RollSec: Automatically Secure Software State against General Rollback

Weiqi Dai[1], Yukun Du[1], Weizhong Qiang[1], Deqing Zou[1], Shouhuai Xu[2], Zhongze Liu[1], and Hai Jin[1]

[1] School of Computer Science and Technology,
Huazhong University of Science and Technology, Wuhan, 430074, China
wzqiang@hust.edu.cn,
[2] Department of Computer Science,
University of Texas at San Antonio, San Antonio, TX 78249, USA

Abstract. The rollback mechanism is critical in crash recovery and debugging, but its security problems have not been adequately addressed. This is justified by the fact that existing solutions always require modifications on target software or only work for specific scenarios. As a consequence, rollback is either neglected or restricted or prohibited in existing systems. In this paper, we systematically characterize security threats of rollback as abnormal states of non-deterministic variables and resumed program points caused by rollback, which can generally apply to other scenarios of rollback problems. Based on this, we propose RollSec (for Rollback Security), which provides general measurements including state *extracting*, *recording* and *compensating*, to maintain correctness of these abnormal states for eliminating rollback threats. RollSec can automatically extract these states based on language-independent information of software as protection targets, which will be monitored during run-time, and compensated to correct states on each rollback without requiring extra modifications nor supports of specific architectures. At last, we implement a prototype of RollSec to verify its effectiveness, and conduct performance evaluations which demonstrate that only acceptable overhead is introduced.

Keywords: rollback security, general rollback problem, automated protection, non-deterministic state

1 Introduction

Rollback mechanism can directly resume previous states of applications or virtual machines (VM), which is widely used in system recovery[1–4], software debugging[5–7], and so on [8]. However, those resumed states can be stale and cause serious security problems, e.g., resuming expired keys or duplicate nonce, which also happen on specific architectures like vTPM (virtual Trusted Platform Module)[9] and SGX (Intel Software Guard Extensions)[10]. Due to these potential security risks, rollback mechanism is severely hindered in practice.

Unfortunately, some existing security systems like TrustVisor[11], CloudVisor[12] and H-SVM[13] have not taken rollback threats into consideration, while some other systems roughly restrict[14] or prohibit[15] rollback for security concerns. Memoir[16], ICE[17] and Ariadne[18] aim to maintain *state continuity* of target software against rollback through replay strategies, which only work for deterministic software. Moreover, these solutions also require modifications on source codes causing tedious manual work in practice. ROTE[10] and rvTPM[9] focus on rollback problems under specific architectures of Intel SGX and vTPM respectively. However, their solutions are confined to specific architectures, which cannot be migrated to other architectures easily.

In this paper, we made three main contributions as follows. Firstly, by revisiting and analyzing rollback threats in representative scenarios, we demonstrate that abnormal states in software caused by rollback are the root cause of rollback problems, which can generally apply to software in other scenarios. Moreover, we propose a fine-grained classification of *rollback-related states* to identify those abnormal states that can undermine software after rollback as further protection targets, i.e., states of non-deterministic variables and inappropriate program points for resuming. Based on this, we can simplify security problems of general rollback to equivalent problems of abnormal software states instead.

Secondly, we propose the design of RollSec based on virtualization environment, which is an automated state securing solution for software against general rollback problems, including modules of state *extracting*, *recording* and *compensating*. Extracting module provides algorithms based on language-independent information of target software to automatically identify offsets of those rollback-related states in virtual address space as further protection targets. Moreover, recording and compensating modules can monitor historical state transitions of these targets, and correct those abnormal states caused by rollback through directly accessing targets states in corresponding locations during run-time. Both recording and compensating are based on underlying hypervisor, and will not require modifications on software. RollSec also has three strategies for compensating module to ensure the correctness for different types of states, meanwhile avoiding extra security problems.

Lastly, we present the implementation of RollSec based on Xen hypervisor aiming to provide real-time rollback protections for C/C++ based software. All the components of RollSec are running in control domain `Dom0` to prevent itself from rollback threats, while providing protections for target software in user domain `DomU` against security problems of general rollback. Furthermore, we conduct experiments to evaluate its effectiveness and performance, which shows that RollSec is effective and will only introduce acceptable overheads.

The rest of this paper is organized as follows: §2 presents scenarios of rollback problems and analyses. §3 proposes system mode, threat mode and design of RollSec. §4 describes implementation details. §5 shows results of evaluation experiments. §6 introduces related research works. §7 concludes the whole paper.

2 Rollback Scenarios and Analyses

In this section, we will revisit rollback threats in two representative scenarios, and analyze the root cause in perspective of software states.

2.1 Scenario I: State Loss of Counter caused by Rollback

Linux login module `login` provides access for users to sign in under successful authentications. Moreover, `login` module can load `pam_tally` module, which maintains a counter to record failed login attempts, to prevent accounts from attacks with brute-force password guessing.

We take function `tally_check` of `pam_tally` module as an example described in Listing 1.1. Function `tally_check` will load threshold of failed login attempts from configuration file as variable `deny` at line 2. Then, the current number of failed login attempts will be loaded from log file as variable `tally` at line 6, which will be compared to threshold value `deny` at line 9. If threshold has been exceeded which denotes a potential password guessing attack, function `tally_check` will return error code `PAM_AUTH_ERR`, and related account will be locked for a certain period as penalty. Therefore, it's vital for `tally` to maintain its state correctness, and record number of failed login attempts truthfully, since it constructed a security check against password guessing attacks. Otherwise, such secure protection based on `tally` might act unexpectedly and reveal more sensitive information.

```
1  static int tally_check(time_t oldtime, pam_handle_t *pamh, uid_t uid,
       const char *user, struct tally_options *opts) {
2    tally_t deny = opts->deny;
3    tally_t tally = (tally_t)~0L;
4    ......
5    // load tally with the numbers of failed login attempts
6    i = get_tally(pamh, &tally, uid, opts->filename, &TALLY, fsp);
7    ......
8    if( (deny != 0) &&
9      (tally > deny) &&
10     (((opts->ctrl & OPT_DENY_ROOT) || uid))){
11     /* infomation output */
12     ......
13     return PAM_AUTH_ERR;
14   }
15   return PAM_SUCCESS;
16 }
```

Listing 1.1. Checking of failed login attempts in `pam_tally` module

Figure 1 describes a rollback scenario for `tally_pam` module, where `login` module received requests with wrong password for three times during $t_1 \sim t_3$. Meanwhile, variable `tally` increased from zero to three accordingly, which exceeded the threshold. Function `tally_check` returned an error `PAM_AUTH_ERR` and related account will be locked since t_3 to prevent potential attacks with password guessing. However, at t_4 during account lockout duration, rollback can resume previous states of variable `tally` during $t_0 \sim t_3$ as highlighted by the red dashed lines in Figure 1. Since the value of variable `tally` after rollback is less than the threshold, function `tally_check` will return `PAM_SUCCESS` to denote a "legal" checking for password guessing attacks, although the threshold

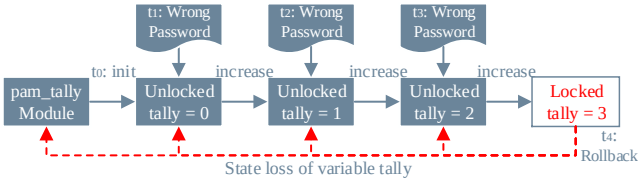


Fig. 1. Rollback impacts on `pam_tally`

has already been exceeded at t_3. Consequently, this secure protection based on variable `tally` in `pam_tally` module acted unexpectedly after rollback, which allows attackers to submit infinite login attempts through repeating rollback.

2.2 Scenario II: Bypassed Authentication Check caused by Rollback

Rollback can also recover program points of previous executions from checkpoint files. However, if the resumed program point is in branches that are determined by security checks, directly resuming such point is equivalent to bypassing these security checks.

```
1  /* deterministic part */
2  ......
3  if(compat20) {
4    /* for SSH 2.0 */
5    ......
6  } else {
7    do_ssh1_kex();
8    do_authentication(authctxt);
9  }
10 /* session configuration */
11 .....
12 /*  do_exec_pty will built both master and slave side for a pty master
       side handles data transmission with client */
13 ret = do_exec_pty(s, command);
```

Listing 1.2. Code snippet of authenticating remote user in `OpenSSH`

We take the code snippet of authenticating remote user in `sshd` daemon of `OpenSSH` as an example described in Listing 1.2. The `sshd` daemon will exchange session key by calling function `do_ssh1_kex` at line 7 after receiving login request from remote client, and then authenticate corresponding user by calling function `do_authentication` at line 8, which will return only if user authentication is successful. After a valid authentication, a remote shell will be established at line 13, through which remote client can access the server.

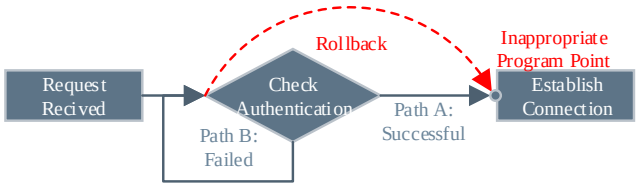


Fig. 2. Example for two executions path of `sshd` daemon

Figure 2 describes two execution paths for the code snippet in Listing 1.2. `Path A` denotes a successful authentication check, while `Path B` denotes a failed one. If rollback resumed a program point as highlighted by the red dashed line in Figure 2, `sshd` daemon will bypass the user authentication process, and establish connection via an expired and untrusted socket.

2.3 Analyses of Rollback Scenarios

Although rollback can happen at multiple levels ranging from thread to VM level along with various architectures, such as SGX, vTPM, only memory and execution states can be recovered to previous ones by rollback according to its definition. We have already demonstrated that stale states of these two kinds can cause unexpected behaviors and undermine security functions in specific Scenario I and II respectively. Since memory and execution states are not confined to specific scenarios, we will extend this conclusion for more general rollback scenarios in this section. Moreover, because software is ultimate victim for rollback problems, we will only discuss the states variables and control flow in software correspondingly.

Based on the general scenario described in Figure 3, we will explain how do these stale states cause unexpected behaviors of software. Since deterministic states can always recover from stale states caused by rollback, only those non-deterministic states can cause security problems after rollback. In Figure 3, `state S` is a state of non-deterministic variable, which is changed at t_0 and t_3. The validity of `state S` will be checked in branch judgment, which results in two different paths (i.e., `Failed Path` during $t_1 \sim t_2$ and `Succeeded Path` during $t_4 \sim t_5$) as depicted in Figure 3. Furthermore, we also locate five program points (i.e., `Point A, B, C, D, E`), which can be resumed by rollback, to show all the possible consequences of rollback.

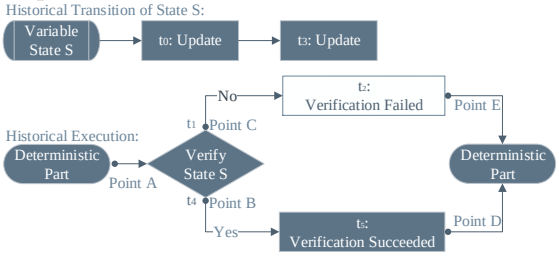


Fig. 3. Abstract scenario of problems caused by rollback

Situation A: When `Point A` is resumed by rollback, control flow of software will execute the branch judgment determined by `state S` next. If `state S` loses its essential transition at t_3, same problems as described in §2.1 will happen. Otherwise, software will not be undermined by rollback.

Situation B: When `Point B` or `C` is resumed by rollback, branch judgment has already been bypassed. The control flow of software is now determined by which program point (`Point B` or `C`) is resumed by rollback instead of `state S`, causing same security problems as described in §2.2.

Situation C: When `Point D` or `E` is resumed by rollback, control flow will execute deterministic part next, which is rollback-irrelevant. Therefore, software will not suffer rollback problems.

Consequently, the problems decried in Scenario I and II can also apply to security problems of general rollback as Situation `A` and `B`, which can demonstrate that the abnormal states of non-deterministic variables and control flow are the root causes of general rollback problems. Hence, we can reduce security problems

of general rollback to two kinds of problems about manipulation of software states through rollback as follows.

Rollback-related variable: Rollback can cause losses of essential state transitions of non-deterministic variables, which makes branch judgments behave unexpectedly causing security problems as described in Situation `A`.

Rollback-related program point: Through directly resuming such program point, the control flow of software may bypass security essential branches resulting in security problems as described in Situation `B`.

3 Platform-Independent Design

In this section, we will present the platform-independent design of RollSec for securing rollback-related states against security problems of general rollback.

3.1 System Model

Our system model is based on virtualization environment, like Xen hypervisor, which mainly includes three components including dependent layer, user and control domain as depicted in Figure 4.

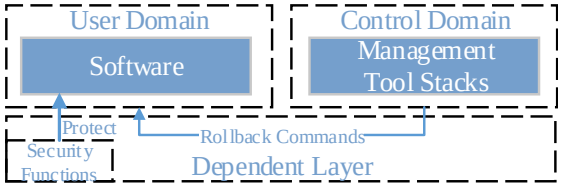


Fig. 4. System model

Dependent layer includes hardware resources and hypervisor. The hardware resource includes all the basic devices, such as storage, CPU and extra secure functions, e.g. TPM [19], TXT [20], SGX [21]. Since we are focusing on security problems caused by rollback on software level, we will not require specific hardware architectures in our system model.

User domain includes all the non-privileged and untrusted parts, like `DomU` in Xen. Software running in user domain can interact with outside world.

Control domain includes privileged parts including all the controlling and managing software tool stacks, like `Dom0` in Xen. The control domain can conduct privileged operations towards user domain, including taking snapshot or rollback/resume of a specific target.

3.2 Threat Model

We assume that both dependent layer and control domain are rollback-free and rollback can only happen in user domain, which is reasonable in current architectures. We also assume that arbitrary rollbacks can happen in user domain at any time, and both illegitimate or legitimate rollback will be taken in to consideration. The legitimate rollbacks are those invoked by administrators,

owners or checkpoint-based recovery strategies[22], while illegitimate rollbacks are those caused by attackers through infiltrating bugs and vulnerabilities in software tool stacks. Whenever rollback happens, software inside user domain may suffer those security problems of rollback as described in §2.

Since other secure problems such as hardware attacks or untrusted hypervisor are orthogonal in this paper, we assume that dependent layer and control domain are trusted and software in user domain can be protected with state-of-the-art protections against known attacks and vulnerabilities excluding rollback problems. Since rollback needs assistance from both dependent layer and control domain, we assume that every rollback event of user domain can be detected.

3.3 Design Goal

Analyses in §2.3 demonstrate that solving security problems for software against general rollback is equivalent to secure states of rollback-related variables and program points of software. Therefore, our design goals can be described as follows.

Goal I: Our solution should identify those rollback-related states in software, and make sure that all the states can maintain their correctness after rollback.

Goal II: Our solution should not be influenced by rollback.

Goal II: Our solution should not require modifications on target software and provide protections against rollback problems without manual involvements.

3.4 Architecture

We propose RollSec, which includes three modules of state *extracting*, *recording* and *compensating*, to secure rollback-related states after rollback based on virtualization environment as depicted in Figure 5.

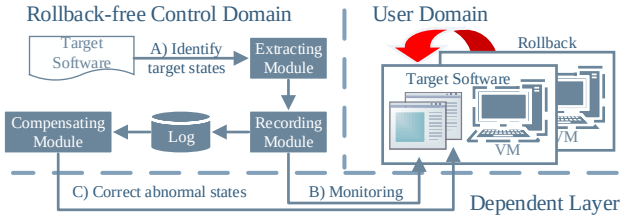


Fig. 5. Architecture of RollSec

1) **Automated extracting module for rollback-related states:** As we characterize the root causes of rollback problems in §2.3, extracting module can identify the information of rollback-related states from software automatically based on syntax, control-flow and data-flow information as step `A` of Figure 5. More details are discussed in §3.5.

2) **Recording module for states of rollback-related variable:** Recording module will only trace state transitions of those rollback-related variables during run-time as step `B` of Figure 5, which can be a guidance for correcting abnormal variable states after rollback in compensating module. We ignore

those rollback-related program points during run-time, since such information can be obtained in extracting module during preliminary phase. More details are discussed in §3.6.

3) **Compensating module for rollback-related states:** Compensating module will correct those abnormal states of rollback-related variables and program points, to remove security problems of general rollback as step `C` of Figure 5. To avoid extra security problems, compensating module also proposed three strategies for different states, which can also be turned off for particular needs, such as debugging to increase flexibility. More details are discussed in §3.7.

3.5 Automated Extracting Module for Rollback-Related States

In this section, we will introduce automated methods for extracting information of those rollback-related states from software based on syntax, control-flow and data-flow information. The control-flow and data-flow information are language-independent, while syntax information is based on a specific programming language. However, syntax information can also be general and language-independent as researches in [23, 24]. Therefore, extracting module described in this section is also general and language-independent.

Automatically extracting rollback-related variables: Based on definitions of rollback-related variables mentioned in §2.3, extracting module will firstly identify those non-deterministic variables in software, and then calculate state transfers among variables to obtain all the rollback-related variables. The detailed extracting processes are described as follows.

Step I. Identify non-deterministic variables via external function calls: According to the analyses in §2.3, rollback-related variables can be regarded as those variables that receive non-deterministic inputs from external function calls, e.g., variable `tally` in `pam_tally` module. There are two ways for external functions to modify variables: 1) If a variable is assigned by return value of the external function, then it can be modified by external function. 2) The external function can also modify variables via influencing corresponding arguments, which are passing-by-pointer, passing-by-reference, or through other mechanisms (e.g., the output parameter in C#). By tracing these external function calls, those non-deterministic variables as rollback-related variables in target software can be identified.

Step II. Calculate state transfers among variables: We defined three kinds of state transfer rules among variables as follows to identify those variables that receives states from those non-deterministic variables obtained in step I. Through calculating these three kinds of transfers iteratively, we can extract all rollback-related variables in target software.

- **Rule #1:** If rollback-related variable `A` is assigned to define variable `B`, then variable `B` should be marked as rollback-related variable.

– **Rule #2:** If variable `B` is assigned to define rollback-related variable `A`, and variable `A` is a pointer, reference, or in other equivalent forms, then variable `B` should be marked as rollback-related variable.
– **Rule #3:** If rollback-related variable `A` is used as condition in branch judgment `C`, then judgment `C` is rollback-related. If variable `B` is modified in branches controlled by branch judgment `C`, then it should be marked as rollback-related variables as well. Moreover, all define statements of variables inside functions, which are called in branches controlled by judgment `C`, should all be marked as rollback-related variables.

```
1  def internalFunc(paramA, *paramB){
2    // Rule #2, equivalent to varD = varA
3    *paramB = paramA
4    // Rule #3, function call inside judgment
5    varF = varX
6    return paramA
7  }
8  ...
9  // assigned by external non-deterministic inputs
10 varA = externalFunc()
11 // Rule #3
12 if varA:
13   // Rule #1, assigned by the return variable
14   varB = internalFunc(varA, &varC)
15   // Rule #1, #3
16   varD = varB
17   // Rule #3, varX isn't rollback-related
18   varE = varX
```

Listing 1.3. Example for state transfer rules

An example of state transfer rules is described in Listing 1.3. Since `varA` is assigned by return value of external function at line 10, it will be marked as rollback-related variable according to step I. Because `varA` is passed to internal function as an argument at line 14, `paramA` will be marked as rollback-related variable according to Rule #1. Since `paramA` is returned by `internalFunc` to assign `varB` at line 14, `varB` will be marked as rollback-related variable according to Rule #1.

Since `varC` is a passing-by-pointer argument passed to `paramB`, and `paramB` is assigned by rollback-related `paramA` at line 3 in Listing 1.3, `paramB` will be marked as rollback-related variable according to Rule #1, and `varC` will also be marked as rollback-related variable according to Rule #2.

Since `if` statement at line 12 in Listing 1.3 is determined by rollback-related variable `varA`, `varD` and `varE` will also be marked as rollback-related variable according to Rule #3. Because `internalFunc` is called at line 14, which is controlled by rollback-related branch at line 12, `varF` will also be marked as rollback-related variable according to Rule #3.

Step III. Extract locations for rollback-related variable: After step I and II, we can get all those rollback-related variables. However, in order to monitor these variables during run-time through underlying hypervisor, corresponding offsets in virtual address space and memory size of those variables should be obtained based on information in executable file, such as `ELF` header. The results of this section will be handed to recording and compensating module for further protections during run-time.

Automatically extracting rollback-related program point: Based on definition of rollback-related program point mentioned in §2.3, extracting module will firstly identify those program points controlled by rollback-related branches. Then, for each program point `V`, extracting module will calculate the corresponding *compensating node* to denote those bypassed rollback branch judgments if point `V` is resumed for further compensating needs. The detailed extracting processes are described as follows.

Step I. Generate rollback-related control flow graph: We will identify those rollback-related branch judgments in control flow graph (CFG) according to Rule #3 in §3.5 at first, and delete other nodes to generate a rollback-related CFG.

Step II. Extract rollback-related program point: If program point `V` in CFG can be inserted to rollback-related CFG of step I, it means that directly this resuming point `V` will bypass corresponding rollback-related branches. Then, program point `V` will be marked as rollback-related according to analyses in §2.3, otherwise we will ignore it, and move to next program point.

Step III. Calculate compensating node for resumed program point: For each rollback-related program point `V` obtained from step II, we will calculate its closest dominator node `D` in rollback-related CFG firstly. By re-executing from this dominator node `D`, all the bypassed rollback-related branches between node `D` and `V` can be compensated to maintain correctness. However, if node `D` is a rollback-related program point, then compensating control flow to node `D` will introduce extra problems as described in §2.2. Therefore, we will skip it to get another closest dominator node. Otherwise, we will mark node `D` as the compensating node to denote those bypassed branch judgments when program point `V` is resumed. At last, both node `V` and `D` will be stored and interpreted to corresponding offsets in code section for further compensating purpose.

3.6 Recording Module for Rollback-Related Variable States

Since the compensating node for each rollback-related program point has been obtained through preliminary offline processes in §3.5, therefore, we only need to trace state transitions of those rollback-related variables during run-time for further compensating purpose. The recording module takes advantages of underlying hypervisor, e.g., virtual machine introspection (VMI), to directed access memory in user domain from control domain, which will not require modifications on target software.

Through dynamically implanting hooks in target software during run-time, recording module can provide real-time monitoring for each state update of those rollback-related variables. A timestamp will also be added for each state transition record, which will be used to decide whether states of those rollback-related variables have been lost during compensating phase. All the historical state transitions will be a guidance for maintaining the correctness of those rollback-related variables after rollback.

3.7 Compensating Module for Rollback-Related States

Compensating module can correct those abnormal states for both rollback-related variables and program points immediately after rollback, through monitoring rollback events in user domain. In this section, we will discuss strategies for compensating these two kinds of rollback-related states in appropriate way without introducing extra security problems.

Strategy for compensating rollback-related variable: We classify variables into two categories, non-random and random variables, and propose corresponding strategies as follows:

- **Compensating non-random variables:** Compensating module will recover the latest states for non-random variables to maintain state continuity against rollback. For example, variable `tally` described in §2.1 will be compensated to its latest state after t_3.
- **Compensating random variables:** This strategy will discard resumed states of random variables after rollback, and randomize variable states again. For example, a used random number seed in `sshd` daemon resumed by rollback can be re-randomized with this strategy, avoiding duplicate nonce.

The two strategies above are adequate for eliminating security problems caused by rollback mentioned in §2. Moreover, compensating module also allows users to flexibly indicate states that should not be compensated for special purpose like debugging.

Strategy for compensating rollback-related program point: If rollback-related program point is resumed by rollback, compensating module must make sure that all those bypassed rollback-related branch judgments are not in a stale state, through adjusting control flow to corresponding compensating point as described in §3.5. Then, software will re-execute all these bypassed branch judgments under correct states.

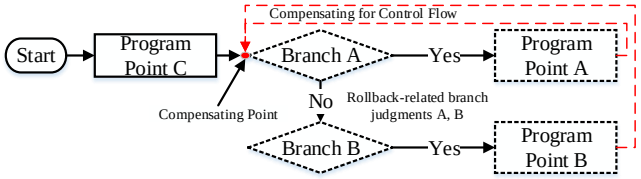


Fig. 6. Example For program flow compensation

Figure 6 shows an example for compensating those rollback-related program points. Directly resuming checkpoint `A` will bypass branch `A`, while resuming checkpoint `B` will bypass both branch `A` and `B`. According to this compensating strategy, when checkpoint `C` is resumed, there is no need for compensating. When checkpoint `A` or `B` is resumed, control flow will be adjusted to compensating node just before branch `A` as highlighted by the red dashed line in Figure 6. Thus, target software will re-executing from compensating node to correct abnormal

states of those bypassed branch judgment `A` and `B`, which can eliminate the security problems described in §2.2.

4 Platform-Specific Implementation

In this section, we propose the implementation of RollSec based on Xen hypervisor, which locates in rollback-free control domain `Dom0` to avoid security problems of general rollback. Since we have already discussed the generality of extracting module in §3.5, at present, we only implement extracting methods of rollback-related states for C/C++ based software to demonstrate its effeteness. RollSec will utilize LibVMI library for recording and compensating those rollback-related states in target software of user domain `DomU`.

4.1 Extracting Module based on Code Property Graph

Extracting module will construct code property graph[25] of target software, and store it to Neo4J[26] graph database, which contains abstract syntax tree (AST), control flow graph (CFG) and program dependency graph (PDG). The extracting module described in §3.5 are implemented based on Python querying interfaces of Neo4J database, which can obtain those platform-independent information, including identifier name, modification location, rollback-related program point in CFG.

Furthermore, RollSec will interpret such information to offset in data and code sections for recording and compensating purpose during run-time based on debugging information. The extracting module is an offline preliminary procedure, which should be completed before recording and compensating.

4.2 VMI-based Recording and Compensating Module

RollSec will utilize LibVMI library to directly access target states in target software for state recording and compensating from control domain `Dom0`, without requiring modifications on target software in `DomU`. RollSec injects breakpoint interrupts (INT3) to all the modification locations of those variables in code segment to trace state transitions accordingly. After recording process is completed for an INT3 event, RollSec will remove this interrupt and recover the original code to make sure that target software can work properly. Then, RollSec will re-inject INT3 to enable monitoring again after the origin code is executed.

Since hooks are added to rollback functions (i.e. save and resume) in Xen tool stack libxl, whenever rollback happens RollSec will pause target software and invoke compensating process to correct those abnormal rollback-related states. For those rollback-related variables, RollSec will directly replace those abnormal states through VMI according to strategies described in §3.7. For those inappropriate resumed program point, RollSec will modify current control flow (i.e. process context) to corresponding compensation point as described in §3.5. Finally, paused target software will be unpaused without influenced by rollback security problems described in §2.

5 Evaluation

In this section, we conduct security evaluation to verify the effectiveness of RollSec firstly. Then, we present the results of performance experiments of RollSec.

5.1 Rollback Protection for Rollback-Related Variable

In this part, we will take Scenario I in §2.1 as example to verify the effectiveness for securing rollback-related variables. We apply RollSec to protect `pam_tally` module, which is loaded by `login` module as a library. Therefore, RollSec needs to relocate offsets of those rollback-related states in `pam_tally` module based on library mapping information in `login` at first. Since variable `tally` receives the number of failed login attempts from outside log file, it will be marked as rollback-related variables according to extracting methods described in §3.5. Then, RollSec will inject breakpoint interrupt to the modification location of variable `tally` for recording purpose, i.e., program point between line 6 and 7 in Listing 1.1.

Since state transitions during $t_1 \sim t_3$ are lost after rollback according to historical state log, RollSec will compensate this abnormal state to the latest state at t_4 according to strategy for non-random variables described in §3.7. Consequently, after rollback `login` module can still prevent attacks with password guessing, without becoming undermined by rollback as described in Scenario I.

5.2 Rollback Protection for Rollback-Related Program Point

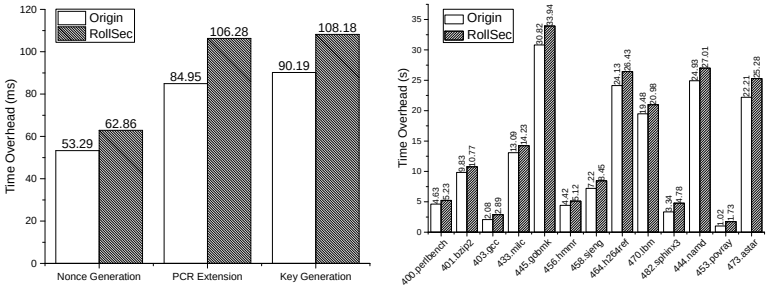
In this part, we will take Scenario II in §2.2 as example to verify the effectiveness for securing rollback-related program points. The program point after line 9 in Figure 2 can be reached, only if those rollback-related branch judgments for authentication is successful. Thus, program points after line 9 will be marked as rollback-related according to §3.5.

Since all branch judgments inside function `do_authentication` are bypassed after resuming program point at line 14 in Listing 1.2, RollSec will compensate the resumed program point by modifying control flow to line 4 for re-executing all those bypassed branch judgments to discard those abnormal control flow state for current time point. Moreover, before target software is unpasued, compensating for rollback-related variables are also required to make sure those bypassed judgments can be re-executed in correct states, which can eliminate security problems described in Scenario II.

5.3 Performance Evaluation

Our physical environment is equipped with a quad-core 2.00 GHz Intel Xeon E5405 processor, 8 GBytes RAM, 1 TBytes hard disk, which is running a 64-bit Linux Kernel 4.4.0 with hypervisor Xen 4.7.0 as `Dom0`. The VM in `DomU`

is based on Xen HVM (Hardware Virtual Machine) equipped with single vCPU and 2 GBytes memory. RollSec utilizes Neo4J 2.1.8 graph database to store code information of target software. The recording and compensating module utilize LibVMI 0.10.1 to manipulate rollback-related states in target software of user domain.



(a) Software TPM emulator (b) SPEC CPU2016

Fig. 7. Performance

We choose IBM's software TPM 2.0 (Version: 832) [27] and C/C++ benchmarks (ten C and three C++ programs) from SPEC CPU2016 [28] to evaluate the run-time performance of RollSec. We conduct performance experiments on frequently-used functions of software TPM, i.e. PCR extension, generation of key and nonce to evaluate run-time overhead of RollSec as depicted in Figure 7(a). Moreover, performance of RollSec, when applying to C/C++ based benchmarks of SPEC CPU2016 is shown in Figure 7(b). These experiments can demonstrate that RollSec will only cause acceptable overhead during run-time.

Table. 1 shows the overhead for compensating rollback-related variables in different memory size. We also evaluate the maximum time overhead for compensating rollback-related program point through modifying the control flow of target software, which shows that each time for modifying corresponding process context will cost 3.96 ms on average. Since rollback of virtual machine is a time-consuming process that usually cost seconds of time, the overhead of compensating those rollback-related states is negligible comparing to the entire overhead for VM rollback.

Table 1. Overhead for compensating rollback-related variable

512KB	1MB	2MB	4MB	8MB	16MB
12.15ms	25.45ms	53.38ms	106.78ms	180.66ms	373.56ms

6 related work

6.1 Applications of Rollback Mechanism

Rollback can help software recover from intrusion [2], failures [29, 4], meanwhile it also widely applies to software debugging scenarios such as [5, 30, 31]. Many research works on rollback are focusing on application fields, while its security problems are rarely mentioned.

6.2 Securing Software against Rollback

State continuity against rollback: Memoir [16], ICE [17] and Ariadne [18] all focused on state continuous system and algorithm to protect software from rollback problems. They are all based on replay strategies to maintain state continuity for target software, which can only work for those deterministic software and require modifications on target software according to dedicate libraries. Thus, applications of these solutions are largely limited.

State consistency against rollback: Jin et al.[9] proposed system to solve the rollback-induced state inconsistency between virtual machines (VM) and its attached virtual Trusted Platform Module (vTPM). However, Jin's solution only focused on a specific rollback problems on vTPM-VM architecture, which is not applicable for other scenarios with different architectures.

Securing rollback based on VM level: Garfinkel et al. [32] proposed the first research work to demonstrate that VM rollbacks can have severe security problems in virtualization environment. Xia et al.[14] proposed system to maintain a trusted log to record every rollback event of VM, which can only be analyzed manually by administers. These solutions cannot solve rollback impacts inside VM adequately.

7 Conclusion

In this paper, we revisited rollback problems in two representing scenarios, and summarized rollback-related states, including states of non-deterministic variables and program points, as the root causes of security problems of general rollbacks, and simplify it to the equivalent software state problems. Moreover, we propose RollSec to automatically protect software against general rollback problems, through correcting those abnormal rollback-related states caused by rollback. RollSec can automatically extract those rollback-related states from language-independent information of target software as preliminary work. During run-time RollSec can log state transitions, and maintain the correctness of those rollback-related states without requiring any modifications on target software. As last, we implement a prototype of RollSec based on Xen hypervisor for C/C++ software, and conduct evaluation experiments to verify its effectiveness and performance, which demonstrate that RollSec is effective and only acceptable overhead is introduced.

References

1. Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.
2. Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. Asynchronous intrusion recovery for interconnected web services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 213–227, 2013.

3. Haogang Chen, Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Identifying information disclosure in web applications with retroactive auditing. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 555–569, 2014.
4. Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, et al. Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 227–240, 2015.
5. Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the General Track: USENIX Annual Technical Conference*, pages 29–44, 2004.
6. Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T King, et al. Quickrec: prototyping an intel architecture extension for record and replay of multithreaded programs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 643–654, 2013.
7. Nima Honarmand and Josep Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *Proceedings of the 41st International Symposium on Computer Architecture*, pages 445–456, 2014.
8. Min Fu, Len Bass, and An Liu. Towards a taxonomy of cloud recovery strategies. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 696–701, 2014.
9. Hai Jin, Deqing Zou, Weiqi Dai, and Fengwei Wang. Synchronized virtual trusted platform modules (vtpm) and virtual machine (vm) rollbacks, March 1 2016. US Patent 9,275,240.
10. Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. Rote: Rollback protection for trusted execution. Cryptology ePrint Archive, Report 2017/048, 2017. http://eprint.iacr.org/2017/048.
11. Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 143–158, 2010.
12. Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 203–216, 2011.
13. Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 272–283, 2011.
14. Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Defending against vm rollback attack. In *Proceedings of International Conference on Dependable Systems and Networks Workshops*, pages 1–5, 2012.
15. Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 437–450, 2012.
16. Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

17. Raoul Strackx, Bart Jacobs, and Frank Piessens. Ice: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 106–115, 2014.
18. Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *Proceedings of 25th USENIX Security Symposium*, pages 875–892, 2016.
19. Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vtpm: virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium*, pages 305–320, 2006.
20. James Greene. Intel trusted execution technology. *Intel Technology White Paper*, 2012.
21. Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.
22. Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
23. Karl Trygve Kalleberg. Programming language independent abstract syntax trees, 2003. Presentation at NWPT'03: Fifteenth Nordic Workshop on Programming Theory, Turku, Finland.
24. Karl Trygve Kalleberg. Programming language independent abstract syntax trees, September 2004. Poster Presentation at the LASER 2004 Summer School in Software Engineering, Elba, Italy.
25. Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceeedings of IEEE Symposium on Security and Privacy*, pages 590–604, 2014.
26. Neo4J Developers. Neo4j. *Graph NoSQL Database [online]*, 2012.
27. YC Wang, Lin Yang, and WF Sun. Implementation of ibm vtpm with xen [j]. *Journal of Military Communications Technology*, 31(3):67–71, 2010.
28. John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
29. Min Fu, Liming Zhu, Len Bass, and An Liu. Recovery for failures in rolling upgrade on clouds. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 642–647, 2014.
30. Masao Maruyama, Tomoaki Tsumura, and Hiroshi Nakashima. Parallel program debugging based on data-replay. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 151–156, 2005.
31. Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32st International Symposium on Computer Architecture*, pages 284–295, 2005.
32. Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems*, 2005.