A Program Manipulation Middleware and Its Applications on System Security

Ting Chen[1](✉), Yang Xu[2], and Xiaosong Zhang[1]

[1] Center for Cyber Security, University of Electronic Science and Technology of China, Chengdu 611731, China
chenting19870201@163.com, brokendragon@uestc.edu.cn

[2] School of Mathematic Science, University of Electronic Science and Technology of China, Chengdu 611731, China
18215522740@163.com

Abstract. A typical program analysis workflow heavily relies on Program Manipulation Software (PMS), incurring a high learning curve and changing to another PMS requires completely recoding. This work designs a middleware, that sits between the applications and the PMS, hides the differences of various PMS, and provides a unified programming interface. Based on the middleware, programmers can develop portable applications without learning the PMS, thereby reducing the learning and programming efforts. The current implementation of the middleware integrates Dyninst (static analysis) and Pin (dynamic analysis). Moreover, we develop five security applications, aiming to prevent systems from stack overflow, heap corruption, memory allocation/deallocation flaws, invocations of dangerous functions, and division-by-zero bugs. Experiments also show that the middleware incurs small space & runtime overhead, and no false positives. Furthermore, the applications developed on the middleware require much less code, negligible runtime overhead, compared with the applications developed directly on Dyninst and Pin.

Keywords: Program manipulation middleware · System security
Unified programming interface · Portable applications

1 Introduction

Program analysis is a fundamental technique for various applications, such as software optimization [23], understanding [13], verification [22], debugging [3], testing [17], and software system protection [20]. When developing a particular application, programmers have to handle the Software Under Analysis (SUA) in a nontrivial way, e.g., translating the machine/source code into an analysis-friendly form, extracting control flows, tracking data flows, parsing symbol information. To alleviate programmers' burden, various Program Manipulation Software (PMS) [4,25,26,31,32] has been proposed to provide programming interfaces. Based on PMS, programmers can handle the SUA by directly invoking the programming interfaces without the need to parse the SUA by hand.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 606–626, 2018.
https://doi.org/10.1007/978-3-319-78813-5_31

However, the current development mode of program analysis applications has several drawbacks. First, the learning curve of using PMS is high and non-general, because different PMS has different programming interfaces. Second, for the same reason, programmers have to completely recode when their applications are required to change PMS. Different PMS has their own strengths. For example, both Pin [26] and Valgrind [32] are commonly-used dynamic instrumentation tools for offline binary analysis. Pin runs obviously faster than Valgrind [26] but demands application developers to handle machine code/assembly statements. However, Valgrind translates machine code into the VEX Intermediate Representation (IR) which is more analysis-friendly. Hence, the third drawback is that choosing an adequate PMS before developing is tricky because changing to another PMS is difficult.

Although many instrumentation languages have been proposed to simplify program manipulation, they suffer from one or more problems that can limit their effectiveness and utility in practice. These problems include the incapability of languages [19,29,34], the restrictions of PMS [8,9,15,19,27,28,34], the limited kinds of insertion points [8,15,19,27,28,30], the requirement that applications should be programmed by their proposed languages [8,15,18,19,30], lack of applications and experiments [27,28,33], and the learning efforts to grasp the proposed languages [9,18].

To overcome the aforementioned drawbacks, this work firstly designs a middleware that integrates different PMS, interacts with underlying PMS, handles the differences of various PMS, and provides an unified programming interface which is independent with PMS. Second, we propose a quick-start programming fashion, allowing programmers to execute arbitrary code feeding with various parameters in specified occasions. Application programmers can benefit from the two innovations. First, programmers can build their applications on the middleware without a deep understanding of the underlying PMS. Specifically, programmers need neither to understand the technical details of PMS nor to learn how to invoke PMS's programming interfaces. Second, applications can change to any other PMS on demand, requiring no modifications to the source code of applications. Consequently, programmers wouldn't feel difficult to choose PMS because porting to another PMS is effortless.

Compared to existing studies, our approach has the following advantages. First, it is designed to be general enough to support the development of various applications. Second, its design puts no restrictions on PMS and the current implementation supports a static PMS, Dyninst, and a dynamic PMS, Pin. Besides, current implementation supports various insertion points that can manipulate the SUA in different granularities. Furthermore, its effectiveness and efficiency are validated by several applications and experiments.

Our work has the following contributions.

1. We design and implement a middleware that provides an unified and easy-to-use programming interface to application developers.
2. On top of the middleware, we implement five applications that intend to protect systems from stack overflow, heap corruption, memory

allocation/deallocation flaws, invocations of dangerous functions, and division-by-zero bugs respectively. For comparison, we also implement those applications directly on top of Pin and Dyninst [5] respectively.
3. We conduct experiments to validate the effectiveness and efficiency of our approach. Results demonstrate that the middleware leads to acceptable space overhead, minimal runtime overhead, and no false positives. Besides, comparisons show that the applications developed on the middleware require much less code and have comparable performance with those developed directly on Dyninst and Pin. Furthermore, the applications are evaluated to be successful in protecting systems from CVE-2004-0597 and CVE-2011-3328.

The remainder of this paper is organized as follows. Section 2 introduces a motivating example. The design & implementation of the middleware are described in Sect. 3. Section 4 presents five applications for system security built on the middleware. Section 5 gives experimental results. We introduce the related work in Sect. 6 and conclude the paper with future work in Sect. 7.

2 Motivating Example

In this section, we use a simple example to illustrate the motivations of our work. The example application is an instruction logger that records the number of executed instructions. Figure 1(a), (b), (c) present the logger's source code built on Pin, Dyninst and our middleware respectively. Figure 1(d) is the related configuration to Fig. 1(c). For the sake of presentation, we omit less important code in Fig. 1(a) and (b), while we show complete code in Fig. 1(c) and (d).

As shown in Fig. 1(a), after initialization (Line 6), the logger registers a callback (Line 7), termed by *Instruction* (Line 3) that will be invoked immediately before a code sequence is executed for the first time. In *Instruction*, an analysis function *docount* (Line 2) is inserted before each instruction (Line 4), ensuring that *docount* will be executed exactly before the execution of each instruction. The *docount* just increases the global variable *icount* by 1, that indicates the number of instructions has been executed so far. Finally, the SUA will be run after invoking *PIN_StartProgram* which is a Pin's API.

The implementation on Dyninst looks more complex (Fig. 1(b)). It firstly opens the SUA (omitted in Line 8) and then creates an integer *intCounter* which can be inserted into the SUA (Line 9). Then the logger enumerates all modules (Line 10) and further all functions of each module (Line 11). The function *rtndeal* is called (Line 12) whenever it finds a function in the SUA. In *rtndeal* (Line 1), the logger enumerates all blocks (Line 2) and further all instructions of each block (Line 3). Afterwards, all insertion points of each instruction are enumerated (Line 4). Then the logger constructs an arithmetic expression *addOne* (Line 5), that has the same effect with the C code *intCounter++*. The constructed expression will be inserted into each insertion point (Line 6). Finally, the modified SUA should be written back to the disk (Line 13).

The implementation on our middleware (Fig. 1(c)) is much simpler, that is a function *TargetCount* increasing a global variable *insnum*. To make the code be

```
1   static UINT incount=0;
2   VOID docount(){icount++;}
3   VOID Instruction(INS ins,VOID *v){
4     INS_InsertCall(ins,IPOINT_BEFORE,docount,IARG_END);}
5   int main(int argc,char *argv[]){
6     if(PIN_Init(argc,argv))return Usage();
7     INS_AddInstrumentFunction(Instruction,0);
8     PIN_StartProgram();
9     return 0;
10  }
```

(a) Instruction logger built on Pin

```
1   void rtndeal(...){
2     for(...)//enumerate blocks
3       for(...)//enumerate instructions
4         for(...){//enumerate insertion points
5           BPatch_arithExpr addOne(BPatch_assign,*intCounter,
    BPatch_arithExpr(BPatch_plus,*intCounter,BPatch_constExpr(1)));
6           addSpace->insertSnippet(addOne,**point_iter);}}
7   int main(int argc,char *argv[]){
8     //open the SUA
9     intCounter=addSpace->malloc(*(appImage->findType("int")));
10    for(...)//enumerate modules
11      for(...)//enumerate functions
12        rtndeal(*func_iter,*module_iter);
13    dynamic_cast<BPatch_binaryEdit*> (addSpace)->writeFile("out");
14    return 0;
15  }
```

(b) Instruction logger built on Dyninst

```
1   int insnum=0;
2   extern "C" void TargetCount(){
3     insnum++;}
```

(c) Instruction logger built on our middleware

```
1   image a.out function all insnstring all before
2   funcalllib TargetCount
```

(d) Configuration of (c)

Fig. 1. The source of an instruction logger

interpreted by the middleware, a configuration (Fig. 1(d)) should be prepared, which is also very simple. It indicates that exactly before each instruction of each function in the SUA *a.out* executes (Line 1), a function *TargetCount* (Fig. 1(c)) should be called (Line 2). The reserved keywords, such as *image*, *function*, *all* are self-explanatory. The grammar of configuration will be introduced in Sect. 3. The technical details and differences of various PMS are hidden by the middleware. For example, application developers need not to write code to enumerate all functions in this example.

Several interesting observations can be found from the example. First, the source code of the application on Pin differs greatly from that on Dyninst, indicating that the programmers obeying conventional programming mode have to completely recode when they prepare to change the underlying PMS. Second, programmers have to spend a period grasping the programming interfaces of a particular PMS. Third, based on our middleware, programmers can get start

to code much quicker, and develop more concise, PMS-independent as well as PMS-portable applications.

3 Design and Implementation

3.1 Design

Figure 2 shows the high-level architecture of the middleware which sits between the applications and the PMS. The middleware integrates various PMS (PMS 1 to PMS n), that directly interacts with PMS, and provides a PMS-independent programming interface to above applications (App 1 to App m). Applications cannot communicate with PMS directly; instead, they have to delegate the work of program manipulation to the middleware. This work proposes a unified and simple programming mode, so programmers just need to learn how to use the middleware. Programmers need to compile their applications into the form (always binaries) that the chosen PMS can understand, regardless of the source code language used. The middleware works like a virtual machine because it hides the details of the underlying PMS from the upper applications, and it can seamlessly switch from one PMS to another according to the demand of analysts. The middleware is designed to be general-purpose, that should support various applications. We will present several applications that are developed on the agent in Sect. 4.

The architecture takes inputs as the SUA and a configuration which describes where is the code specified by programmers and when the code should be executed. Take Fig. 1(d) as an example, the configuration can be interpreted as "the code is in the function *TargetCount* and the code should be executed exactly before each instruction of each function in the binary *a.out*". The configuration should be provided by application programmers. But as we will show in Sect. 3.2, the grammar is simple and self-explanatory. The outputs of the middleware should be the SUA after process or using the application to analyze the SUA, depending on whether the PMS manipulates the SUA statically or dynamically. Specifically, if a PMS manipulates the SUA when running it, such as Pin, Valgrind, Qemu [4], the code specified by application programmers will be loaded into memory at runtime. On the contrary, if a PMS handles the SUA statically, such as Dyninst, LLVM [25], CIL [31], a modified SUA with the inserted code will be generated. When executing the modified SUA, the inserted code has opportunities to run.

The workflow of the middleware is as follows. First, it loads the SUA and the applications. After that, it parses the configuration to get to know when to execute the code provided by programmers. Third, it interacts with the underlying PMS to insert the specified code into right places, ensuring that the inserted code should be executed at the right time. Finally, it analyzes the SUA or generates a modified SUA depending on the underlying PMS.

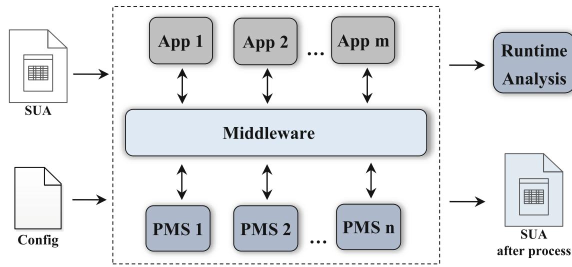

Fig. 2. Architecture

3.2 Implementation

To validate the proposed middleware can integrate various PMS, we choose a representative dynamic PMS, Pin and a representative static PMS, Dyninst. Please note that, although Dyninst is capable of handling programs dynamically, we just take advantage of its static manipulation ability. Few additional efforts are required to extend current implementation to support dynamic instrumentation of Dyninst, because Dyninst exports the same programming interface for both static and dynamic instrumentation abilities.

The versions of the integrated PMS are Pin-2.14-71313 and Dyninst 8.1 respectively. But we believe minor revisions are required in the middleware when porting to other versions, because a PMS usually provides a stable programming interface among different versions. However, Pin and Dyninst differ significantly in programming fashion, as shown in the motivating example (Sect. 2). Therefore, the most coding effort for our implementation is made to handle the differences of Pin and Dyninst. Both Pin and Dyninst are binary manipulation tools, so the SUA and the applications should be given in binary form.

Before starting analysis, the SUA and the applications should be loaded into memory. For Pin, the SUA is specified in the command line, so Pin loads the SUA automatically. On the contrary, programmers need to invoke *BPatch::openBinary* using Dyninst. Pin loading applications is as usual as a normal desktop program loading dynamic libraries, for example, invoking *dlopen*. Differently, to load the applications, Dyninst provides a special API, *BPatch_binaryEdit::loadLibrary*.

The middleware converts the configuration into a special designed structure, *execution bag* that consists of multiple *execution block*s. One execution block provides the code specified by programmers, when the code should be executed, as well as the parameters accepted by the specified code. The specified code should be in form of a function resided in the loaded application. Programmers should give the function names so that the middleware can find function addresses. For Pin, function addresses are found by invoking *dlsym*; while Dyninst-based applications should invoke *BPatch_image::findFunction*. The execution bag puts no restrictions on the number of execution blocks, facilitating application programmers to develop complicated applications.

Table 1. Insertion points supported by the middleware

No.	Granularity	Description
1	Image level	Before image loading
2		Before image unloading
3	Function level	Before function entry
4		Before function exit
5	Instruction level	Before execution of instructions with specified opcode
6		After execution of instructions with specified opcode
7		Before calling a specified function
8		After calling a specified function
9		Before execution of instructions with specified number of operands
10		After execution of instructions with specified number of operands

The middleware provides ten types of insertion points (as shown in Table 1) where programmers can insert their code. We are in process of enriching the insertion points to enable programmers to control the SUA more flexibly. The current implementation allows the inserted code to run when a given image is loading (row 1) or unloading (row 2). Programmers should give the image name in the configuration or 'all' indicating all images should be monitored. When an application chooses to run on Pin, the middleware registers two callbacks, *Imageload* and *Imageunload* respectively by calling *IMG_AddInstrumentFunction* and *IMG_AddUnloadFunction*. The *Imageload* will be invoked whenever an image is loading, while the *Imageunload* will run whenever an image is unloading. When the two callbacks run, they firstly check whether the loading/unloading image is the desired one; if so, the application's code will be invoked. For Dyninst, the middleware firstly enumerates all modules of the SUA and then inserts application's code into the entry points (by invoking *BPatch_module::insertInitCallback*) and the exit points (by invoking *BPatch_module::insertFiniCallback*) of the specified module respectively.

The current implementation of the middleware also allows programmers to run their code before (row 3) or after (row 4) the execution of a specified function. Programmers should give the function name to be monitored or 'all'. When using Pin, the *Imageload* enumerates all functions whenever an image is loading. Then, a function *rtndeal* is used to handle each enumerated function. After that, the *rtndeal* checks whether the handled function is of interest by invoking *RTN_FindByName*. If so, the application's code is inserted into the entry points or exit points by calling *RTN_InsertCall* with the parameter *IPOINT* being *IPOINT_BEFORE* or *IPOINT_AFTER* respectively. The implementation on Dyninst is similar except that it finds the entry points and exit points of each function using Dyninst's APIs *BPatch_function::findPoint*(*BPatch_entry*) and *BPatch_function::findPoint*(*BPatch_exit*) respectively.

In instruction level, the middleware allows programmers to handle the SUA more flexibly, as shown in Table 1 that six types of insertion points are supported. Programmers can insert application's code before (row 5) or after (row 6) the instructions with specified opcode. To facilitate programmers, the opcode can be given in string, such as 'add' and 'div'. Moreover, it allows programmers to insert code before (row 7) or after (row 8) calling a specified function. This type of insertion points is useful because function calls are sometimes related to security bugs, e.g., format string vulnerabilities, insecure string functions, memory allocation/deallocation, and taint sources/sinks. Programmers need to give the concerned function name, or simply 'all' indicating all function calls deserve attentions. Furthermore, programmers can specify the operand number, and insert application's code before or after the instructions with the specified operand number (Line 9, 10). The two insertion points can benefit the development of data-flow-related applications (*e.g.* taint analysis) since programmers can handle different instructions with the same operand number in an unified way.

To enable instruction-level program manipulation, the middleware invokes *INS_InsertCall* of Pin. For Dyninst, the *rtndeal* firstly enumerates all blocks of a function and then enumerates all instructions of each block, followed by checking whether the instructions are concerned. If so, the application's code is inserted by invoking *insertSnippet* exported from *BPatch_addressSpace*. To coordinate different PMS, we do not consider implicit operands. Hence, programmers need to handle implicit operands in her own way.

The programming mode is designed to be flexible that programmers can specify a composite insertion point by combing several default ones (one example is shown in Fig. 1(d)). For instance, one can ask the middleware to insert code before each call to *malloc* of a function named *main* in an image *helloworld*, by giving a composite insertion point like '**image** helloworld **function** main **funcall** malloc *before*'.

The middleware is able to handle the parameters specified by application programmers, and send the parameters to application's code. The ability can benefit programmers because the application's code usually needs the information from the SUA, context, and runtime environment, *etc.* The current implementation supports seven kinds of parameters as shown in Table 2. The types of parameters and their usages are easy to understand. We just need to mention that when programmers specify one operand of an instruction as a parameter, both the type (*i.e.*, immediate number, register or address) and the value of the operand will be obtained as two consecutive parameters. We plan to develop a GUI allowing programmers to prepare the configuration by simply choosing and clicking, thereby removing the requirement of learning the grammar of the configuration.

The middleware provides another functionality that may interest programmers when they need to stop the running of the SUA and investigate the runtime context. For example, if a security bug is discovered, programmers always want to know how the bug is triggered. The middleware encapsulates the functionality into a function *dump*, that can be called anywhere in the application. When

Table 2. Types of parameters supported by the middleware

No.	Type	Example	Parameters
1	Constant	Constant 10	Constant value
2	Register	Reg eax	Register's value
3	Disassemble	Dis	Disassemble of the specified instruction
4	String	String abc	String's value
5	Funname	Funname	Function name of specified function
6	Imagename	Imagename	Image name of specified image
7	Operand	Operand 0	Type of operand and the value of the operand

dump is called, the SUA is stopped and the context information including the current instruction, register values, the call stack *etc.* is dumped. Section 5.3 will show that the *dump* function benefits the localization, analysis and debugging of software vulnerabilities.

3.3 Future Extensions for Other PMS

Currently, the middleware supports Pin and Dyninst, while the idea and design are general. We are working to extend our implementation to support more PMS. From the perspective of implementation, we classify the current PMS into several categories. Therefore, we can use similar methods to handle different PMS which belongs to the same category.

The first category is dynamic instrumentation tools, such as Pin, Valgrind, and DynamoRIO [6] that usually provide explicit programming interfaces. Similar to what we have done for Pin, we can take advantage of their APIs, so that we need not care about their internal technical details. The second category is static instrumentation tools, for example, Dyninst and CIL. Fortunately, existing static instrumentation tools also provide rich APIs that facilitate program manipulation. We can extend our implementation to other static instrumentation tools in a similar way with how we deal with Dyninst.

The third category is virtual machines, such as Qemu, Temu [36] and Java Virtual Machine (JVM), which are able to monitor and modify SUA's execution flow. As the current VMs do not often provide explicit APIs, our implementation needs to embed the middleware into the VM which is responsible for inserting application's code into proper places. Alternatively, we can use Virtual Machine Introspection (VMI) [16] to monitor the SUA out of the box. Although the implementation for VMs is more tricky than handling instrumentation tools, the two aforementioned methods have been widely applied in existing VM-based program analysis techniques.

The last category is complier-like program analysis tools, such as GCC and LLVM that conduct analysis statically. In this case, our implementation needs to register the middleware as a plugin (*i.e.*, compiler pass), ensuring that it has

chances to manipulate the SUA during the compiling procedure. The compiler-like tools often provide programmer-friendly programming interfaces to develop plugins (for example, KLEE [7] is a symbolic executor which is a plugin of LLVM). Hence, it is technically practical to enhance our implementation with the ability of supporting those compiler-like program analysis tools.

To make our implementation adaptable to various CPU infrastructures (*e.g.*, x86, x64, ARM) and different representations of the SUA (*e.g.* sources, binaries, bytecodes), the introduction of an Intermediate Representation (IR) can benefit programmers a lot. In most cases, there is no need to design a novel IR because existing IRs (*e.g.* VEX used by Valgrind [32], LLVM-IR proposed by LLVM [25], and CIL [31]) could be adequate. Next, what we need to do is translating the SUA into the selected IR. Fortunately, some open-source PMS supports IR translation that could be directly reused by our middleware. For example, Valgrind can convert x86, x64, ARM, PPC, MIPS *etc.* into VEX. LLVM can translate C, C++, Objective-C, Java and so on into LLVM-IR. As another example, McSema [14] and S2E [11] can translate x86 binaries into LLVM-IR.

The current implementation does not modify the program logic of the SUA; instead, it just observes and analyzes. We believe it is not difficult to extend our implementation for program transformation. In most cases, PMS (*e.g.*, Pin, Dyninst, CIL, LLVM) has already provided APIs for program transformation. In the cases that program transformation is not explicitly supported (*e.g.*, VMs), our implementation can achieve this goal by inserting a jump before the code needed to be transformed and then inserting the code after transformation into the jump target.

4 System Security Applications

Various applications can be built upon our middleware, such as instruction tracers, memory operation tracers, code coverage profilers, taint analyzers, concolic executors. This section describes the implementation of five applications that aim to protect software systems from stack overflow, heap corruption, memory allocation/deallocation errors, invocations of dangerous functions and division-by-zero bugs respectively. We only give the full details related to division-by-zero bugs, including the configuration and the inserted code due to page limitation. In the end of the section, we will explain how to implement a taint analyzer and a concolic executor (two of the most compelling and complicated program analysis techniques) based on our middleware. The applications can run on various PMS (Dyninst and Pin of the current implementation) by specifying the PMS through the command line. That's to say, there is no need to modify the applications' source code and the associated configurations.

4.1 Division-by-Zero Bugs

Figure 3 presents the source code as well as related configuration of division-by-zero bugs protector. Please note that the configuration should be written

according to a simple grammar as shown in Fig. 3(b), while the source code allows any programming languages that can be compiled into binaries. The configuration informs the middleware that the code in function *TargetDiv* (Line 4) should be executed exactly before *div* (Line 1) and *idiv* (Line 2) in the SUA *a.out*. The first operand of *div* and *idiv* should be passed to *TargetDiv* as a parameter (Line 3). According to Intel instruction manual, the first operand of *div* and *idiv* is the divisor that should be checked in *TargetDiv*.

```
1  extern "C" void TargetDiv(int type,int opVal){
2    bool nonZero=(type==ADDR)? *opVal:opVal;
3    if(!nonZero)dump();}
```

(a) Source of the division-by-zero protector

```
1  image a.out function all insnstring div before
2  image a.out function all insnstring idiv before
3  operand 0
4  funcalllib TargetDiv
```

(b) Configuration

Fig. 3. Source code and configuration of division-by-zero protector

If the programmer specifies an operand parameter, the type and the value of the operand will be passed to the target code. As defined, the first operand of *div* and *idiv* can be an immediate number, a register or an address. If the operand is an address, the value stored will be retrieved. Otherwise, *opVal* itself is the divisor. If the divisor is zero, indicating a division-by-zero bug, the application invokes *dump* to stop the execution of *a.out* and output the vulnerability information.

4.2 Stack Overflow

The stack overflow protector shares the same idea with TRUSS [35], that is similar with StackShield [1]. When calling a function, the return address of the function is stored in a shadow stack. When the function returns, the return address picked from the runtime stack will be compared with the one in the shadow stack. If they do not match, an attack will be detected and the SUA will be terminated. The major difference between TRUSS with our application is that the former is directly built on DynamoRIO, while our application is developed on top of the middleware, thus our protector can port to another PMS easily. StackShield differs a little in idea that it directly restores the return address from the shadow stack without checking.

To record return addresses, the application's code should be executed exactly before the entry points of each function. To compare return addresses, the related code should run before the exit points of each function. The two types of insertion points are supported by the middleware (Table 1 rows 3 and 4). However, during evaluation, we find that the simple store-match method may introduce high

false positives due to dynamic loading or compiler optimizations (*e.g.*, *setjmp/longjmp*). As a consequence, modern PMS tries to find all entry points and exit points of a given function, but success is not guaranteed. To reduce false positives, when recording a return address, the protector also records the current stack pointer and the function name (if existed) in the shadow stack. The application will report a stack overflow attack only if the stack pointers and function names match; meanwhile, the return addresses do not match.

4.3 Heap Corruption

The idea to prevent heap corruption is (1) recording the locations and sizes of allocated heaps; (2) monitoring all heap operations; (3) reporting bugs, if any operations override the boundaries of heaps. Our heap corruption protector records the locations and boundaries of heaps by monitoring the invocations of heap allocation and deallocation functions, such as *malloc*, *calloc*, *realloc* and *free*. The protector allows programmers to run specified code before or after calling a given function (Table 1 rows 7 and 8). Whenever the SUA requests for allocating memory, the heap location and size are recorded by the inserted code. After the request, the inserted code checks whether heap allocation is successful. If not, the related record will be removed. Additionally, after the successful deallocation of a heap, the related record will also be deleted.

To monitor heap operations, the protector keeps an eye on the invocations of string and memory functions, such as *strcpy*, *strcat*, *strncpy*, *memcpy* and *memmove*. To detect heap corruption, some parameters of the monitored functions are required to be passed to the inserted code. To do so, the application specifies proper registers as parameters, because function parameters can usually be found in registers or the stack (can be located by the stack register).

4.4 Memory Allocation/Deallocation Errors

The memory allocation/deallocation errors handled in the protector include double free, free a non-heap memory location and free a pointer that points to the middle of a heap and so on (*i.e.*, any memory bugs that deallocate wrong heap locations). The protector firstly records the addresses and boundaries of allocated memory by instrumenting functions like *malloc*, *calloc*, *realloc*. Then it checks memory deallocation to ensure that the freed address is the exact address recorded. Otherwise, the SUA will be stopped and a detailed report will be given.

4.5 Invocation of Dangerous Functions

Detecting the invocations of dangerous functions are similar to the way of detecting the calls of *malloc*. The protector can detect *getpw*, *gets*, *random*, *vfork*, *mktemp*, *mkstemps*, and *mkdtemp*. It is straightforward to enrich the set of dangerous functions by adding a few lines in the configuration, because the application handles various dangerous functions in a unified way as follows.

One kind of functions are dangerous whenever they are invoked. For example, *getpw* is extremely dangerous because it gets user names, passwords and other privacy information from */etc/passwd*. For this kind, the application inserts code before the instructions that call dangerous functions. Whenever a dangerous function of this kind is invoked, the application can stop the attack immediately. The other kind can be considered as dangerous under specific context. For example, *mktemp* becomes dangerous when the file name is too short. To detect this kind of dangerous functions, the application passes the demanded information as parameters to the inserted code.

4.6 Taint Analyzer and Concolic Executor

Taint analysis [12] consists of marking taint sources, tracking taint propagation, and warning if taints enter taint sinks. Taint analysis has wide applications in vulnerability detection, malware analysis, privacy protection *etc.* Programmers are able to develop a taint analyzer based on the middleware via the simple programming mode. Taint sources are usually functions that get data from environment, such as *ReadFile*, *recv*, *getenv*. To mark taint sources, the taint analyzer needs to instrument before and after the related functions. To track taint propagation, instruction-level instrumentation is needed that takes responsible for tainting target operands if source operands are tainted. Taint sinks are usually special functions which operate on tainted data, such as *WriteFile*, *send*, *system*. Therefore, those sensitive functions should be monitored by function-level instrumentation.

Concolic execution [10] is a variant of traditional symbolic execution [24] that collects path constraints along with concrete execution, and then explores other paths triggered by new test inputs that are solved from negated path constraints. Concolic execution is an iterative procedure that runs the SUA with given inputs, symbolizes inputs, tracks symbol propagation, collects constraints, and then generates new test inputs. Input symbolization is similar with marking taint sources. However, symbolic inputs could be the parameters of given functions, register values, memory values *etc.*, that could be specified as parameters as shown in Table 2. To track symbol propagation, the application also needs instruction-level instrumentation. The instrumented code should interpret instruction semantics, and then compute symbolic expressions of the influenced operands. The application needs to collect constraints when executing symbol-related conditional jumps (*e.g.*, *jz*, *jnz*, *ja*, *jb*). Programmers will feel convenient to handle specific instructions because the middleware allows programmers to specify the concerned instructions by giving the string forms of opcodes, as shown in Table 1 (rows 5 and 6).

5 Experiments

5.1 Research Questions

We attempt to answer the following research questions through experiments. **QA1**: Will the SUA modified by the middleware bring about unacceptable

runtime overhead, space overhead and false positives (Sect. 5.2)? **QA2**: Can the application prevent the attacks against real CVE vulnerabilities (Sect. 5.3)? **QA3**: Can the middleware facilitate the localization, analysis and debugging of software defects (Sect. 5.3)? **QA4**: Can the middleware reduce code amount of application development (Sect. 5.4)? **QA5**: Will the middleware lead to obvious runtime overhead (Sect. 5.5)?

All experiments are conducted on a laptop, equipped with a two-core Celeron CPU (1.8 GHz), 2 GB main memory and 64-bit CentOS 7.

5.2 Experiments with Benchmark Programs

We select ten daily-used programs in CentOS arbitrarily as a benchmark set including compilers, compression/decompression software, SSL tools, a multimedia processing library *etc.* As shown in Table 4, the sizes of the SUA range from 7,136 bytes to 772,704 bytes, 385,904 bytes on average (standard deviation is 299,779, indicating significant differences). For convenience, we integrate the five applications into one multi-functional software protector, and demonstrate the experimental results when testing the integrated software protector on Dyninst in Fig. 4.

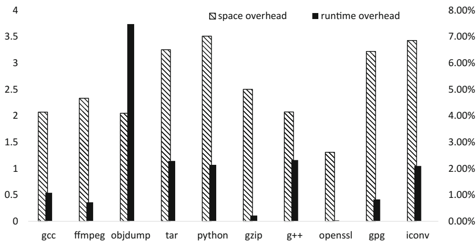

Fig. 4. Experimental results with ten benchmark programs

Figure 4 shows the size expansion of the SUA after processing by the middleware. The space overhead is 2.57x on average, because the software protector is multifunctional that inserts protection code before all entry points and exit points of each function, before all *div* and *idiv* instructions, before and after all memory allocation/deallocation functions, and also before all invocations of dangerous functions. Fortunately, disk space is not as scarce as decades ago, so nowadays it is worthy of trading space for security. Figure 4 also shows that the modified SUA runs slightly slower than the original SUA, 1.92% on average. Furthermore, we find that the runtime overhead has no direct bearing upon the space overhead, because the former depends on how the SUA is executed, while the latter relies on how the SUA is processed. No security problems are reported in those benchmark software, so, there are no false positives.

Hence, we can answer QA1 that *the SUA modified by our middleware incurs acceptable space overhead, minimal runtime overhead, and no false positives.*

5.3 Practical Case Studies

In this section, we will evaluate the effectiveness of the software system security protector against CVE-2004-0597 and CVE-2011-3328. Please note that both the two cases are successfully reproduced on Pin and Dyninst, and we get identical reports regardless of underlying PMS.

CVE-2004-0597. CVE-2004-0597 consists of multiple buffer overflows in *libpng* 1.25 and earlier, allowing remote attackers to execute arbitrary code via malformed PNG images. We examine our protector by testing with one stack overflow vulnerability. The malformed input is shown in Fig. 5(a) that the 48 bytes starting from offset 0x129 are overwritten with 0x41 (*i.e.*, the letter 'A'). The overwritten part is in an *IDAT* truck that contains the actual image data.

```
00000100h: C1 1A 52 07 43 E6 82 0F B9 E0 01 15 CC 21 05 6B ; ?R.C鎮.灌..?.k
00000110h: A8 DA 9B E6 30 50 7D 53 CD A9 FB B0 03 E5 9B EA ; 丫泼0P}S桐  .鍥?
00000120h: D6 75 18 B0 E6 A6 3A 18 94 41 41 41 41 41 41 41 ; 謚.版?.擨AAAAAAA
00000130h: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ; AAAAAAAAAAAAAAAA
00000140h: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 ; AAAAAAAAAAAAAAAA
00000150h: 41 41 41 41 41 41 41 41 41 33 23 E3 FD 22 E2 F7 ; AAAAAAAAA3#泯"愒
00000160h: E7 FB FB FE 1E FC 21 64 58 D8 D5 CF 8F 86 2B 80 ; 琨  .?dX卣蠌?€
```

(a) Malformed input

```
Breakpoint 1, 0x00007f41768dba06 from ./libpng12.so.0 //where to trigger the bug
#0   0x00007f41768dba06  ┐
#1   0x4141414141414141  ├ Call stack
#2   0xe7f7e222fde32333  │
...                      ┘
rax  0x1                 ┐
rbx  0x4141414141414141  │
...                      ├ Regs
rbp  0x4141414141414141  │
...                      ┘
=> 0x7f41768dba06:  retq                                          ┐
   0x7f41768dba07:  lea     -0x324c7d(%rip),%rsi                  │
   0x7f41768dba0e:  callq   0x7f4176582060 <png_error@plt>        │
   0x7f41768dba13:  movzbl  0x276(%rbx),%edx                      ├ Ins
   0x7f41768dba1a:  cmp     $0x3,%dl                              │
   0x7f41768dba1d:  jne     0x7f41768db91c                        │
   ...                                                            ┘
```

(b) Report

Fig. 5. Test results with CVE-2004-0597

The report (Fig. 5(b)) which is produced by the *dump* function mentioned in Sect. 3.2, consists of four parts. The first part shows the address (0x7f41768dba06) of the instruction that triggers the vulnerability and the buggy executable (*libpng*12.*so*.0). Part 2 presents the call stack. We can see that the call stack is corrupted due to stack overflow. That is, the function address with depth 1 is 0x4141414141414141 (i.e., multiples 'A's) which comes from the malformed input. Besides, the function addresses from depth 2 to the bottom are weird values that should not be function addresses. Part 3 gives register values. We can find that some registers, especial *rbp* that involves control flow transfers are polluted by the input. The final part shows the critical instruction (address

and disassemble) as well as some instructions after it. As expected, the critical instruction is *retq*, before which the protector inserts checking code.

CVE-2011-3328. CVE-2011-3328 is a division-by-zero bug located in the *png_handle_cHRM* function of *pngrutil.c* in *libpng* 1.5.4, enabling a denial of service attack via a malformed PNG image containing a *cHRM* chunk. Figure 6(a) shows a malformed input that triggers CVE-2011-3328. Three DWORD variables *y_red*, *y_green* and *y_blue* correspond to offset 0x35, 0x3d and 0x45 respectively in the first *cHRM* chunk. The sum of *y_red*, *y_green* and *y_blue* will be used as a divisor. Hence, we set all of them to be zeros.

```
00000000h: 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52 ; 塒NG........IHDR
00000010h: 00 00 00 F0 00 00 00 87 08 02 00 00 00 A7 10 43 ; ...?..?....?C
00000020h: BC 00 00 00 20 63 48 52 4D 00 00 7A 25 00 00 80 ; ?.. cHRM..z%..€
00000030h: 83 00 00 F9 FF 00 00 00 00 00 00 75 30 00 00 00 ; ?.?......u0...
00000040h: 00 00 00 3A 98 00 00 00 00 30 C6 DF BF 00 00 00 ; ...:?...0七?..
00000050h: 09 70 48 59 73 00 00 0B 13 00 00 0B 13 01 00 9A ; .pHYs.........?
```

(a) Malformed input

```
Breakpoint 1, 0x00007f9280d18b56 from /libpng15.so.15//where to trigger the bug
#0   0x00007f9280d18b56  ┐
#1   0x00007fff9de7f8a0  ├ Call stack
#2   0x00007fff9de7f890  │
...                      ┘
rax  0x0  ┐
rbx  0x0  │
rcx  0x0  ├ Regs
rdx  0x0  │
...       ┘
=>   0x7f9280d18b56:  div   %ecx                ┐
     0x7f9280d18b58:  xor   %edx,%edx           │
     0x7f9280d18b5a:  mov   %ax,0x42a(%r15)     ├ Ins
     0x7f9280d18b62:  mov   %ebx,%eax           │
     0x7f9280d18b64:  shl   $0xf,%eax           │
     ...                                        ┘
```

(b) Report

Fig. 6. Test results with CVE-2011-3328

The report is shown in Fig. 6(b) that the instruction which triggers a division-by-zero error locates in 0x7f9280d18b56. The vulnerable binary is *libpng*15.*so*.15. We can see that the attack doesn't subvert the call stack. However, several registers are polluted, especially *rcx* whose lower 32 bits (i.e., *ecx*) are treated as a divisor. As expected, the vulnerable software stops before running the division-by-zero operation, because the protector inserts checking logic before all *div* and *idiv* instructions.

Therefore, we can answer QA2 and QA3 that *(1) the application can prevent software systems from real attacks; (2) the report produced by our middleware can facilitate the localization, analysis and debugging of software defects.*

5.4 Comparison with Dyninst and Pin

This subsection presents the code amount (in lines) of the applications developed on middleware, and those directly developed on Dyninst and Pin respectively,

as shown in Table 3. Please note that we implement all five applications directly on Dyninst and Pin respectively for comparison and the code amount is counted by SourceCounter [2]. The figures after '|' indicate times. For example, the code amount of the division-by-zero protector built directly on top of Dyninst is about 30.2 times larger than that developed on our middleware. The last row shows the averages. The observation is that the applications based on our middleware require the fewest code lines, answering Q4 that *our approach can reduce the code amount of applications obviously*. Second, the code amount for Pin-based applications is comparable with that for Dyninst-based applications, indicating that the two PMS encapsulates the manipulations of the SUA in comparable degrees. We have to remind that the metric, code amount cannot reflect the learning curve of various PMS. For example, a well-documented PMS is easier to learn than the PMS with few documents. Besides, code amount can just partially reflect the developing efforts. For example, a line of code invoking a complicated API needs more time to debug and test than a line of assignment.

Table 3. Code amount of the applications (LOC) developed on the middleware, Dyninst, and Pin

Application	Agent	Dyninst	Pin
Division-by-zero	6	181|30.2	130|21.7
Stack overflow	60	169|2.8	174|2.9
Heap corruption	86	279|3.2	272|3.2
Memory allocation/deallocation	41	214|5.2	206|5
Dangerous function	16	133|8.3	159|9.9
Average	**41.8**	**195.2|4.7**	**188.2|4.5**

5.5 Runtime Overhead of the Middleware

Our middleware will lead to runtime overhead for dynamic PMS because it controls dynamic PMS at runtime. Table 4 answers QA5 that *the middleware incurs minimal runtime overhead (i.e., 1.44% on average)*. The figures after '|' in the last column indicate the runtime overhead incurred by our middleware, compared to the application directly developed on Pin. Please note that the application tested here is the integrated application which consists of all five functionalities mentioned in Sect. 4.

The overhead incurred by our middleware is negligible, compared to the overhead incurred by the application. The third column gives the time for running each SUA in the environment of Pin (i.e., the SUA is loaded by Pin with an empty application). The figures after '|' in the fourth column present the overhead caused by the application compared to the time consumption shown in the third column, which is 31.11x on average. Hence, the middleware just leads to less than a thousandth of the overhead caused by the application.

Table 4. Runtime overhead incurred by the middleware

SUA	Size (Byte)	Baseline (sec)	Directly on Pin (sec)	On our middleware (sec)
iconv	60320	0.64	5.34|7.37	5.39|0.95%
gpg	749840	2.07	186.73|89.40	186.94|0.11%
openssl	508680	1.24	87.15|69.33	87.74|0.67%
g++	772704	1.15	53.70|45.54	54.99|2.41%
gzip	100744	0.82	6.21|6.54	6.65|7.06%
python	7136	23.40	26.76|14%	26.85|0.34%
tar	345976	1.38	35.28|24.62	36.03|2.11%
objdump	332248	1.34	41.84|30.17	41.89|0.13%
ffmpeg	212800	131.14	141.47|8%	141.96|0.35%
gcc	768592	1.26	49.19|37.96	49.35|0.32%
Average	**385904**	**17.44**	**63.37|31.11**	**63.78|1.44%**

6 Related Work

Substantial studies have been made to reduce the difficulty of manipulating the SUA. However, existing works suffer from a few drawbacks. We just summarize the drawbacks and describe a few of related works due to page limitation.

First, some proposed languages are incapable of supporting complicated applications. To name a few, the capability of Atune-IL [34] is restricted due to the limited expressiveness of *#pragma* annotations. Metric Description Language (MDL) [19] is not general enough for program analysis (MDL is designed for performance measurement) that supports two types of inserted code only. Besides, DiSL [29] is a domain-specific instrumentation language for handling Java program. Several works put restrictions on PMS. Atune-IL [34] can be used by source-level PMS only. MDL [19], Lynx [15], EBT [27,28], and DTrace [8] are designed for dynamic binary instrumentation, while MAQAO Instrumentation Language (MIL) [9] is for static binary instrumentation.

Several studies restrict the types of insertion points [8,15,19,27,28,30]. For example, [30] does not support instruction-level instrumentation, that would be a serious restriction for application development. The types of insertion points supported by DTrace [8] depend on the instrumentation providers (dubbed PMS in this paper). However, we find that the providers integrated into DTrace are single functional, such as function boundary tracing, statically-defined tracing, locking tracing, probably restricting the applications of DTrace. Several works demand programmers write their applications in the proposed languages [8,15,18,19,30]. For instance, to develop on Sprocket Program Rewriting Interface (SPRI) [18], programmers should write the code in the Sprocket-based Assembly Language. As another example, Dtrace [8] requires programmers to develop applications in the proposed D language.

Some proposed languages lack applications and experiments [27,28,33]. Concretely speaking, Reiss and Renieris [33] proposed the requirements of a general dynamic instrumentation language. However, as they admitted, they had not designed a language that meets the requirements. The EBT language might be immature because the related papers [27,28] did not present any practical applications and experimental results based on it, though a motivating example was given. Several works may result in considerable effort for grasping the features of their proposed languages [9,18]. SPRI is a low-level language, so it may not be that easy to use. For example, programmers have to find the addresses where to insert code through static analysis. MIL is a general language that extends the syntax of Lua [21]. Consequently, the programmers who intend to use MIL should be familiar with the rich language features of Lua.

7 Conclusions

This work designs a middleware that hides the differences of PMS and provides an unified programming interface. Based on it, developers can start to develop concise, PMS-independent and PMS-portable applications quickly. Besides, we build five applications on the middleware for protecting system security and conduct extensive experiments on them. Experiments show that the middleware leads to reasonable space overhead, negligible runtime overhead, and no false positives. Then, two practical cases validate that the applications can prevent real attacks. We plan to improve this work in three directions. First, we are working to integrate more PMS (Valgrind, CIL *etc.*) to give programmers more options. Second, we will enrich the types of insertion points and parameters, allowing programmers to handle the SUA more flexibly. Third, we plan to write more applications on the middleware, so that programmers can develop their applications by reusing our code.

Acknowledgment. This work was supported in part by the National Natural Science Foundation of China, No. 61402080, No. 61572115, No. 61502086, No. 61572109, and China Postdoctoral Science Foundation founded project, No. 2014M562307.

References

1. StackShield: A “stack smashing” technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/
2. SourceCounter, August 2016. http://boomworks.googlecode.com/files/SourceCounter-3.5.33.73.zip
3. Ball, T., Rajamani, S.: The slam project: debugging system software via static analysis. ACM Sigplan Not. **37**(1), 1–3 (2002)
4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX, pp. 41–46, April 2005
5. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools, pp. 9–16. ACM, September 2011

6. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. ACM Sigplan Not. **47**(7), 133–144 (2012)
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, vol. 8, pp. 209–224 (2008)
8. Cantrill, B., Shapiro, M., Leventhal, A.: Dynamic instrumentation of production systems. In: USENIX, pp. 15–28, June 2004
9. Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S., Malony, A., Jalby, W.: MIL: a language to build program analysis tools through static binary instrumentation. In: HiPC, pp. 206–215. IEEE, December 2013
10. Chen, T., Zhang, X., Guo, S., Li, H., Wu, Y.: State of the art: dynamic symbolic execution for automated test generation. Future Gener. Comput. Syst. **29**(7), 1758–1773 (2013)
11. Chipounov, V., Candea, G.: A platform for in-vivo multi-path analysis of software systems. In: ASPLOS, pp. 265–278 (2011)
12. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: ISSTA, pp. 196–206. ACM, July 2007
13. Corbi, T.: Program understanding: challenge for the 1990s. IBM Syst. J. **28**(2), 294–306 (1989)
14. Dinaburg, A., Ruef, A.: Mcsema: static translation of x86 instructions to LLVM. In: ReCon, June 2014
15. Farooqui, N., Kerr, A., Eisenhauer, G., Schwan, K., Yalamanchili, S.: Lynx: a dynamic instrumentation system for data-parallel applications on GPGPU architectures. In: ISPASS, pp. 58–67. IEEE, April 2012
16. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, vol. 3, pp. 191–206, February 2003
17. Godefroid, P., de Halleux, P., Nori, A., Rajamani, S., Schulte, W., Tillmann, N., Levin, M.: Automating software testing using program analysis. IEEE Softw. **25**(5), 30–37 (2008)
18. Hiser, J., Nguyen-Tuong, A., Co, M., Rodes, B., Hall, M., Coleman, C., Knight, J., Davidson, J.: A framework for creating binary rewriting tools (short paper). In: EDCC, pp. 142–145. IEEE, May 2014
19. Hollingsworth, J., Niam, O., Miller, B., Xu, Z., Gonçalves, M., Zheng, L.: MDL: a language and compiler for dynamic program instrumentation. In: PACT, pp. 201–212. IEEE, November 1997
20. Huang, Y., Yu, F., Hang, C., Tsai, C., Lee, D., Kuo, S.: Securing web application code by static analysis and runtime protection. In: WWW, vol. 17, pp. 40–52. ACM, May 2004
21. Ierusalimschy, R., Figueiredo, L.D., Filho, W.C.: Lua-an extensible extension language. Softw. Pract. Exper. **26**(6), 635–652 (1996)
22. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_31
23. Kildall, G.A.: A unified approach to global program optimization. In: POPL, pp. 194–206. ACM, October 1973
24. King, J.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
25. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–86. IEEE, March 2004

26. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. ACM Sigplan Not. **40**(6), 190–200 (2005)
27. Makarov, S., Brown, A.D., Goel, A.: An event-based language for dynamic binary translation frameworks. In: PACT, pp. 499–500. ACM, August 2014
28. Makarov, S., Brown, A.D., Goel, A.: PACT: U: an event-based language for dynamic binary translation frameworks (2015). https://src.acm.org/binaries/content/assets/src/2014/sergueimakarov.pdf
29. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: AOSD, pp. 239–250. ACM, March 2012
30. Mußler, J., Lorenz, D., Wolf, F.: Reducing the overhead of direct application instrumentation using prior static analysis. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6852, pp. 65–76. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23400-2_7
31. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_16
32. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan Notices, vol. 42, no. 6, pp. 89–10, June 2007
33. Reiss, S., Renieris, M.: Lynx: a dynamic instrumentation system for data-parallel applications on GPGPU architectures. In: WODA ICSE, pp. 41–44, May 2003
34. Schaefer, C.A., Pankratius, V., Tichy, W.F.: Atune-IL: an instrumentation language for auto-tuning parallel applications. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 9–20. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_5
35. Sinnadurai, S., Zhao, Q., fai Wong, W.: Transparent runtime shadow stack: protection against malicious return address modifications (2008)
36. Yin, H., Song, D.: Temu: binary code analysis via whole-system layered annotative execution. Technical report UCB/EECS-2010-3, EECS Department, University of California, Berkeley (2010)