, then the number of dependence edges is $O(n * m)$.

To obtain a modular and scalable PDG-construction system, we introduce parameter trees. In this approach, for each parameter of a function, we build a *formal parameter tree* according to the parameter's type. The parameter tree contains nodes that represent all the storage (memory) regions that the function can access through the parameter directly or indirectly.

We will present a formal algorithm for parameter-tree building in Sec. 4.3. An example is instead discussed in this subsection. The parameter tree for the `plaintext` parameter of `encrypt` in the running example can be found inside Fig. 3. It has a root node labeled "`plaintext:char*`" for representing the storage of the pointer, and a child node labeled "`*plaintext:char`" for the memory region that the pointer points to. The type in a parameter-tree node specifies the type of elements the corresponding memory region holds.

In addition to formal parameter trees, we also construct an *actual parameter tree* for each argument at a function call site, and connect nodes in an actual tree with corresponding nodes in a formal tree by data-dependence edges. Fig. 3 draws a PDG snippet for the running example that shows the interaction between `main` and `encrypt`. The call site in `main` has two arguments: `text` and `strlen(text)`; each is built with an actual parameter tree. The `encrypt` function has two parameters, each with a formal parameter tree.

Parameter trees enable modular construction of PDGs. To build a PDG for a large program, we can first build a PDG for each function using an intra-procedural analysis, and then "glue" the functions' PDGs together using parameter trees. A global analysis is avoided. Put it in another way, all data-dependence edges become local, either between two instruction nodes or between an instruction node and a parameter-tree data node. Inter-procedural data dependence is represented transitively via local data-dependence edges. Let us revisit the running example; recall that there is a RAW dependence between the `scanf` call in `main` and the instruction in `encrypt` that reads the `plaintext` buffer; this interprocedural dependence is broken into three edges in Fig. 3: one from the `scanf`

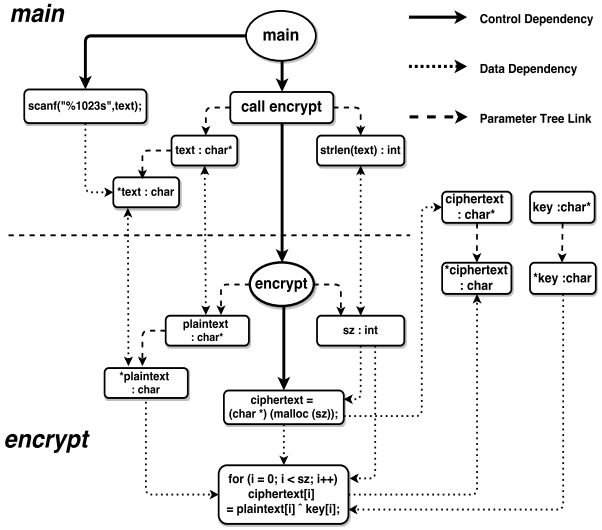

Figure 3: A PDG snippet for our running example. For clarity, the graph uses a single node for the entire `for` loop in `encrypt`; in contrast, PtrSplit's PDG construction breaks a loop into LLVM IR instructions and has one node for each IR instruction. The graph also omits labels on data-dependence edges.

node to parameter-tree node `*text`; one from `*text` to parameter-tree node `*plaintext`; one from `*plaintext` to the loop node that reads memory via `plaintext`.

Thanks to parameter trees, if the caller has n instructions that can write to a buffer and all n writes can affect the result of m reads in the callee, the number of edges becomes $O(n + m)$: we add $O(n)$ edges from the write instructions to the data nodes in the actual parameter trees, $O(m)$ edges from the data nodes in the formal parameter trees to the m read instruction, and a constant number of edges between actual and formal parameter trees.

We note that return values and global data are also represented as parameter trees. For instance, the `key` and `ciphertext` global data in the running example are represented using trees similar to the one for `plaintext`, as shown in Fig. 3. Also, data-dependence edges between instructions that perform global data access and corresponding tree nodes for global data are added.

4.3 LLVM PDG Construction

We next outline PtrSplit's algorithm for PDG construction in LLVM.

Parameter-tree building. PtrSplit's parameter tree building is type based. We next formalize the process. Fig. 4 presents the syntax of a subset of LLVM IR types. A type t can be an integer type int, a pointer type t_1*, an anonymous struct type that contains a list of types for the struct's fields, and a named type with name tn. We use tn for a type name. We further assume a type map TM, which is a finite map from type names to their type definitions (that is, a collection of `typedefs`). For example, the named struct type "`struct Node {int value; Node *next}`" is represented as

$$\text{TM} = \{\texttt{Node} \mapsto \text{struct } \{\text{value} : \text{int};\ \text{next} : \texttt{Node}*\}\}$$

$$\begin{array}{ll} \textit{Type} & t := \text{int} \mid t_1 * \mid \text{struct } \{id_1 : t_1; \ldots; id_m : t_m\} \mid tn \\ \text{TM}: & \textit{TypeName} \rightharpoonup_{\textit{fin}} \textit{Type} \end{array}$$

Figure 4: Syntax of types and a type map from type names to their type definitions.

Fig. 5 presents the algorithm for type-based parameter tree building. Given an identifier id with type t, it builds a tree with root annotated with "$id : t$" and child trees based on components of t. Notation Tree($id : t, tr_1, \ldots, tr_m$) is for a tree that has root "$id : t$" and m child trees in tr_1 to tr_m. The algorithm in Fig. 5 is recursive. If t is a struct type, it recursively builds subtrees for field types before constructing a tree for the struct type. If t is a pointer type "t_1*", we construct a tree of root "$id : t$" and a subtree based on type t_1 and identifier $*id$.

Since types may be recursive (as in the case of the `Node` type), the build-tree algorithm adopts a k-limiting approach to stop expanding types after k expansions, avoiding an infinite expansion at the type level. This is implemented in the cases when t is a type name tn: it decreases k after expanding the type name using the type map TM and stops expanding when k hits zero. Our implementation fixes k to be one. For the example `Node` type, the 1-limiting parameter tree is presented in Fig. 6.

Semantically, each node in a parameter tree represents an abstract memory region. The type on the node tells the type of elements stored in the memory region. Take the tree of Fig. 6 as an example: the root node represents an abstract memory region that holds a sequence of `Node*` pointers (it is a sequence as `head` may actually point to an array of `Node*` elements); the "`(*head).next` node represents a sequence of `Node*` pointers as well as all storage those pointers can reach.

Computing intra-procedural dependence. A function's PDG is built as follows: (1) add nodes for instructions in the function; (2) build formal parameter trees for parameters of the function; (3) for a call instruction in the function, build actual parameter trees for arguments of the call; (4) add intra-procedural dependence edges. For a function, we only need to build its formal parameter trees once; by contrast, the actual parameter trees need to be built per function call site.

We next discuss how dependence edges are computed. Intra-procedural dependence edges for a function consists of control-dependence and data-dependence edges. Control dependence can be computed with a classic algorithm [11] based on post-dominator trees.

Def-use data dependence can be computed easily because LLVM IR uses the SSA (Static Single Assignment) form. For a use of a variable in an instruction, it suffices to find the single definition of the variable and add a edge from the definition to the use. A function parameter is conceptually defined at the beginning of the function; therefore, data-dependence edges are added from the root node of a parameter's tree representation to uses of the parameter.

RAW (Read-After-Write) data dependence is computed with the help of an intra-procedural pointer analysis. In LLVM IR, only `store` instructions can write to memory and only `load` instructions can read from memory. Therefore, for each `load` instruction, our implementation checks every `store` instruction in the same

$$\text{buildTree}(t, id, k) = \begin{cases} \text{Tree}(id : t) & \text{if } t = \text{int} \\ \text{Tree}(id : t, tr_1) & \text{if } t = t_1 * \text{ and } tr_1 = \text{buildTree}(t_1, *id, k) \\ \text{Tree}(id : t, tr_1, \ldots, tr_m) & \text{if } t = \text{struct } \{id_1 : t_1, \ldots, id_m : t_m\} \\ & \text{and } tr_i = \text{buildTree}(t_i, (id).id_i, k) \text{ for } i = 1 \ldots m \\ \text{buildTree}(\text{TM}(tn), id, k-1) & \text{if } t = tn \text{ and } k > 0 \\ \text{Tree}(id : \text{TM}(tn)) & \text{if } t = tn \text{ and } k = 0 \end{cases}$$

Figure 5: Type-based parameter-tree building.

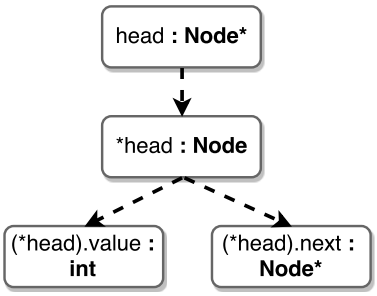

Figure 6: Parameter tree for head of type `Node *`.

function and see whether their destination memory locations can overlap, using the DSA pointer analysis [18]; if so, an edge is added from the `store` to the `load` instruction. This construction is flow-insensitive as it ignores the ordering of instructions; it makes an over-approximation.

In addition, we add RAW data-dependence edges between instruction nodes and parameter-tree nodes; examples can be found in Fig. 3 (note for succinctness the figure omits RAW labels on data-dependence edges). Conceptually, nodes in a formal parameter tree of a function represent potential reads/writes in the callers of the function; therefore, if the function has a load/store instruction and the instruction accesses memory regions represented by a parameter-tree node, a data-dependence edge should be added between the instruction's node and the parameter-tree node. Similarly, nodes in an actual parameter tree at a function call site conceptually represent potential reads/writes in the callee function; therefore, data-dependence edges are also added between corresponding instruction nodes and nodes in actual parameter trees.

Computing inter-procedural dependence. With parameter trees, inter-procedural dependence representation becomes trivial. For each function call site, we just connect nodes in the actual parameter trees to the corresponding formal parameter trees of the callee function, using bidirectional flow edges.

We note that library function calls (e.g., calls to *scanf*, *printf*, *exit*...) are represented as regular instruction nodes with dependence edges added according to the library functions' semantics. Alternatively, we could treat library functions as ordinary functions and represent them using PDGs based on their source code, but it would substantially increase the PDG size.

4.4 PDG-Based Program Partitioning

PtrSplit's partitioning algorithm takes the PDG of a program and separates it into a sensitive partition with access to sensitive data and an insensitive partition for the rest of the code. The algorithm performs function-level partitioning and does not split a single function. Furthermore, since our PDG represents both data and control dependence, the algorithm considers both explicit flows of sensitive data (via data dependence) and implicit flows (via control dependence) when deciding what part of code may have access to the sensitive data.

Algorithm 1 PDG-based program partitioning

Input: G is a PDG
Output: F_s: the set of sensitive functions; Gl_s; the set of sensitive global variables

```
sensitive ← {n | n is marked sensitive}
worklist ← sensitive
while worklist is not empty do
    n ← worklist.pop()
    for data/control dependence edge n → n′ do
        if n′ is not declassified and n′ ∉ sensitive then
            sensitive ← {n′} ∪ sensitive
            worklist ← {n′} ∪ worklist
F_s ← {f | f has a node n in sensitive}
Gl_s ← {g | g's parameter tree has a node n in sensitive}
```

The partitioning algorithm is standard. The focus of the paper is on supporting program partitioning in the presence of general pointers so that any partitioning algorithm can be supported no matter where the algorithm decides to split the program. There are many interesting aspects of the partitioning algorithm that can be improved, including performing instruction-level partitioning instead of function-level partitioning and balancing between performance and security. We plan to explore these issues in future work (discussed in Sec. 8).

Algorithm 1 presents the PDG-based partitioning algorithm. The input is a PDG and the output is a set of functions F_s and a set of global variables Gl_s that should be put into the sensitive partition; the rest of the program is in the insensitive partition. The *sensitive* set starts with the set of nodes that programmers mark as sensitive using attributes (an example is line 1 in Fig. 2). Then a worklist algorithm is used to compute the set of nodes that a sensitive node can reach along the data-dependence edges (explicit data flow) and the control-dependence edges (implicit data flow) in the PDG, while excluding nodes that programmers mark as declassified nodes (also using attributes). At the end of the algorithm, any function whose PDG contains sensitive nodes is put into the

set of sensitive functions and any global variable whose parameter-tree representation contains sensitive nodes is put into the set of sensitive global variables.

For the example PDG in Fig. 3, the node with label "`*key:char`" is marked sensitive. As a result, the `encrypt` function is sensitive because it has a node with an incoming data-dependence edge from "`*key:char`". Similarly, `initkey` is marked sensitive (its PDG is not shown in Fig. 3). Then node "`*ciphertext:char`" is marked sensitive because of an incoming data-dependence edge. Consequently, `main` is marked sensitive (because of an edge from "`*ciphertext:char`" to a node in `main`, not shown in Fig. 3). In contrast, if node "`*ciphertext:char`" were marked declassified, then `main` would not be marked sensitive.

5 SELECTIVE POINTER BOUNDS TRACKING

As discussed before, a core challenge in partitioning C/C++ programs is that pointers do not carry the bounds of the underlying buffers, making marshalling/unmarshalling of pointer data a manual and error-prone process. Bounds information is also critical for another security application: spatial memory safety. There have been many systems (e.g., [8, 9, 24, 25]) that track bounds information as metadata on buffers or pointers and insert checks before pointer operations to prevent out-of-bound buffer access. However, systems that enforce spatial memory safety incur high performance overhead; e.g., SoftBound's performance overhead on the SPEC and Olden benchmarks is 67% on average.

For marshalling and unmarshalling it is necessary to perform only bounds tracking, but not bounds checking. That is, it is sufficient to track the bounds of pointers without performing bounds checking on pointer accesses; even if the insensitive partition had an out-of-bound pointer, it would not be able to access the sensitive data through the pointer as it is in a separate process. We further observe that it is necessary to track the bounds of pointers that can cross the boundary of partitions, but not the bounds of all pointers. Therefore, by performing only bounds tracking for a subset of pointers, the performance overhead should be lower than those systems that enforce spatial memory safety.

Based on this observation, we have designed a Selective Pointer Bounds Tracking (SPBT) system, which (1) computes a set of Bounds-Required (BR) pointers given a partitioning of the program, and (2) instruments the program to track the bounds of those BR pointers.

Computing the set of bounds-required pointers. The algorithm for computing the set of BR pointers is presented in Algorithm 2. It operates on a PDG and takes as input a partitioning of the program, in the form of two sets of functions F_0 and F_1, one for each partition. The BR set is initialized with the set of pointers that are sent across from one partition to the other partition; obviously, bounds information are required for automatic marshalling and unmarshalling of these pointers.

With a backward propagation process along the data-dependence edges in the PDG, the algorithm further computes the set of pointers whose values can flow transitively to the initial BR set. Such pointers also need bounds information because, when a pointer $p1$'s value flows to $p2$, the bounds of $p2$ is set according to the bounds of $p1$; therefore, if $p2$ is sent over the partition boundary afterwards, $p1$'s bounds need to be tracked as well. As an example, suppose

Algorithm 2 Compute a set of BR pointers

Input: G is the PDG for a program; F_0 and F_1 are two disjoint sets of functions that cover the program
Output: P is the set of bounds-required pointers

$BR \leftarrow \emptyset$
for function $f \in F_0 \cup F_1$ and call $C \in f$ **do**
  if ($f \in F_i$ and C's callees $\cap F_{1-i} \neq \emptyset$) **then**
    for node n in C's parameter trees **do**
      if n's label is $(id : t*)$ **then**
        $BR \leftarrow BR \cup \{(n, id)\}$
$worklist \leftarrow BR$
while *worklist* is not empty **do**
  $(n, id) \leftarrow worklist.pop()$
  for data-dependence edge $n' \rightarrow n$ with label id_1 **do**
    if alias(id, id_1) **then**
      for all pointer-typed id' in n' **do**
        if $(n', id') \notin BR$ **then**
          $BR \leftarrow BR \cup \{(n', id')\}$
          $worklist \leftarrow worklist \cup \{(n', id')\}$
$P \leftarrow \{id \mid (n, id) \in BR\}$

$p1$ is the result of a memory allocation and its value flows to $p2$, which is sent over the boundary; it is then necessary to create the bounds information for $p1$ at the site of memory allocation and then propagate the information from $p1$ to $p2$.

The algorithm tracks a set of pairs of nodes and identifiers in the sensitive set, instead of a set of nodes. This improves the precision of the algorithm. To illustrate, suppose the PDG has a node n for instruction "$p1 = p2 + i$", where $p1$ is a BR pointer and i is an integer. The algorithm then puts $(n, p2)$ into the BR set and performs further processing along n's incoming data-dependence edges; during this processing, all edges with label i can be ignored. Such distinction could not be made if the algorithm used a set of nodes in the BR set.

SPBT instrumentation. PtrSplit's SPBT instrumentation is based on SoftBound [24], an LLVM-based code transformation for enforcing spatial memory safety (another version also enforces temporal memory safety). For each pointer, SoftBound keeps its base and bound. Metadata is created for pointers at allocation sites. Metadata is propagated along with the propagation of pointer values, for example, when pointers are passed during function calls. Finally, before load/store instructions, metadata is used to check for memory-safety violations.

Our SPBT instrumentation removes memory-safety checking before load/store instructions. Furthermore, at an allocation site, if the returned pointer is not in the set of BR pointers (as computed by Algorithm 2), SPBT removes instrumentation that records the pointer's base and bound metadata. Similarly, when pointer values are propagated, if the involved pointers are not in the set of BR pointers, the instrumentation that propagates metadata is removed.

6 TYPE BASED MARSHALLING AND UNMARSHALLING

Since partitions are loaded into separate processes, some function calls are turned into Remote Procedure Calls (RPCs). During an RPC, arguments from the caller are marshalled into a data buffer and sent to the callee, which unmarshalls the data buffer and recreates the values for the parameters in the callee process. Data marshalling is straightforward for values of most data types, including integers, arrays of fixed sizes, and structs.

However, pointer types cause many troubles. First, C pointers do not carry bounds information; so marshalling does not know the sizes of underlying buffers and cannot marshall the buffers as a result. Second, it is possible to create recursive data structures such as linked lists and arbitrary aliases when using pointers, which makes marshalling/unmarshalling difficult. For instance, if the caller sends a pointer that points to a circular linked list, after marshalling and unmarshalling, a linked list with the same circularity and aliasing should be recreated in the callee process.

Previous program-partitioning frameworks [5, 32] avoid the pointer issue by asking programmers to write manual marshalling and unmarshalling code when pointer data are involved. General Interface Description Languages (IDLs) also do not provide a satisfactory solution. For instance, the Microsoft COM IDL [4] requires manual annotation about the size of a variable-sized array and also annotation about aliasing when multiple pointers are involved. The popular SWIG IDL [2] adopts the approach of *opaque pointers*: pointers are wrapped as opaque objects and are sent over the boundary without copying the underlying buffers; whenever the callee domain needs to perform operations on those pointers, the control is transferred back to the caller domain for the actual operations. The opaque-pointer approach avoids the pointer issue, but it may lead to frequent domain crossings; further, it may cause a security problem if an untrusted partition can spoof opaque pointers to read arbitrary memory; some solution for opaque pointer integrity would be needed. Finally, popular RPC libraries (e.g., Google's gRPC [12] and Oracle's TI-RPC [27]) also do not provide good support for pointers and require manual intervention.

Thanks to SPBT, all pointers that cross the partition boundary in our system are equipped with bounds information, making it possible to automatically marshall/unmarshall even pointer data. Therefore, we propose the approach of *type-based deep copy* of pointer data: a pointer value is marshalled along with the buffer the pointer points to; if the buffer itself contains pointers, those pointers are marshalled recursively; the callee process unmarshalls the received data to recreate the pointer and the buffer, also in a recursive way; furthermore, as we will discuss, our approach takes care of circularity and aliasing in data.

Before proceeding, there are several points worth mentioning. First, the deep-copying approach is type directed and relies on types to identify pointers within data to be marshalled; consequently, if an application performs a type cast on some data and the result type hides pointers, some necessary data may not be deep copied. More discussion about this (especially on void pointers) is in Section 8.

Another concern about deep copying is its efficiency, when a large amount of data needs to be marshalled during deep copying. However, our main focus in this paper is to enable any partitioning of an application, even if the partitioning creates the situation of sending pointer data across the partitioning boundary. A good partitioning algorithm would take efficiency of deep copying into account when choosing among multiple partition choices and balance between efficiency and security; this will be an interesting research direction.

We also mention that the deep copying approach is not the only design choice. An alternative would be the opaque-pointer approach we discussed before; however, it would create very frequent domain crossings, which we would like to avoid. Another approach is to set up a shared memory region between the two partitions for communication; this could potentially eliminate some data copying. However, this assumes a custom memory allocator or some level of programmer cooperation so that relevant data is put into the shared memory. For instance, if a linked list is sent across the boundary, all nodes in the linked list have to be allocated in the shared memory. This is a nontrivial assumption and requires either programmers to transform their code or at least the support of automatic program transformation.

6.1 Algorithm for Deep Copying

We next present a formal algorithm for type-based marshalling/unmarshalling that performs deep copying of pointer data. We will use the same set of types in Fig. 4 when presenting the algorithm. In addition, the syntax for values is as follows:

$$\textit{Value} \quad v := n \mid \text{struct}\ \{id_1 = v_1, \ldots, id_n = v_n\} \mid p_{(b,e)}$$

A value can be an integer n, a struct value with field values inside, or a pointer value of the form $p_{(b,e)}$. After SPBT, all pointers that cross the boundary have bounds information in the form of (b, e), where b is the beginning of the underlying buffer, e is the end of the buffer, and the buffer size is $e - b$. A null pointer is encoded as $0_{(0,0)}$ (that is, it points to an empty buffer).

Type-based marshalling. Fig. 7 presents a recursive algorithm for encoding a value v of type t into a list of bytes. In the figure, we use [ ] for an empty list, and $l_1 + l_2$ for the concatenation of two lists. The algorithm assumes a list of utility functions, which are explained in the caption.

The case when t = int is simple; just encode the type and the integer. For a struct type, all field values and their types are encoded. For a named type *tn*, the value is encoded according to the type definition for *tn* as defined in the type map TM.

The case for a pointer type is challenging since the algorithm has to deal with circularity caused by pointers. For that, the encode function also takes a parameter B, which remembers a list of buffers (in the form of (b, e)) that have already been encoded; when encoding a pointer that points to an already encoded buffer, there is no need to encode the buffer again. If the buffer has not been encoded, (b, e) is added to B and every element in the buffer is then encoded recursively (with the help of function $\text{enc\_buf}_B(b, e, t)$).

A marshalling example. As an example, assume we need to marshall a circular linked list of two nodes, shown in Fig. 8. Each node is of type `Node` with two fields, one is type int and one `Node*`; each field is assumed to occupy four bytes. To marshall this data

$$\text{encode}_B(v,t) = \begin{cases} \text{enc\_typ}(t) + \text{enc\_int}(n) & \text{if } t = \text{int and } v = n \\ \text{enc\_typ}(t) + l_1 + \ldots + l_n & \text{if } t = \text{struct } \{id_1 : t_1, \ldots, id_n : t_n\} \\ & \text{and } v = \text{struct } \{id_1 = v_1, \ldots, id_n = v_n\} \\ & \text{and } l_i = \text{encode}_B(v_i, t_i) \text{ for } i \in [1..n] \\ \text{enc\_typ}(t) + \text{encode}_B(v, \text{TM}(tn)) & \text{if } t = tn \\ \text{enc\_typ}(t) + \text{enc\_ptr}(p_{(b,e)}) + l_{buf} & \text{if } t = t_1 * \text{ and } v = p_{(b,e)} \\ & \quad l_{buf} = \begin{cases} [\,] & \text{if } (b,e) \in B \\ \text{enc\_buf}_{B \cup \{(b,e)\}}(b,e,t_1) & \text{otherwise} \end{cases} \end{cases}$$

$$\text{enc\_buf}_B(b,e,t) = \begin{cases} \text{encode}_B(v,t) + \text{enc\_buf}_B(b + \text{size}(t), e, t) & \text{if } b + \text{size}(t) \le e \text{ and } v = \text{read\_mem}(b,t) \\ [\,] & \text{otherwise} \end{cases}$$

Figure 7: Type-based marshalling. In the algorithm, we assume a set of basic utility functions: enc_typ(t) **for encoding a type into a list of bytes;** enc_int(n) **for encoding an integer;** enc_ptr($p_{(b,e)}$) **for encoding a pointer;** size(t) **for the size of values in type** t**;** read_mem(b,t) **for reading a value of type** t **from memory at address** b**.**

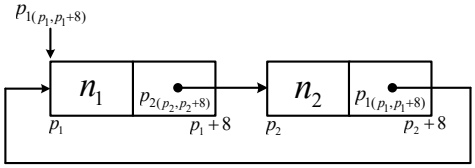

Figure 8: A two-node circular linked list.

structure, we make the following call:

$$\text{encode}_{\emptyset}(p_{1(p_1,p_1+8)}, \text{Node}*)$$

This call encodes the p_1 pointer as well as the buffer it points to; the buffer contains the first node (viewed as an array of one node). When encoding the buffer, because of the pointer inside the first node, the encoder is recursively invoked as follows:

$$\text{encode}_{\{(p1,p1+8)\}}(p_{2(p_2,p_2+8)}, \text{Node}*)$$

This call encodes the p_2 pointer and the second node. Since the second node contains another pointer, the following call is triggered:

$$\text{encode}_{\{(p1,p1+8),(p2,p2+8)\}}(p_{1(p_1,p_1+8)}, \text{Node}*)$$

At this point, however, only the pointer is encoded, not the underlying buffer since it has already been encoded.

Typed-based unmarshalling. Fig. 9 presents the algorithm for type-based unmarshalling. The decode function takes a list of bytes and returns a value, a type, and the remaining list of bytes that have not been decoded. The cases for integer types, struct types, and named types are straightforward.

For a pointer type, the algorithm needs to remember the map between buffers in the sender partition and buffers in the receiver partition. This is why the decode function has an additional parameter M for remembering the map. There are two cases, for pointer $p_{(b,e)}$ that is sent, if (b,e) is not recorded in M, then the receiver has not allocated an corresponding buffer yet; in this case, a new buffer is allocated and initialized by the dec_buf function. If (b,e) has been recorded in M, then the corresponding buffer has already been allocated and there is no need for reallocation. In both cases, the returned pointer value uses the bounds information of the buffer in the receiver and p is adjusted to be $b' + p - b$ to maintain the offset between the pointer and the beginning of the buffer.

For the example of circular linked lists, the decoder allocates a node for each node in the original linked list and at the same time adjusts pointer values according to the buffer map M.

Other issues and LLVM implementation. The previous algorithm shows how to marshall/unmarshall one argument, but our implementation marshalls and unmarshalls all arguments at the same time. This is important, not just for efficiency, but for correctness in the case when multiple pointer arguments alias the same buffer; the buffer should be encoded just once so that the receiver can recreate aliasing. Essentially, this approach treats multiple arguments as a value of a tuple type.

When a pointer is passed from a caller to a callee partition, PtrSplit performs deep copying of pointer data. Since the callee may modify such data, it is necessary to copy back the entire pointer data from the callee and caller at the end of the RPC call. This implements the copy-in and copy-out semantics for pointer data, which is compatible with single-threaded code.

After marshalling, arguments of a function call are encoded as a byte array, which is sent to the receiver via the help of an RPC library. We use the popular TI-RPC library [27] for sending and receiving byte arrays.

In our system, deep copying of pointer data applies to only user-space data pointers. Our implementation maintains a whitelist of other kinds of pointers that are not deep copied, including pointers to OS-kernel data structures and pointers to code. It is not possible to deep copy these pointers; therefore we adopt the opaque-pointer approach for them. For instance, when one partition creates a file pointer through the OS, our marshalling wraps the file pointer as an opaque object without performing deep copying. The receiver is transformed to send the file pointer back to the sender for operating on the underlying file. For a code pointer that crosses the boundary, our system also wraps it as an opaque pointer with a runtime tag; an indirect call via a code pointer is instrumented to decide whether the code pointer is local or remote before performing a local or an RPC call.

7 EVALUATION

The latest versions of LLVM do not support DSA, the alias analysis that PtrSplit uses. Therefore, we implemented PtrSplit in LLVM 3.5, an older version of LLVM. SoftBound's public release only supports

$$\text{decode}_M(l) = \begin{cases} (n, \text{int}, l_2) & \text{if dec\_typ}(l) = (\text{int}, l_1) \text{ and dec\_int}(l_1) = (n, l_2) \\ (\text{struct } \{id_1 = v_1, \ldots, id_n = v_n\}, t, l_{n+1}) & \text{if dec\_typ}(l) = (t, l_1) \text{ and } t = \text{struct } \{id_1 : t_1, \ldots, id_n : t_n\} \\ & \text{and decode}_M(l_i) = (v_i, t_i, l_{i+1}) \text{ for } i \in [1..n] \\ (v, tn, l_2) & \text{if dec\_typ}(l) = (tn, l_1) \text{ and decode}_M(l_1) = (v, \text{TM}(tn), l_2) \\ ((b' + p - b)_{(b', e')}, t_1*, l_3) & \text{if dec\_typ}(l) = (t_1*, l_1) \text{ and dec\_ptr}(l_1) = (p_{(b,e)}, l_2) \\ & \text{and } (b, e) \notin \text{dom}(M) \text{ and } b' = \text{alloc}(e - b) \text{ and} \\ & e' = b' + e - b \text{ and } l_3 = \text{dec\_buf}_{M \cup \{(b,e) \mapsto (b', e')\}}(b', e', t_1, l_2) \\ ((b' + p - b)_{(b', e')}, t_1*, l_2) & \text{if dec\_typ}(l) = (t_1*, l_1) \text{ and dec\_ptr}(l_1) = (p_{(b,e)}, l_2) \\ & \text{and } (b, e) \in \text{dom}(M) \text{ and } (b', e') = M(b, e) \end{cases}$$

$$\text{dec\_buf}_M(b, e, t, l) = \begin{cases} l_2 & \text{if } b + \text{size}(t) \le e \text{ and decode}_M(l) = (v, t, l_1) \\ & \text{and write\_mem}(b, v) \text{ and dec\_buf}_M(b + \text{size}(t), e, t, l_1) = l_2 \\ l & \text{otherwise} \end{cases}$$

Figure 9: Type-based unmarshalling. Function dec_typ(l) **is for decoding a type in the first bytes of** l**;** dec_int(l) **for decoding an integer;** dec_ptr(l) **for decoding a pointer;** alloc(n) **for allocating a buffer of size** n**;** write_mem(b, v) **for writing** v **at address** b **in memory.**

LLVM 3.4; so we had to upgrade its code base to support LLVM 3.5. Several LLVM passes were added to implement the components of PtrSplit. We evaluated PtrSplit using a set of benchmarks on a system running x86-64 Ubuntu 14.04 with the Linux kernel version 3.19.0, an Intel Core i5-4590 at 3.3GHz, and 16GB of physical memory.

The evaluation aims to answer several questions: (1) whether PtrSplit can automatically partition realistic C applications and scale to relatively large C applications, (2) whether the performance overhead of a partitioned application is acceptable, given the overhead of performing SPBT and deep copying of RPC data, (3) whether SPBT significantly reduces the overhead, when compared with a solution that enforces full spatial memory safety.

We first evaluated with a set of microbenchmarks to validate the major functionalities of PtrSplit. The programs include the running example in Fig. 2 and programs that send data structures (including trees, linked lists, and circular linked lists) over RPC calls.

We then evaluated PtrSplit with a set of security-sensitive programs and programs from SPECCPU 2006. For each program, we ran its partitioned version and checked that the partitioned version functioned well using the reference data set attached with the program. During the process, we also measured the performance overhead of the partitioned version. These experiments are detailed next.

Security-sensitive programs. We evaluated PtrSplit on four security-sensitive programs. Considering that all of these programs are networking programs, which are greatly affected by the network latency, we used another machine that was in the same LAN as a remote server. The remote server machine has the same hardware and OS configuration as the local machine. For each program, we analyzed its functionality and marked some sensitive data that need isolation; recall that sensitive data means data of either high secrecy or low integrity. Then PtrSplit is used to partition these programs to isolate sensitive code and data into a separate partition. Results for these programs are presented in Table 1. We next discuss in detail how experiments were performed for each program.

`ssh` is a networking utility included in `OpenSSH` (version: 7.4p1), which is a suite of utilities based on the SSH protocol. The `ssh` utility implements the client-side of the SSH protocol. We annotated the buffer that receives the RSA private key as the sensitive data. We also declassified the return results of functions `sshkey_load_file` and `sshkey_load_private`; although these functions compute on sensitive data, their return results are status/error codes, which are not correlated to sensitive data. (The reason for declassification in `wget` and `telnet` is the same and we will not repeat it when we discuss those programs.) In total, twelve functions were put into a sensitive partition. For measuring performance overhead, we used our partitioned `ssh` to log in to the server one hundred times.

`wget` (version: 1.18) is a command-line program for retrieving files from a remote HTTP or FTP server. We annotated the buffer for receiving the downloaded file from an FTP server as the sensitive data because the file may contain malicious content. We also declassified the return results of functions `fd_read_body` and `skip_short_body`. For measuring performance overhead, we downloaded a 1KB file from the FTP server one hundred times.

`thttpd` (version: 2.27) is an open-source http server program. We chose its authentication file as the sensitive data, and annotated the corresponding buffer that reads contents from the authentication file in the source code; a single declassification annotation was also added to declassify the result of function `auth_check`. After separation, five functions that access the authentication-file buffer were put into a sensitive partition. To measure the average overhead, we set up a server on the remote machine with our partitioned `thttpd` and downloaded a 1KB file on that server multiple times through a local client.

`telnet` (version: inetutils-1.9.4) is a networking client utility based on the telnet protocol. We consider the threat of a remote entity that pretends to be a server and the client somehow connects to the fake server (in a phishing attack, e.g.) and the fake server tries to use a vulnerability to attack the client. To counter the threat, we annotated the buffer that receives packets from the server as the sensitive data because the received packets may contain malicious content. We also declassified the return results of functions `telrcv`, `ttyflush` and `process_rings`. In total, eleven functions were put into a sensitive partition. We measured the average performance overhead of using our partitioned telnet to log in to a remote server one hundred times.

Benchmark	SLOC	Sensitive Data	# of functions/ sensitive functions	Total/BR pointers	PBT overhead	SPBT overhead	Total overhead
ssh	64,671	private key file	1235/12	21020/591	45.0%	2.6%	7.4%
wget	61,216	downloaded file	666/8	14939/466	52.5%	3.4%	6.5%
thttpd	21,925	authentication file	145/5	3068/189	56.3%	3.6%	8.8%
telnet	11,118	received data from server	180/11	2068/233	74.1%	5.1%	9.6%

Table 1: Partitioning results of security-sensitive programs. (Abbreviations: "Total pointers": total pointer variables in LLVM-IR; "BR pointers": bounds-required pointer variables; "PBT": pointer bounds tracking; "SPBT": selective pointer bounds tracking.)

Overall, our experiments showed promising results, shown in Table 1. For each program, the table lists its lines of source code, the sensitive data, the total number of functions in the program versus the number of functions in the sensitive partition computed by the partitioning algorithm, the total number of pointers (i.e., static counts of pointer variables) versus the number of Bounds-Required (BR) pointers computed by SPBT, the performance overhead (compared to the vanilla, uninstrumented program) when full pointer bounds tracking is applied, the performance overhead when only SPBT is applied, and the total performance overhead for the partitioned application.

As shown in the table, SPBT is effective at reducing the overhead of pointer bounds tracking and the overall performance overhead of the security-sensitive applications is acceptable. They demonstrated that PtrSplit can be used for partitioning realistic security-sensitive applications to improve security, with a modest amount of performance overhead.

SPECCPU 2006 benchmarks. We then evaluated PtrSplit using the SPECCPU 2006 C benchmarks. These programs are compute-intensive benchmarks and are not security-sensitive benchmarks. However, we felt it is important to evaluate PtrSplit using compute-intensive benchmarks as they stress test the instrumentation mechanism of PtrSplit; furthermore, we would like to compare the performance overhead of SPBT with the overhead of full pointer bounds tracking (PBT) on SPEC benchmarks. For each of the benchmarks, we randomly selected a global variable, marked it sensitive, and fed it to PtrSplit; in this experiment, only explicit flows are taken into account and no declassification is used during partitioning since it is not for evaluating security but for evaluating the instrumentation mechanism.

Table 2 presents the experimental results. We note that three programs (`perlbench`, `gcc`, and `gobmk`) were excluded because SoftBound's memory-safety instrumentation produces runtime crashes due to SoftBound's implementation bugs and PtrSplit's SPBT implementation is on top of SoftBound. The original SoftBound paper also did not report results on `perlbench` and `gcc`; further, a recent paper [6] reported the difficulty of instrumenting SPEC benchmarks using SoftBound. We also excluded `mcf` because it is a small program with 24 functions and any global variable marked as sensitive would lead to all functions being in one partition (adding declassification annotations would produce a separation, but we refrained from doing so since it is unclear where to declassify based on a randomly selected global variable).

For SPBT, we can see from the table the total number of pointers that require bounds is typically a small percentage of the total number of pointers in a program (we counted the number of pointers statically, based on their types). As a result, the average SPBT runtime overhead for the benchmarks is 7.2%, which is much lower than the average overhead of 136.2% for full pointer bounds tracking (PBT). This shows the effectiveness of SPBT. Note that `milc` has no BR pointers because no pointer data are passed between the created partitions.

The runtime overhead of PtrSplit comes from two sources: pointer bounds tracking and data marshalling/unmarshalling for RPC calls. Table 2 also shows the total runtime overhead. `libquantum`'s overhead is rather large; we found that RPC call overhead is positively correlated to the RPC call frequency. For `libquantum`, the randomly selected variable leads to a partitioning with a high RPC call frequency (94 Hz); the RPC call frequency of other benchmarks is below 3Hz. Choosing a different global variable of `libquantum` would lead to a similar result.

To further validate the robustness of our partitioning framework, for each SPEC benchmark, we built a script that randomly splits the benchmark's set of functions into two disjoint sets of functions and creates a partitioning based on the split. The script was run multiple times and for each run we checked that the partitioned application worked as intended (using the reference data set included in SPECCPU 2006). Some of these random partitionings created complex interfaces that required exchange of complex data (structs, pointers, etc.) and provided good stress tests of PtrSplit's RPC mechanism. Table 3 presents the results. For each benchmark, the table includes the BR-pointer ratio (the number of BR pointers divided by the number of total pointers), the SPBT overhead, and the total overhead, averaged over multiple runs of performing random partitioning. The total overhead is on the high side, which indicates random partitioning would not lead to efficient partitionings.

8 LIMITATIONS AND FUTURE WORK

In this section, we discuss current limitations of PtrSplit and how it can be extended to address them. PtrSplit's PDG-based partitioning algorithm can be extended to produce multiple partitions instead of just two. Programmers can use attributes for different kinds of sensitive data (e.g., one for networking data and one for data retrieved from a database) and then the same reachability-based algorithm can be used to produce multiple partitions, one for processing one kind of sensitive data. It is possible that a function can access multiple categories of sensitive data, in which case the function can be duplicated in multiple partitions. Another design would be to employ a security lattice; all functions that access the same categories of sensitive data are assigned the same label and put into their own partition.

Benchmark	SLOC	Sensitive data and type	# of functions/ sensitive functions	Total/BR pointers	PBT overhead	SPBT overhead	Total overhead
lbm	1,156	LBM_Grid* srcGrid	19/5	695/131	141.4%	19.7%	24.3%
libquantum	4,358	struct quantum_reg* lambda	115/3	1690/128	282.3%	11.2%	179.2%
bzip2	8,393	char* progName	100/6	4356/8	59.4%	3.1%	5.3%
sjeng	13,547	char* realholdings	144/5	3415/81	41.7%	3.4%	10.2%
milc	15,042	double[] path_coeff	235/2	5001/0	111%	0%	2.2%
sphinx3	25,090	char** liveargs	369/3	9491/37	90.5%	5.1%	7.1%
hmmer	35,992	int ser_randseed	538/7	17692/175	128.5%	5.8%	26.7%
h264ref	51,578	int[] FirstMBInSlice	590/5	32212/461	234.4%	9.6%	15.5 %
Average					136.2%	7.2%	33.8%

Table 2: Partitioning results for SPECCPU 2006 benchmarks (use a random global as sensitive variable).

Table 3: The random partitioning results for SPECCPU 2006 benchmarks.

Benchmark	Average BR-pointer ratio	Average SPBT overhead	Average total overhead
lbm	14.3%	15.4%	55.1%
libquantum	16.2%	51.5%	163.3%
bzip2	12.4%	16.4%	71.3%
sjeng	15.2%	14.1%	63.9%
milc	10.7%	23.4%	83.2%
sphinx	8.7%	17.9%	37.5%
hmmer	8.8%	29.8%	89.7%
h264ref	9.1%	38.4%	101.9%
Average	11.8%	29.4%	79.3%

A second straightforward improvement of the partitioning algorithm is to produce partitions that balance between security and performance. A program profiling tool can be used to profile the frequency of function calls and size of data sent over function calls; such performance numbers can be used to annotate PDG edges. Then an algorithm such as the one used by ProgramCutter [32] can be used to produce a partitioning that takes into account of both security and performance. Another interesting direction is to target specific application domains. For instance, OS developers have long been interested in privilege separating kernel code and there had been manual privilege separation effort [13]; similarly, there is a need to partition legacy applications to be compatible with a trusted execution environment such as Intel's SGX. Finally, extending the support to C++ applications requires extending our type-based marshalling and unmarshalling to cover more types including C++ classes.

PtrSplit automatically partitions single threaded code and it will be a technical challenge to extend it to cover multi-threaded code. One issue with multi-threading is that the computation of data dependence is more complex, because of shared data between threads. Furthermore, the mechanism of deep copying pointer data naturally leads to a sequential execution model: pointer data is copied in at the beginning of an RPC call and copied out at the end. For a multi-threaded application, one thread can perform an RPC call, which leads to a copy of the passed pointer data; without synchronization, a second thread can modify the original pointer data, while the callee can separately modify its own copy. Proper synchronization code needs to be generated for pointer data, while not sacrificing too much performance.

PtrSplit also makes assumptions about how types are used in order for partitioned applications to function correctly. We mentioned before that PtrSplit's computation of call edges in PDGs assumes the lack of certain type casts on function-pointer types. Another assumption is that pointers can be identified through the types of cross-partition data. This assumption may be violated by type casts. As an example, suppose an application casts the type of a struct pointer to a void pointer. For bounds tracking, LLVM treats a void pointer as an `i8*` pointer (pointer to bytes) and SoftBound can track its bounds. As a result, a void pointer across the boundary is marshalled as a byte array. This is problematic when the original struct contains other pointers as they are hidden from the type-based marshalling process. Therefore, PtrSplit currently assumes void pointers do not appear at the boundary and raises an alarm when such a situation occurs. We did not encounter this case in our experiments. An alternative design would be to add runtime type information (RTTI) for data pointers and use the runtime types for marshaling and unmarshalling. As a side note, PtrSplit's PDG-construction builds on DSA alias analysis, which can handle type casts when calculating aliases; so dependence edges are not missed in PDGs because of type casts.

PtrSplit can be further optimized by using more efficient pointer bounds tracking tools to reduce the overhead. Recent work such as Low-Fat Pointers [14, 15] and CUP [6] seem to provide general pointer tracking with lower overhead than SoftBound, while still maintaining the Application Binary Interface (ABI). Unfortunately,

these systems are not yet open sourced. When they are available, we wish to combine our SPBT approach and these state-of-the-art bounds tracking techniques to make PtrSplit more efficient.

9 CONCLUSIONS

Automatic partitioning security-critical applications is an effective way of improving software security. It is important to support the automatic partitioning of C/C++ applications given their lack of memory safety and that trusted execution environments including SGX and TrustZone can run only native code; managed code such as Java bytecode cannot run directly inside SGX without first porting the whole language virtual machine into an SGX enclave. In this paper, we describe several techniques that support general pointers in C/C++ applications, including parameter trees, selective pointer bounds tracking, and type-based marshalling/unmarshalling. These techniques push forward the state-of-the-art of privilege separating C/C++ applications and experiments suggest they have the potential of making automatic program partitioning practical.

10 ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments, which helped us substantially improve the paper. This research is based upon work supported by US NSF grants CNS-1624126 and CNS-1408880. The research was also supported in part by NSF grant CCF-1624124 and the Defense Advanced Research Projects Agency (DARPA) under agreement number N66001-13-2-4040. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

REFERENCES

[1] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome. 2008. *The security architecture of the Chromium browser.* Technical Report.
[2] David M. Beazley. 1997. *SWIG Users Manual: Version 1.1.*
[3] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 309–322.
[4] Don Box. 1997. *Essential COM.* Addison-Wesley Professional.
[5] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th Usenix Security Symposium*. 57–72.
[6] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2017. CUP: Comprehensive User-Space Protection for C/C++. (2017). https://arxiv.org/abs/1704.05004v1.
[7] Stephen Chong, Jed Liu, Andrew Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 31–44.
[8] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure virtual architecture: a safe execution environment for commodity operating systems. *SIGOPS Oper. Syst. Rev.* 41 (Oct. 2007), 351–366. Issue 6.
[9] Dinakar Dhurjati and Vikram S. Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*. 162–171.
[10] Petros Efstathopoulos, Maxwell Krohn, Steve Vandebogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 17–30.
[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349.
[12] Google Inc. *GRPC: A high performance, open-source universal RPC framework.* Google Inc. http://www.grpc.io.
[13] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. 2016. Lightweight Capability Domains: Towards Decomposing the Linux Kernel. *SIGOPS Oper. Syst. Rev.* 49, 2 (Jan. 2016), 44–50.
[14] Gregory J.Duck and Roland H.C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *CC*.
[15] Gregory J.Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS*.
[16] Douglas Kilpatrick. 2003. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX track*. 273–284.
[17] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 321–334.
[18] C. Lattner, A. Lanharth, and V. Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 278–289.
[19] D. Liang and M. J. Harrold. 1998. Slicing objects using system dependence graphs. In *ICSM*. 358–367.
[20] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (ATC)*. 285–298.
[21] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *22nd ACM Conference on Computer and Communications Security (CCS)*. 1607–1619.
[22] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. 2016. Trellis: Privilege Separation for Multi-user Applications Made Easy. In *International Symposium on Research in Attacks, Intrusions and Defenses*. 437–456.
[23] Andrew Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology* 9 (Oct. 2000), 410–442. Issue 4.
[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI*. 245–258.
[25] George Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)*. 128–139.
[26] Ben Niu and Gang Tan. 2014. Modular Control Flow Integrity. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 577–587.
[27] Oracle. *Introduction to TI-RPC.* Oracle. https://docs.oracle.com/cd/E18752_01/html/816-1435/rpcintro-46812.html.
[28] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing privilege escalation. In *12th Usenix Security Symposium*. 231–242.
[29] Charles Reis and Steven D. Gribble. 2009. Isolating web programs in modern browser architectures. In *EuroSys*. 219–232.
[30] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated partitioning of Android applications for trusted execution environments. In *International Conference on Software engineering (ICSE)*. 923–934.
[31] Robert Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *19th Usenix Security Symposium*. 29–46.
[32] Yang Liu Yongzheng Wu, Jun Sun and Jin Song Dong. 2013. Automatically partition software into least privilege components using dynamic data dependency analysis. In *ASE*. 323–333.
[33] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. 2002. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS)* 20, 3 (2002), 283–328.
[34] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 263–278.
[35] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy (S&P)*. 236–250.