An Application-Level Approach for Privacy-preserving Virtual Machine Checkpointing

Yaohui Hu Tianlin Li Ping Yang Kartik Gopalan
Department of Computer Science, State University of New York at Binghamton, Binghamton, NY, USA
Email: {yhu15,tli16,pyang,kartik}@binghamton.edu

***Abstract*—Virtualization has been widely adopted in recent years in the cloud computing platform to improve server consolidation and reduce operating cost. Virtual Machine (VM) checkpointing refers to the act of saving a persistent snapshot (or checkpoint) of a VM's state at any instant. VM checkpointing can drastically prolong the lifetime and vulnerability of confidential or private user data in applications that execute within VMs. Simply encrypting the checkpoint does not reduce the lifetime of confidential data that should be quickly discarded after its use. In this paper, we present an application-level approach, called Privacy-preserving Checkpointing (PPC), which excludes confidential data from VM checkpoints, instead of encrypting such data. PPC enables an application programmer to register memory locations that represent the origins of confidential data. During the VM's execution, PPC performs information flow analysis to automatically track the propagation of confidential data through the application and various components of the VM, including the guest operating system. During VM checkpointing, the locations identified during the information flow analysis are excluded from the persistent checkpoint. We present the design and implementation of the PPC system in VirtualBox VMs running the commodity Linux operating system. We demonstrate the use of our system using the `vim` and `gedit` text editors. We also show that PPC introduces acceptable performance overhead.**

I. Introduction

Cloud computing enables users to provision remote resources to solve a large variety of computationally demanding problems. As the entity that controls the cloud makes the decisions about how the resources are provisioned and used, cloud users need to trust cloud providers to keep the confidential data in their applications protected from access by other users. Cloud providers extensively use virtual machines (VMs) to manage their cloud platforms. Automated tools for virtual resource management make it easy to instantiate and manage large numbers of VMs. On one hand, virtualization can help improve security through greater isolation and more transparent malware analysis and intrusion detection [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. On the other hand, it also gives rise to new privacy challenges for confidential data handled by applications that execute within these VMs.

This paper addresses the problem of protecting confidential application data during Virtual Machine (VM) checkpointing. VM checkpointing refers to the act of saving a permanent snapshot (or checkpoint) of a VM's state at any instant. A VM's state includes, at the minimum, its memory image and CPU execution state and possibly additional state such as virtual disk contents. The checkpoint can be later used for various purposes such as restoring the VM to a previous state, recovering a long-running process after a crash, distributing a VM image with a preset execution state among multiple users, archiving a VM's execution record, conducting forensic examination, etc. Most hypervisors such as VMware, Hyper-V, VirtualBox, KVM, and Xen support VM checkpointing.

Despite the above benefits, VM checkpoints can drastically prolong the lifetime and vulnerability of confidential data that should normally be discarded quickly after processing. Information such as passwords, credit card numbers, or trade secrets can now be saved forever in persistent storage through VM checkpointing. We have conducted an experiment in which we started the Firefox web browser in a VirtualBox VM, entered the credit card number in a commercial website, and checkpointed the VM. We found that, the checkpoint file contains the string "addCreditCardNumber={the credit card number we entered}", which would enable an attacker to locate the credit card number easily by searching for the string "CreditCard" in the checkpoint. The string can be found even if the VM is checkpointed *after* Firefox terminates. Users are not aware that their input data may still reside in memory even after the application that has processed such data terminates, and hence may mistakenly assume that checkpointing the VM is safe after terminating the browser. Therefore, automated tools are needed to identify memory locations that may contain confidential data and to safely exclude them from being checkpointed.

Encrypting the checkpoint can help protect confidential data if the data is needed after the restoration of the VM. However, encrypting the checkpoint is not suitable to protect confidential data that should be quickly discarded after its use due to the following reasons: (1) it does not reduce the lifetime of such data, (2) the VM can be restored from the checkpoint at any arbitrary time in the future, and hence exposing the data again, and (3) the checkpoint may be shared by multiple users and encryption does not prevent other authorized users from accessing the data. One can transparently exclude applications that have processed users' confidential data and their memory footprint from the VM checkpoint [11]. As a consequence, such applications will terminate when the VM is restored from the checkpoint. A better approach is needed if the user wants to keep the process alive after VM restoration, while excluding the confidential data from the VM checkpoint. Although existing work on taint analysis can potentially be used to track and identify memory locations that store confidential data, they are not suitable in the context of VM checkpointing due to the following reasons. Current techniques for static analysis

may over-estimate or under-estimate memory locations storing confidential data, whereas techniques for memory taint analysis [12], [13], [14] have a high performance overhead for large applications (e.g. web servers) due to the use of binary emulation or interpretation. Simply searching for the confidential data in the checkpoint does not guarantee to find all locations that store the confidential data because the data may be stored in multiple memory segments that are not adjacent to each other and may not be stored in clear-text.

In this paper, we present an application-level approach, called Privacy-preserving Checkpointing (PPC) that excludes confidential data from VM checkpoints. PPC uses a combination of static transformation and dynamic information flow tracking to identify memory locations that may store confidential data and to safely exclude them from being checkpointed. Application programmers can register the origin of confidential data, such as the input from user. An information flow analysis technique tracks how the confidential data propagates through the application and the guest operating system (e.g. through assignments, I/Os, sockets, etc) and automatically registers all confidential memory locations. In practice, the application programmers only need to register a small number of variables that are used to receive confidential data. For example, in the *vim* text editing application, a programmer needs to register only two variables, assuming that the text typed by the users is confidential. Although PPC requires application programmers to identify the origins of confidential data, it enables efficient detection and safe exclusion of all memory locations that may store confidential data. In addition, PPC clears confidential data only in the checkpoint file, not in the DRAM, and hence will not affect the current execution of the application.

The rest of the paper is organized as follows. Section III provides an overview of our application-level approach. Section IV presents our application programmer interface and our information flow analysis technique. Section V presents performance evaluation. Section VI compares our work with related research and Section VII concludes the paper.

II. Assumptions and Threat Model

We assume that the host system, the hypervisor, and the VM on which PPC runs, are not compromised prior to or during checkpointing. After the VM has been checkpointed or restored, the attacker may assume full control over the host system, the VM, and the checkpointer. The attacker may scan the contents of the checkpoint files with the goal of obtaining confidential data. The attacker may also copy checkpoints stored in the computer to a remote site for later analysis. In addition, the attacker can restore the VM from the checkpoint and modify the checkpoint.

III. Overview of Our Approach

The primary goal of PPC is to provide application developers with greater control over specifying the origins of confidential data. This allows the hypervisor-level checkpointing mechanism to perform a more informed and finer-grained exclusion of confidential information from the checkpoint file without resorting to brute-force approaches such as terminating entire user-level processes. It also allows application developers to maintain application stability after a VM is restored, when memory locations that previously contained confidential data no longer do so.

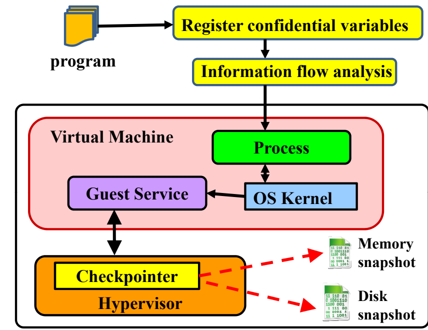

Fig. 1. The architecture of Privacy-preserving Checkpointing system.

Figure 1 gives the architecture of PPC. First, the application programmer registers variables storing confidential inputs from users (without considering the propagation of inputs). Next, an information flow analysis technique is applied, which tracks how the confidential data propagates and automatically registers all memory locations that may contain the data. When the VM checkpointing is performed, a special process inside the VM called "guest service" collects a list of logical addresses of memory locations storing the registered variables, converts them to physical memory addresses, and sends them to the checkpointer in the hypervisor. The checkpointer then excludes the corresponding memory locations from being checkpointed (i.e., contents of such locations are zeroed out).

VM restoration is handled as follows. All confidential memory locations in the checkpoint are marked inaccessible upon VM checkpointing; such locations will be made accessible again after checkpointing is done in order not to affect the current execution of applications. If a confidential memory location is accessed by an application after VM restoration, a SIGSEGV signal will be sent to the application. The application developer can write a signal handler which redirects the execution of the application to a desirable state (as dictated by application semantics) upon receiving the signal. This might include resetting all the internal application state (such as data structures and the execution flow) that might be related to the processing of confidential data. For example, if the program has registered the password entered by the user, then when the password is accessed during restoration, the program will be redirected to the place where the user is prompted to re-enter the password.

IV. Privacy-Preserving Virtual Machine Checkpointing

This section presents our privacy-preserving checkpointing (PPC) mechanism, which identifies memory locations that store users' confidential data such as credit card numbers and passwords. Such memory locations will be cleared in the checkpoint file when the VM is checkpointed. Note that PPC does not affect the current execution of the application

int register(unsigned long addr, unsigned int len): register user-space memory $[addr, addr + len]$.

int unregister(unsigned long addr, unsigned int len): unregister user-space memory $[addr, addr + len]$.

int kregister(void *kaddr, unsigned int len): register kernel memory $[addr, addr + len]$.

int kunregister(unsigned long addr, unsigned int len): unregister kernel memory $[addr, addr + len]$.

int isregister(unsigned long addr, unsigned int len): check if memory $[addr, addr + leng]$ is registered.

void lockreg()/unlockreg(): disable/enable checkpointing.

Fig. 2. Application programmer interface for registering confidential data.

because it clears only specific locations in the checkpoint file, not in the DRAM that the application is currently using.

A. Identifying and Tracking Confidential data

An application developer uses the PPC application programmer interface (API) shown in Figure 2 to specify the origins of confidential data. PPC then performs information flow analysis to track memory locations that contain confidential data.

The API $register$ is used to register variables that store user's confidential inputs; $addr$ and $length$ specify the starting address and the size of memory locations storing such variables, respectively. The API $unregister$ is used to unregister variables and $isregistered$ checks if a memory location contains confidential data.

To prevent a VM from being checkpointed after a variable is assigned confidential data and before the variable is registered, PPC provides the API $lockreg()$ and $unlockreg()$ that store 1 and 0, respectively, in the pseudo filesystem interface $/proc/pid/lock$ where pid is the process ID of the program. When checkpointing is initialized, the guest service will check the value stored in $/proc/pid/lock$ and inform the checkpointer. If the value is 0, then the VM is checkpointed. Otherwise, the guest service periodically checks the value and informs the checkpointer when the value becomes 0.

An application may interact with the guest kernel by invoking system calls and exchanging data for I/O operations, and hence the confidential data may reside in kernel's memory. PPC provides kernel-level interfaces $kregister$ and $kunregister$ to identify and track confidential data within the kernel space. Further, if one simply clears the confidential data from kernel memory locations in the checkpoint file, the kernel may crash after the VM is restored. To ensure that the guest kernel can resume safely after a VM's restoration, when the checkpointing is initiated, the kernel temporarily pauses new system calls and I/O requests from the application and completes (i.e. flushes) any pending I/O operations such as disk I/O, network packets, display buffers, etc. The kernel then zeroes out all the I/O buffers that store confidential data to prevent data leakage through buffer reuse. Once the kernel memory is sanitized of application's confidential data, then the VM checkpointing operation is allowed to proceed.

B. Tracking the Propagation of the Confidential Data

The APIs presented in Section III are used to register only the origins of confidential data. The confidential data may also be propagated to other variables during normal processing by the application, through e.g. assignments, parameter passing, etc.. This section presents an information flow analysis technique, which tracks the propagation of confidential data and automatically registers all memory locations storing the confidential data. Although our implementation targets the C source code, our technique can be adapted to other programming languages by porting the API for registering confidential memory locations and transforming the source code. Runtime components such as tracking confidential data locations and excluding them during checkpointing do not need to be re-implemented for a new language. For clarity of the presentation, we first use a subset of C language to illustrate our approach and then discuss how to extend it to handle pointers, structures, arrays, and library functions.

Let P denote a program, Ds a sequence of declarations, Fs a sequence of functions, Ss a sequence of statements, D a declaration, F a function, and S a statement. Also, let $x, x_1, \ldots$ and $y, y_1, \ldots$ range over variables, and f range over function names. The syntax of the program is given below.

$$
\begin{array}{rcl}
P & ::= & Ds; Fs \\
Ds & ::= & D; \mid D; Ds \\
D & ::= & Tx_1, \ldots, x_n \\
T & ::= & int \mid float \mid char \\
Fs & ::= & F; \mid F; Fs \\
F & ::= & \text{f}(x_1, \ldots, x_n)\{Ds; Ss\} \\
Ss & ::= & S; \mid S; Ss \\
S & ::= & x = y \mid \text{f}(y_1, \ldots, y_n) \mid \text{register}(x) \\
 & & \mid \text{ if } (C)\{Ss\}\text{else}\{Ss\} \mid \text{return}x
\end{array}
$$

The key idea behind our information flow analysis technique is to insert function statements and their implementations into the application source code in order to keep track of the scope of confidential variables registered by the user. The transformed program will then be executed and all memory locations containing confidential data will be registered during execution. Algorithm 1 gives the transformation process. Below, we explain the algorithm in detail.

First, we insert declarations for two global variables $stack$ and $table$ (Line 2), which represent a stack and a symbol table, respectively. The stack is used to keep track of the scope of variables, and the symbol table is used to store memory locations and their confidential flags. The confidential flag of a memory location is either 0 (not confidential) or 1 (confidential). Each symbol table entry has the form $(loc(var), sizeof(var), 0/1)$, where $loc(var)$ is the memory

Algorithm 1 Transformation Algorithm

```
1: procedure PROC_PROG(prog)
2:    print "symtable * stack = null;  symtable table;"
3:    gvar = a set of all global variables in prog;
4:    if gvar ≠ ∅ then
5:        print declarations of global variables;
6:        print "symtablegtable = const_gvar(gvar);" endif
7:    for every function definition fdef{S} of the prog
8:        PROC_FBODY(fdef{S});   end for
9: end procedure
10: procedure PROC_FBODY((fdef{S}))
11:    print "fdef{";
12:    fpara = a set of all parameters in fdef;
13:    if fdef is main then
14:        print "table = const_table();"
15:        print "push(); add_param_main();"
16:    else print "replace_param(fpara);"   end if
17:    for every stmt ∈ S do PROC_STMT(stmt);   end for
18:    if fdef is main then print "pop();" end if
19: end procedure
20: procedure PROC_STMT(stmt)
21:    if stmt is "Tx_1, ..., x_n;" then
22:        print stmt;
23:        for all x_i do
24:            print "add_local(&x_i, sizeof(x_i));" end for
25:    end if
26:    if stmt is "register(x)" then print stmt; end if
27:    if stmt is "x = y" then
28:        if y is a constant then print stmt;
29:        else print "assign(&x, y, &y);" end if
30:    end if
31:    if stmt is if C {Ss_1} else {Ss_2} then
32:        print "if C{";   PROC_STMT(Ss_1);
33:        print "} else {";   PROC_STMT(Ss_2);
34:        print "}"; end if
35:    if stmt is "f(y_1, ..., y_n)" then
36:        PROC_FUNC(f(y_1, ..., y_n)); end if
37:    if stmt is "x = f(y_1, ..., y_n)" then
38:        PROC_FUNC(f(y_1, ..., y_n));
39:        print "get_return_mark(x);" end if
40:    if stmt is "return val"
41:        print stmt;   print "set_return_mark(val);" end if
42: end procedure
43: procedure PROC_FUNC(()funcname)
44:    print "symtabletable = constr_table();"
45:    print "push();"
46:    for all arg ∈ function arguments do
47:        print "add_param(&arg, sizeof(arg));"
48:    end for
49:    print funcname;   print "pop();"
50: end procedure
```

address of variable var, $sizeof(var)$ is the size of var, and 0/1 is the confidential flag of var.

Prior to the invocation of a function, we insert statements $const_table()$ an $push()$, which construct a symbol table and push the table to $stack$, respectively (Lines 14-15, 44-45). When a function exits, we insert a statement $pop()$, which removes the topmost symbol table (i.e., the symbol table associated with the function) from the stack, and clears and deallocates all confidential memory locations whose addresses are stored in the table. This step is necessary since the deallocated confidential memory may be allocated again, and if reallocated, such memory locations should not be cleared during checkpointing. Declarations of global variables are handled through $const_gvar(gvar)$, which constructs a symbol table storing global variables and their confidential flags.

The function $PROC_FBODY$ in Algorithm 1 processes function definitions. The function $add_param_main()$ adds memory locations and confidential flags of arguments of the main function to the symbol table (Line 15). Confidential flags of real parameters of a function are assigned to the corresponding formal parameters as follows. Prior to the invocation of function $f(y_1, \ldots, y_n)$, we insert $add_param()$, which adds all y_is and their confidential flags to the symbol table, in the order of $y_1, \ldots, y_n$ (Lines 44–48). At the beginning of the definition of function f, we insert statement $replace_param(fpara)$, which replaces memory locations of real parameters in the symbol table with memory locations of the corresponding formal parameters $fpara$ (Line 16).

Statements are transformed as follows. If the statement is a variable declaration $Tx_1, \ldots, x_n$;, then for every x_i, we insert a statement $add_local(\&x_i, sizeof(x_i))$, which adds $(\&x_i, sizeof(x_i), 0)$ to the symbol table (Lines 23–24). Statement $register(x)$ is not transformed, which replaces $(\&x, size(x), 0)$ in the symbol table with $(\&x, size(x), 1)$ (Line 26). The assignment statement $x = y$ is transformed as follows: if y is a constant, then the statement is not transformed since y is not confidential (Line 28). Otherwise, we replace $x = y$ with $assign(\&x, y, \&y)$ (Line 29), which works as follows. First, we disable checkpointing and check if the confidential flag of $\&y$ is 0. If so, we locate $\&x$ in the symbol table and set its confidential flag 0. If y's confidential flag is 1, then we set the confidential flag of $\&x$ to 1. Finally, we enable checkpointing. The conditional statement if (C) $\{Ss_1\}$ else $\{Ss_2\}$ is handled by recursively transforming Ss_1 and Ss_2 (Lines 31–34).

The value returned from a function is communicated from the function to its caller as follows. If the statement is "return val", we insert a statement $set_return_mark(val)$ which adds the memory location and the confidential flag of the return value to the end of the symbol table (Lines 40–41). When $x = f(y_1, \ldots, y_n)$ is processed, we add a statement $get_return_mark(x)$, which replaces the variable in the last entry of the symbol table with x (Lines 37–39).

Example: Consider the C program in Figure 3(a). The transformed program is given in Figure 3(b). The bold color highlights the differences between the original program and the transformed program.

C. Handling Pointers, Structures, Arrays, System Calls, and Library Functions

Pointers and arrays: Pointer declaration $T * p$ is handled by inserting a statement after the declaration, which adds $(\&p, size(p), 0)$ to the symbol table. When "malloc" is called to allocate a memory location referred to by a pointer p, we insert a statement that adds $(p, size(*p), 0)$ to the symbol table. Pointer assignment $p = p_1$ is not transformed since p_1 stores a memory address and address is not confidential. The statement $x = *p$ is transformed to $assign(x, *p, \&(*p))$ and

```
main(int argc, char **argv){
   int v;  int x;
   v = atoi(argv[1]);
   register(v);
   x = atoi(argv[2]);
   if (x==1) { v1 = v; }
   else { v1 = 6; }
   v2 = v1;  f(v2);
}

f(int v){
     …
}
```
(a)

```
symtable *stack = null; symtable table;
main(int argc, char **argv){
     table = constr_table();  push();
     add_parame_main();
     int v;  add_local(&v, sizeof(v));
     int x; add_local(&x, sizeof(x));
     v = atoi(argv[1]);  register(v);
     x = atoi(argv[2]);
     if (x==1) { assign(&v1, v, &v); } else {v1 = 6;}
     assign(&v2, v1, &v1);
     table = constr_table(); push();
     add_param(&v2, sizeof(v2));
     f(v2); pop(); pop();
}
f(int v){replace_param({&v}); … }
```
(b)

Fig. 3. An example for illustrating the proposed information flow technique.

$*p = x$ is transformed to $assign(\&(*p), x, \&x)$. When "free" is called, the corresponding memory location will be cleared.

When pointers are passed as parameters to functions, operations on pointers in one function can affect the data outside the function. As a result, the memory location pointed to by a pointer defined in a function f may be stored in the symbol table storing global variables, the symbol table of f, and the symbol tables of functions that (directly or indirectly) call f. For example, consider the following program.

$$f()\{char * p, p_1; p = (char*)malloc(sizeof(char));$$
$$f_1(p); p_1 = p; \};$$
$$f_1(char{*}p_2)\{*p_2 =< confidentialdata >; register(*p_2); \}$$

Since $*p$, $*p_1$ and $*p_2$ refer to the same memory location, when $*p_2$ is assigned confidential data in f_1, $*p$ and $*p_1$ also become confidential. As a result, when $register(*p_2)$ is called, we need to update the confidential flag associated with the address of $*p_2$, which is stored in the symbol table of f. Arrays are handled similarly.

Structures: When a structure s is declared, we insert a statement which adds $(\&s, size(s), 0)$ to the symbol table. Assignment of a variable x to a field $s.field$ of a structure s is transformed to $assign(\&s.field, x, \&x)$. Assignment of a field $s.field$ to a variable x is transformed to $assign(\&x, s.field, \&s.field)$. Assignment of a structure s to another structure s_1 is transformed into a sequence of assignments; each assigns one field of s to the same field of s_1. Assignment of the nested structure is transformed recursively.

System calls: We implement wrappers for the system calls that may process confidential data. For example, the system call $write$ is used to write data stored in a buffer into a file. Since $write$ may be used to write confidential data, we define a new system call $confidential_write()$, which checks if the variable written contains confidential data, and if so, registers the variable. When $write$ is called, the execution is redirected to $confidential_write()$.

Library functions: If the source code for library functions is available, then library functions can be handled in a way similar to user-defined functions. Handling cases where the source code of library functions is not available is a topic of our future work.

Algorithm 2 The optimization.

1: $gvset = \emptyset$; $fset = \emptyset$; $Reach = Reach_1 = \emptyset$;
2: **procedure** PROC_PROG_OPT($prog$)
3: $G = const_callgraph(prog)$;
4: **for all** $register(v)$ in $prog$ **do**
5: **if** v is a global variable **then**
6: $gvset = gvset \cup \{v\}$;
7: $fset = fset \cup \{f \mid f$ contains $v\}$;
8: **else**
9: $fset = fset \cup \{f \mid f$ contains $register(v)\}$;
10: **end if**
11: **for all** $f \in fset$ **do**
12: $Reach = Reach \cup \{(f, \emptyset)\}$; $infoflow(f, \emptyset, \emptyset)$;
13: **end for**
14: **end for**
15: **end procedure**
16: **procedure** INFOFLOW($f, plist, callee$)
17: $lvset = \{$the ith parameter of $f \mid i \in plist\}$;
18: **for all** statement $stmt$ in f **do**
19: **if** $stmt$ is "$register(v)$" **then**
20: $lvset = lvset \cup \{v\}$; **end if**
21: **if** $stmt$ is "$x = y$" and $y \in (gvset \cup lvset)$ **then**
22: replace $x = y$ with $assign(\&x, y, \&y)$;
23: **if** x is a global variable **then** $gvset = gvset \cup \{x\}$;
24: **for every** function f containing x
25: **if** $(f, _) \notin Reach$ **then**
26: $Reach = Reach \cup (f, \emptyset)$;
27: $fset = fset \cup \{f\}$; $infoflow(f, \emptyset, \emptyset)$;
28: **end if** **end for**
29: **else** $lvset = lvset \cup \{x\}$; **end if**
30: **end if**
31: **if** $stmt$ is "$f_1(x_1, \ldots, x_n)$" **then**
32: $plist_1 = \{i \mid x_i \in (gvset \cup lvset)\}$;
33: **if** ($plist_1 \neq \emptyset$ and $(f_1, plist_1) \notin Reach$) **then**
34: $Reach = Reach \cup (f_1, plist_1)$;
35: $fset = fset \cup \{f_1\}$;
36: $infoflow(f_1, plist_1, \emptyset)$; **end if**
37: **end if**
38: **if** $stmt$ is "$v = f_1(x_1, \ldots, x_n)$" **then**
39: **if** ($f_1 \in callee$) **then** $lvset = lvset \cup \{v\}$;
40: **else**
41: $plist_1 = \{i \mid x_i \in (gvset \cup lvset)\}$;
42: **if** ($plist_1 \neq \emptyset$ and $(f_1, plist_1) \notin Reach$) **then**
43: $Reach = Reach \cup (f_1, plist_1)$;
44: $fset = fset \cup \{f_1\}$;
45: $infoflow(f_1, plist_1, \emptyset)$; **end if** **end if**
46: **end if**
47: **if** $stmt$ is "return v" **then**
48: **if** ($v \in (gvset \cup lvset)$) **then**
49: **for all** function f_2 that calls f **do**
50: **if** ($(f_2, f) \notin Reach_1$) **then**
51: $Reach_1 = Reach_1 \cup (f_2, f)$;
52: $fset = fset \cup \{f_2\}$;
53: $infoflow(f_2, \emptyset, \{f\})$; **end if** **end for**
54: **end if**
55: **end if**
56: **end for**
57: **end procedure**

D. Optimization

Applying Algorithm 1 to all functions or transforming all assignment statements to $assign$ may lead to high runtime overhead. This section presents an optimization to reduce the runtime overhead of the algorithm. A slight increase in the static transformation time due to this optimization is acceptable because an application is usually installed once and executed many times. The optimization consists of the following steps.

Step 1: Prior to static transformation, we check if any variable in the program is registered, which can be done in polynomial time. If not, then no transformation is performed. Otherwise, we perform Step 2.

Step 2: We perform a coarse-grained static analysis to overestimate a set $fset$ of functions to which the confidential data may propagate. Algorithm 1 will be applied to only such functions. We also overestimate a set $gvset$ of global variables and a set $lvset$ of local variables that may contain confidential data. Assignment of the form $x = y$, where $y \notin gvset \cup lvset$, will not be transformed to $assign(\&x, y, \&y)$, since y is not confidential. The pseudocode of Step 2 is given in Algorithm 2. For clarity of presentation, in the pseudocode, we assume that different variables are represented using different names. Cases where different variables may be represented using the same name can be handled using a symbol table stack. The algorithm works as follows.

First, we compute a call dependency graph, in which each node is a function name and each edge $f_1 \rightarrow f_2$ specifies that f_1 calls f_2 (function $const_callgraph(prog)$ in Line 3). Next, we assign names of all functions that contain registered variables, to $fset$ (Lines 4–14). We then process every function f in $fset$ and all functions called by f (function $infoflow$). If the return value of f may be confidential, then we also process functions that call f (Lines 47–55). As an example, consider the following program.

```
main(){int v; v = 1; f(v);}
f(int x){int y; x =1; y = <user's input>; register(y); f1(x,y);}
f1(int z, int u){int v, w; v= z; w=u, f2(v);}
f2(int k){......}
```

With the above optimization, we only need to transform f and f_1. All assignment statements except $w = u$ will not be transformed to $assign$.

E. Accounting For Inter-Process Communication

In cases where multiple processes may communicate confidential data to each other, each process would use the PPC API to register the origins of the confidential data. PPC then performs information flow analysis to identify tainted memory locations. Data in these locations would then be excluded by the checkpointer during VM checkpointing. Our current implementation assists multi-process applications in identifying two types of inter-process data dependencies (although PPC can be effectively used even without this mechanism). First, to identify the communication of confidential data through files, PPC inserts a statement prior to each file write operation to check if any confidential data is written to a file, and if so, registers the file itself. Prior to any file read operation, PPC checks if the file is registered, and if so, registers the variable storing the data retrieved from the file.

For co-located processes that communicate confidential data through sockets, PPC applies a similar approach. Before the data is written to a socket, PPC checks if the memory location storing the data is registered. If so, it registers the socket endpoints storing the data. When a peer process receives data through a socket, PPC checks if the corresponding socket endpoint is registered and if so, registers the variable to which the data retrieved. In addition, PPC clears the socket buffer storing the confidential data after the data is transmitted. Note that our approach does not prevent attackers from sniffing the network to obtain the confidential data. To protect the confidential data transmitted over the network, the programmer can encrypt the data. We are currently extending PPC to assist in the detection data dependencies through other means of inter-process communication.

V. Prototype Development, Experiences, and Experimental Results

To demonstrate the feasibility of our approach, we have implemented a prototype of PPC in the VirtualBox platform. The prototype was implemented on top of PYCparser, a C parser written in Python [15]. We have applied PPC to the `vim` and `gedit` applications besides using our own benchmarks for measurements. Although our techniques are presented in the context of VirtualBox memory checkpointing, they can be adapted to other virtualization platforms.

A. Experiments

Registering confidential data printed on terminals: We wrote a program that keeps printing a string on the terminal console (also called virtual terminal) inside a VM. The VM was checkpointed while the program was printing the string. Without applying our PPC technique, the VM checkpoint file contains six clear-text occurrences of the string. The string appeared in the program memory as well as the TTY buffer of the virtual console. In addition, the string displayed on the virtual console was stored in a memory mapped frame buffer region, but not as clear-text. After applying our PPC technique, the checkpoint did not contain clear-text occurrences of the string, and after restoration, the strings printed to the virtual console both before checkpointing and after restoration were cleared.

Registering confidential data sent through socket: We wrote a simple client-server program, in which the client connects to the server and sends a string to the server. The string appeared in the process memory as well as the socket buffer in the kernel. After applying our PPC technique to register the string, the checkpoint does not contain clear-text occurrences of the string.

Registering users' input in `vim` *text editor:* In this experiment, we started the `vim` application (version 7.2) in a virtual console, typed a string into `vim`, and checkpointed the VM. We found eight clear-text occurrences of the string in the checkpoint. After applying our PPC technique, the checkpoint does not contain any clear-text occurrence of the string, and the string is cleared when the VM is restored.

We found that, in `vim`, the user's input string may appear in the TTY buffer as well as the `vim` application. In order to identify all memory locations storing the string, the programmer needs to register two variables: $inbuf$ in function $fill_input_buf(exit_on_error)$ and c in function $edit()$, which store the input from the user. We then apply our PPC technique to register all memory locations that may store the user's input.

There are more than 5000 functions in `vim`. Algorithm 2 identified around 2000 functions to which the confidential string may propagate. We observed that, in the transformed program, many variables registered in `vim` are global variables. We also observed that, with optimization in Algorithm 2, most assignments are not transformed to $assign$. In addition, the confidential string is passed through pointer parameters from callees to callers, through parameter passing from callers to callees, through return values, and through library functions such as $memcpy$.

Registering users' input in `gedit`*:* We performed the same experiment on `gedit`, as we did for `vim`. During our experiment, we found that the user's input string may appear in the `gedit` source code, the GTK library, and the frame buffer. Before applying our PPC technique, the checkpoint contains four clear-text occurrences of the string. After applying PPC , the checkpoint does not contain the string.

B. PPC overhead

We evaluated the overhead of our PPC prototype implementation on a 2.5 GHz Pentium machine with 4 GB RAM running Linux 2.6.28. The overhead of information flow analysis is determined by the number of functions transformed and the number of statements added to each function. We wrote a script that generates a sequence of programs, each of which contains one $constr_table$, pop, and $push$ statement, and multiple add_local and $assign$ statements that vary from 1 to 481. For each experiment, we computed an average of 100 runs. The overhead varied from 0.85ms for one statement to 1.9ms for 481 statements. Other statements inserted involve similar operations as the above statements, such as adding variables to the symbol table and looking up variables on the symbol table, and hence have similar overhead. We also compared the relative costs of using $write$ and $confidential_write$ (discussed in Section IV) through a program that prints strings of the various sizes to the terminal 100 times. We found that $confidential_write$ does not impose observable overhead compared to $write$.

VI. Related Work

Previous work on minimizing data lifetime has focused on clearing the deallocated memory (also known as memory scrubbing) [16], [17]. However, memory scrubbing does not solve the problem of confidential data being checkpointed *before* the pages are scrubbed. Garfinkel et al. [18] developed a hypervisor-based trusted computing platform, whose privacy features include encrypted disks and the use of a secure counter to protect against file system rollback attacks. Encrypting the checkpoint has also been recommended in [19]. However, encrypting the checkpoint is not suitable to protect confidential data that should be quickly discarded after its use because (1) it does not reduce the lifetime of confidential data; (2) when the VM is restored, the checkpoint will be decrypted and loaded into the memory of the VM, thus exposing the confidential data again; (3) the checkpoint may be shared among multiple users (e.g. among multiple programmers for the purpose of software development), thus increasing the likelihood of data leakage. It is also insufficient to encrypt just the memory containing the confidential data because VM can be checkpointed just when the data is decrypted in memory.

Our prior work [11] excluded the entire process and its memory footprints from the VM checkpoint to preserve user privacy. However, this results in the termination of process after restoration and hence is not suitable if the user wants the process to remain alive after restoration. Our current work provides developers with a finer-grained control over excluding confidential data while maintaining system stability.

A number of researchers have developed static or dynamic information flow analysis (also called taint analysis) techniques [20], [21] for detecting attacks [22], [23], [24], [25], preventing leakage of confidential data [26], [27], [28], and performing malware analysis [29], [30]. Static information flow analysis [31], [26], [27] is not suitable for our work since it may over-estimate or under-estimate memory locations that store confidential data. Dynamic information flow analysis can potentially be applied to identify all memory locations storing the confidential data. However, traditional memory taint analysis systems [12], [13], [25], [14] have a high performance overhead for large applications due to the use of binary emulation or interpretation.

Xu et al. [32] presented a dynamic information flow analysis technique for detecting various attacks. Lam et al. [33] developed a compiler that inserts rules in programs to keep track of the tainted data, and developed mechanisms for selective sandboxing of network client applications based on whether their inputs are tainted or not. Besides the objective being different, the above two approaches aren't suitable in the context of VM checkpointing for the following reasons. (1) we need to identify memory locations in the kernel (e.g. the socket buffer), the frame buffer, X-Windows, and the terminal, which store confidential/tainted data since such memory locations in a VM are also checkpointed. (2) we need to ensure that checkpointing won't be performed after a variable is assigned confidential data and before the variable is registered. (3) we need to clear the confidential memory locations from the checkpoint file during checkpointing. As a result, its not feasible to use shadow variables in their approaches to store the confidential flag. (4) we need to clear deallocated memory locations storing confidential data, while the above approaches do not need to do so. Consequently, the approach in [32] that uses a global array to store the confidential memory

location, cannot be applied to VM checkpointing. (5) we develop techniques to ensure the stability of the VM after the VM is restored, while the above approaches do not need to do so. (6) [32] did not consider structures whereas the mechanism in [33] for handling structures is more coarse-grained than ours. (7) our optimization is stronger in that we not only identify a set of functions but also a set of variables to which the confidential data may propagate; only assignments involving such variables are transformed to $assign$. Finally, some approaches rely on hardware support [34], [35] for taint-tracking and hence cannot be applied to existing systems.

VII. Conclusion and Future Work

This paper presents a Privacy-preserving Checkpointing (PPC) system that enables VM checkpointing mechanism to safely identify and exclude memory locations that contain confidential user data. PPC provides an interface for application programmers to identify the origins of confidential data in memory. Before a VM is checkpointed, PPC performs information flow analysis that automatically tracks the flow of confidential data through both the application's user space and the guest operating system's kernel memory. During checkpointing, all the tracked memory locations are safely excluded from being stored in the checkpoint file. PPC also informs applications right after VM restoration to allow applications to transition to a "safe" state before they restart execution. As future work, we plan to evaluate our approach using more real-world applications, such as web applications, and extend our techniques to automatically account for inter-process communication. We also plan to develop techniques to protect users' confidential input when source code of the application is not available.

Acknowledgement: This work is supported in part by the National Science Foundation through grants CNS-0845832 and CNS-0855204.

References

[1] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "Mavmm: Lightweight and purpose built vmm for malware analysis," in *Annual Computer Security Applications Conference*, 2009, pp. 441–450.

[2] D. A. S. d. Oliveira and S. F. Wu, "Protecting kernel code and data with a virtualization-aware collaborative operating system," in *Annual Computer Security Applications Conference*, 2009, pp. 451–460.

[3] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *the 11th international symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1–20.

[4] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *15th ACM conference on Computer and communications security*, 2008, pp. 51–62.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," in *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002, pp. 211–224.

[6] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium, pages*, 2003, pp. 191 – 206.

[7] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *ACM symposium on Operating systems principles*, 2005, pp. 91–104.

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of Twenty-First ACM SIGOPS symposium on Operating Systems Principles*, 2007, pp. 335–350.

[9] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *IEEE Symposium on Security and Privacy*, 2008, pp. 233 – 247.

[10] K. Kourai and S. Chiba, "Hyperspector: Virtual distributed monitoring environments for secure intrusion detection," in *ACM/USENIX Conference on Virtual Execution Environments*, 2005, pp. 197 – 207.

[11] M. I. Gofman, R. Luo, P. Yang, and K. Gopalan, "SPARC: A security and privacy aware virtual machine checkpointing mechanism," in *ACM Workshop on Privacy in the Electronic Society*, 2011, pp. 115–124.

[12] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *the 13th conference on USENIX Security Symposium*, 2004, pp. 22–22.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, "Practical taint-based protection using demand emulation," in *EuroSys*, 2006.

[14] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, 2011.

[15] E. Bendersky, "pycparser, http://pycparser.googlecode.com/hg/readme.html."

[16] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum, "Data lifetime is a systems problem," in *ACM SIGOPS European workshop*, 2004.

[17] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: reducing data lifetime through secure deallocation," in *Proceedings of the USENIX Security Symposium*, 2005, pp. 22–22.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *SOSP*. ACM Press, 2003, pp. 193–206.

[19] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: security challenges in virtual machine based computing environments," in *Proceedings of the 10th conference on Hot Topics in Operating Systems*, 2005, pp. 20–20.

[20] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Tech. Rep., 1973.

[21] D. E. Denning and P. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20(7), pp. 504 – 513, 1977.

[22] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *ACM Symposium on Operating System Principles*, 2005.

[23] J. Crandall, Z. Su, S. F. Wu, and F. Chong, "On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits," in *Computer and Communications Security*, 2005.

[24] G. E. Suh, J. Lee, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[25] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[26] D. Volpano, G. Smith, and C. Irvine, "Sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4(3), pp. 167–187, 1996.

[27] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proceeding of ACM Symposium on Operating System Principles*, 1997, pp. 129–142.

[28] P. Yang, Z. Yang, and S. Lu, "Formal modelling and analysis of scientific workflows using hierarchical state machines," in *Workshop on Scientific Workflows and Business Workflow Standards in e-Science*, 2007.

[29] C. K. Ulrich Bayer, Andreas Moser and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, pp. 66–77, 2006.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Computer and Communications Security*, 2007.

[31] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, vol. 21(1), pp. 5–19, 2003.

[32] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks," in *the 15th conference on USENIX Security Symposium*, 2006.

[33] L. C. Lam and T. Chiueh, "A general dynamic information flow tracking framework for security applications," in *the 22nd Annual Computer Security Applications Conference*, 2006, pp. 463–472.

[34] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *the 34th International Symposium on Computer Architecture*, 2007, pp. 482–493.

[35] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "Sift: Low-complexity energy-efficient information flow tracking on smt processors," *Accepted, IEEE Transactions on Computers*.