CPR: Cross Platform Binary Code Reuse via Platform Independent Trace Program

Yonghwi Kwon
Purdue University
kwon58@cs.purdue.edu

Weihang Wang
Purdue University
wang1315@cs.purdue.edu

Yunhui Zheng
IBM T.J. Watson Research Center
zhengyu@us.ibm.com

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

ABSTRACT

The rapid growth of Internet of Things (IoT) has been created a number of new platforms recently. Unfortunately, such variety of IoT devices causes platform fragmentation which makes software development on such devices challenging. In particular, existing programs cannot be simply reused on such devices as they rely on certain underlying hardware and software interfaces which we call platform dependencies. In this paper, we present CPR, a novel technique that synthesizes a platform independent program from a platform dependent program. Specifically, we leverage an existing system called PIEtrace which can generate a platform independent trace program. The generated trace program is platform independent while it can only reproduce a specific execution path. Hence, we develop an algorithm to merge a set of platform independent trace programs and synthesize a general program that can take multiple inputs. The synthesized platform-independent program is representative of the merged trace programs and the results produced by the program is correct if no exceptions occur. Our evaluation results on 15 real-world applications show that CPR is highly effective on reusing existing binaries across platforms.

CCS CONCEPTS

• **Software and its engineering → Software reverse engineering**; *Maintaining software*;

KEYWORDS

Binary-reuse; Reverse-engineering; Binary-analysis; Cross-platform

ACM Reference format:
Yonghwi Kwon, Weihang Wang, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. CPR: Cross Platform Binary Code Reuse via Platform Independent Trace Program. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092707

ISSTA'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092707

1 INTRODUCTION

Internet of Things (IoT) allows interconnected software, physical systems, electronics, and sensors to deliver unprecedented intelligent capabilities such as smart houses and auto-driving vehicles [33]. As a consequence of the growing deployment of IoT systems, various new platforms [6, 12, 15–17, 21, 24] have been introduced and deployed [2, 3, 18]. More importantly, the pressing demand of software on such platforms is superinduced as well. Unfortunately, despite the need, many platforms suffer from insufficient software support and developing cross platform software is still challenging especially on such IoT devices. This is mainly because unlike traditional desktop platforms such as Windows and Linux, developers have to work with different hardware and low-level software systems. In particular, besides the different low-level software systems (e.g., OS/system library), some devices may even have a different set of underlying hardware (e.g., displays, HCI devices, and storage devices). As a result, *even with source code, handling such differences is very challenging.* In particular, consider a developer implemented a tool that uses some platform specific APIs/libraries. He/she later wants to try the tool on another platform. To do so, the developer should identify and handle all platform dependencies, which is extremely tedious and error-prone.

Binary reusing techniques have been proven to be useful. First, many legacy applications are running on important systems, and their source code is inaccessible. In particular, US government agencies (e.g., DARPA and ONR) asked for techniques that can extract components from binaries "because the application source code is no longer accessible, which means the applications are running on insecure and outdated systems" [8]. Moreover, many complex legacy commercial off-the-shelf (COTS) software often include unwanted or unused components which might be vulnerable [23].

Hence, automatic binary program reusing technique across platforms is highly desirable. In particular, it avoids re-implementing software which is time consuming and error-prone (often introducing new bugs or vulnerabilities) especially on a new or uncommon platform due to the lack of development support such as debugging tools. Even when different platforms share the same programming interface so that a program can be executed on different versions of platforms (e.g., Windows/Linux), a program may behave differently on different platforms due to the subtle implementation differences of underlying libraries [13, 28, 61]. More importantly, binary program reusing can extract and securely reuse useful components from legacy programs which might include unnecessary or vulnerable other components.

Existing Approaches Existing research efforts on reusing binary programs focus on software dependability and security.

Decompilation has been used to recover source code. However, it suffers limitations in practice due to the nature of static analysis. For example, Hey Rays [14] and Boomerang [65] are the state-of-the-art decompilation tools. They often fail to recover indirect jump/call targets (e.g., function pointers). They also have limited capacity to handle obfuscation, hardened or optimized binaries, as hardening and optimizations often break the common compiler idioms they rely on. Moreover, source code recovered by decompilation tools are often not compilable and platform dependent because mapping a platform specific assembly code to a high-level language is extremely difficult, if not impossible.

To overcome the limitations of static analysis, dynamic analysis tools [42, 49, 73] have been proposed to identify, extract, and reuse parts of software. BCR [48] and Inspector Gadget [51] can extract parts of malicious binaries for security analysis. Kim et. al [50] proposed a system that can identify and reuse certain functionalities. However, they either do not support software reuse across platforms or require manual annotations, and hence are not scalable.

Our Approach In this paper, we propose a novel technique to synthesize a platform independent program from a platform dependent program. We develop an algorithm that allows us to merge a set of platform independent trace programs (a.k.a executable traces) produced by PIEtrace [52] from the original platform dependent program and synthesize a fully functional program that has (part of) the functionalities of the original program and possesses the virtue of platform independence. The synthesized program can handle *new* inputs and is guaranteed to be *sound*. Our evaluation shows CPR is highly effective on reusing existing binaries across platforms.

In summary, this paper makes the following contributions:

- We formally define the problem of reusing platform dependent binary programs across multiple platforms. We reduce the problem to a trace merging and resource virtualization problem, and address the challenges to synthesize a platform independent program.
- We propose a novel scheme, *Resource Index*, which represents the state of resources, and use *Resource Index* to virtualize the platform dependent resource accesses.
- We develop a prototype system CPR to extract binary components from 15 real-world Windows programs and reuse them on Linux, Raspberry PI, and Cisco IOS. We show that CPR is highly effective and the synthesized programs are less vulnerable to attacks.

2 DEMONSTRATIVE EXAMPLE

In this section, we use examples to illustrate the challenges in synthesizing platform independent programs by merging a set of platform dependent program executions and motivate our approach.

2.1 Background: *Trace Program*

A *trace program* is a platform independent program generated by PIEtrace [52] via monitoring an execution of a program. It captures the same control flow path and data dependencies along one execution path of the original program. A trace program is platform independent as it does not have any platform dependencies such as system calls and platform specific instructions. In particular, such platform dependencies are replaced with concrete values observed during the execution which we call *Concretized Values*.

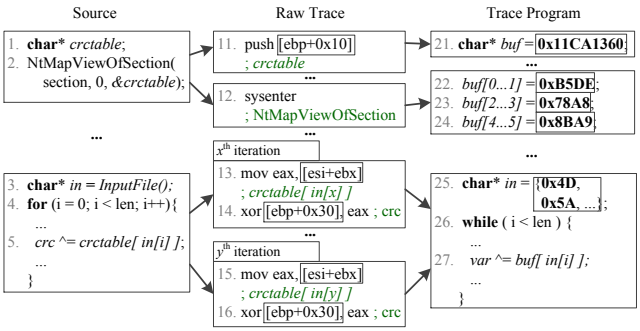


Figure 1: Trace Program and corresponding Source Code and Raw Trace (Boxes connected by lines across columns are correspond to each other).

Fig. 1 shows parts of a trace program generated from an execution of a program.

The column "Source" shows a program that calculates a CRC value. In particular, it calls a platform dependent function (system call) `NtMapViewOfSection()` at Line 2 to map a memory section (a set of memory addresses) into a variable `crctable`. Then it computes a CRC value of an input file and stores the result onto `crc` in a loop at Line 5.

The column "Raw Trace" represents the executed instructions in the original program. `NtMapViewOfSection()` at Line 2 is splitted into two parts: pushing parameters onto the stack at Line 11 and issuing the system call at Line 12. The address of `crctable` is determined and returned by the kernel after the `sysenter` at Line 12. Lines 13-16 represent two instances of the statement at Line 5 in the original program. They access contents of the `crctable` returned by the system call (Lines 13 and 15) and compute xor and store the result onto `crc` (Lines 14 and 16). We only show two iterations of the loop for simplicity.

The column "Trace Program" shows a trace program generated from the concrete execution. Note that the `sysenter` at Line 12 is replaced by concrete values (at Lines 21-24) observed during the execution of Line 5 and Lines 13-16. More importantly, the values of input file at Line 3 are replaced by concrete values at Line 25. By leveraging such concretized values, a trace program can produce the same effects of the system call in a platform agnostic way.

Note that a key technique that makes the trace program platform independent is the concretized values which are used instead of executing actual platform dependent instructions or system calls. However, due to the concretized values, a trace program can only reproduce one recorded execution and cannot take any new inputs rather than the one used in the recorded run. Moreover, it does not have any interface to take an input as all the inputs exist as concretized values in a trace program. We call a program like a trace program that cannot take another input a *closed program*. In contrast, we call a program that can handle any inputs an *open program*.

Hence, in this paper, *we focus on synthesizing a platform independent open program by merging a set of trace programs such that it can take new inputs and produce correct outputs on different platforms.*

2.2 Motivation Example

In this section, we use `Alzip`, a popular file archiver with 1.3M users [5], to show the challenges of merging trace programs and motivate our technique. `Alzip` supports the *egg file format*, which is a new file format designed to enhance the existing zip file format. The egg file format supports Unicode file names and files larger than 2GB/4GB. More importantly, the file format allows `Alzip` to choose different compression algorithms depending on the file type. Specifically, it chooses the best compression method based on the content of the files to be compressed. However, the compatibility of the file format is limited. As `Alzip` only supports a few platforms such as x86/64 Windows/Linux and Android, *the egg file format* cannot be used on other platforms because other file archivers such as 7-zip and gzip do not support egg files. Unfortunately, attackers often use compression to hide malicious files and improve the success rate of attack. Hence, most anti-virus solutions try to inspect common compression file formats [9], Interestingly, we found that most anti-virus programs can detect malware in a zip file, while they fail to inspect an egg file, leaving malware inside egg files undetected. Hence, extracting and reusing the decompression component of `Alzip` is especially useful for Intrusion Detection Systems (IDS) to analyze compressed files and check if malware is present. Therefore, we decide to reuse the component on Cisco IOS [10] which is a popular platform for IDS.

Unfortunately, a decompression module provided by `Alzip` developers does not support *powerpc processors* which most Cisco network products such as routers use. Moreover, while the vendor provided its source code, the hyperlink to the source code is broken [4] as of the time of this paper's writing. Hence, we aim to extract and reuse the *.egg* decompression component in `Alzip` on Cisco IOS, which is not originally supported by the program. Reusing the decompression component on Cisco IOS not only illustrates the challenges of reusing software component across different platforms but also enables a malware detection tool on network routers, which is practically useful.

2.2.1 Approaches. We first present two naive approaches to get an open program from multiple trace programs. Then, we explain the limitations of such approaches and present our approach.

Replacing Concretized Values One way to obtain an open program from a trace program is to replace concretized values, which hold old inputs during the training, with new inputs. However, due to the input changes and non-determinism, *such modified trace program may take a different path* which was not observed during the training hence not included in the trace program, resulting in a crash. For instance, the commonly used inflate algorithm [20] in decompression executes largely different branches depending on the input.

Selecting A Proper Trace Program To handle the path differences due to the input changes, one straightforward approach is adding a set of predicates that selects a proper trace program depending on the input. Fig. 2 shows an example of this approach.

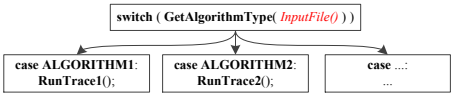


Figure 2: Selecting a Proper Trace Program.

Depending on the input (e.g., an *egg* file), it selects a trace program according to the compression algorithm used in the file. However, such selector requires understanding of the file format to implement `GetAlgorithmType()` which is difficult in practice. More importantly, even with such selector, a single trace program of one particular algorithm may not handle other inputs even if they use a same algorithm. In particular, `Alzip` uses a table-based CRC algorithm. Even with inputs using a same compression algorithm, an execution may take different paths and accesses different portions of the CRC table depending on input values.

Our Approach: CPR Instead of selecting an isolated trace program, we propose a general and sophisticated approach. We combine multiple trace programs that handle different inputs to get an open program. We call such combined program a *merged program*. A salient feature of merged program is that it is not a simple union of all the trace programs collected. It accepts inputs that are well beyond the inputs for the trace programs and can execute along paths different from those represented by the trace programs. It is more like a normal program (with platform independence).

2.2.2 Challenges in Merging Trace Programs. To obtain a merged program which can decompress an egg file, we first collect multiple trace programs by executing `Alzip` on multiple egg files. Note that all platform dependencies are replaced by concretized values in trace programs. However, since each trace program may take a different execution path and required concretized values may be different. Hence, aligning and merging the control flows and concretized values into a merged program are necessary.

Aligning Instructions To allow the same control flow paths of the original executions, a merged program should include all the unique statements from all the trace programs.

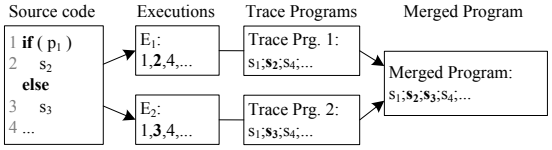


Figure 3: Trace Programs may have different statements.

Fig. 3 shows two executions of a program which has a predicate. One execution (E_1) takes the true branch and the other execution (E_2) takes the false branch. A desired merged program must include all the unique statements (i.e., $\{s_1, s_2, s_3, s_4, \ldots\}$) so that it can reproduce both executions. Note that the merged program should include only one unique statement even if the statement is included in multiple trace programs (e.g., s_1 and s_4) in Fig. 3.

However, aligning and merging statements is challenging as an instruction can have different addresses across runs due to ASLR (Address Space Layout Randomization) which randomizes the address space of loaded executable modules.

Merging Concretized Values Concretized values can be also different in different runs even though they are actually semantically identical. For instance, addresses of heap objects are nondeterministic even without ASLR. Specifically, `Alzip` allocates and accesses a CRC table on heap memory. While values in `crctable` are semantically identical, they may have different absolute addresses across runs. For example, `crctable` can be allocated at address 0x100 and 0x200 across runs. Accessing `crctable[4]` results in

accessing different absolute addresses (e.g., 0x110 and 0x210) while they are semantically accessing same data.

Next, we explain two important sources of such non-determinism on concretized values: *Address Correspondence* and *Resource Driven Concretized Values*.

1) Address Correspondence: Stack variables have different addresses depending on the calling context. Heap objects are allocated on demand non-deterministically. Fig. 4 shows a program which invokes a system call at Line 13 in `D()`. Let one execution (E_1) have $p_1 \equiv true$ and $p_2 \equiv true$, taking a path $A() \rightarrow B() \rightarrow D()$, and another execution (E_2) have $p_1 \equiv false$, resulting in a path $A() \rightarrow C() \rightarrow D()$.

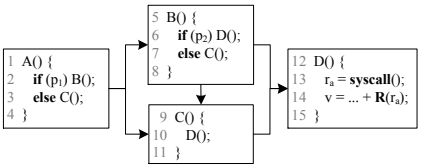


Figure 4: Different Concrete Addresses (r_a at Line 14).

Trace programs from E_1 and E_2 have concretized values on the same stack variable in `D()` at Line 13. However, r_a holds different addresses across the runs due to the different stack layouts caused by different call stacks. Fig. 5 shows two different concretized values of E_1 and E_2 including their stack allocations. Note that if we simply merge the concretized values without appropriate alignment, a concretized value from one execution may overwrite a value of an irrelevant local variable leading to an incorrect execution. For example, in Fig. 5, a concretized value in E_2 (at a_2) will overwrite a local variable v in `D()` in E_1. Note that v in E_1 and r_a in E_2 have the same concrete address 0x12FEF0 as `B()` and `C()` have different frame sizes. The size of `B()` is 0x30 while the size of `C()` is 0x60.

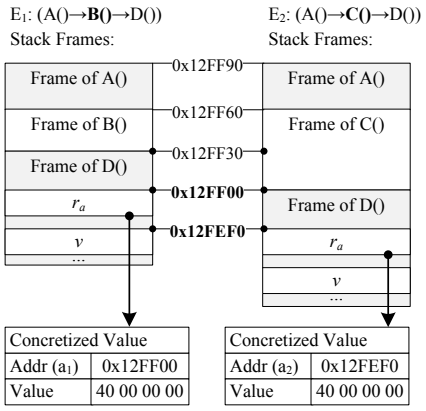


Figure 5: Concretized Values and Stack Allocations.

2) Resource Driven Concretized Values: Concretized values can be different across runs due to the state differences on resources such as file system and network. We call such concretized values *Resource Driven Concretized Values*.

Fig. 6 shows an example. Although two concretized values ($buffer_1$ and $buffer_2$) have the same address (0x12EE00), they have different values as the `ReadFile()` returns different contents depending on the current file position which is adjusted by `SeekFile()` at Line 6. Note that depending on the value of p_1 at Line 4, *offset* can be either

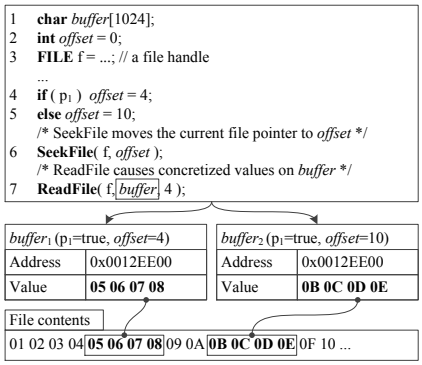


Figure 6: *Resource Driven Concretized Values.*

4 or 10, and then it is used in `SeekFile()` to change the current file position at Line 6.

2.3 Our Approach

Extraction and Merging To extract and reuse a binary program, we first use PIEtrace to get multiple trace programs. We then merge them and synthesize *a merged program*. To do so, we need to overcome the challenges discussed in Sec. 2.2.2. First, to align instructions across trace programs, we use *Symbolic Label* (Sec. 4.1), a consistent label across runs, to index instructions. Second, to merge concretized values, we use *Symbolic Address* (Sec. 4.2) to unify address differences across runs. Third, to handle *Resource Driven Concretized Values*, we propose *Resource Index* (Sec. 4.2.3) that essentially represents the current state of external resources (e.g., file system/network).

Reusing Extracted Component Reusing the merged program is straightforward. Note that a trace program generated by PIEtrace is a set of C source code files [52]. Hence, *the merged program* is also a C program and can be compiled on any platforms. To supply new inputs, we identify input related concretized values by searching trained inputs among the concretized values. Then, we replace them with new buffer variables. Note that this process does not require any knowledge on program specific semantics or interfaces. Similarly, to obtain outputs from the merged program, we identify buffers holding outputs by logging all output system calls (e.g., `write()/send()`) during tracing and then searching for the contents representing the outputs in the log. We mark the corresponding buffer variables as output buffers. In the merged programs, since syscalls are eliminated, outputs are available through the output buffers (in memory) and one can just copy the result after the execution.

3 PROBLEM DEFINITION

In this section, we briefly explain how PIEtrace removes platform dependencies and generates trace programs. Then, we formally define the problem considered in this paper.

3.1 Platform Independent Trace Programs

To extract and reuse components from binaries on different platforms, the extracted components must have no dependencies on the underlying platforms. In general, there are three types of such

dependencies. First, a compiled binary program depends on a certain instruction set architecture (ISA). For example, x86 instructions cannot be executed on ARM processors and vice versa. Second, system calls are dependent on the underlying OS. For instance, Linux and Windows have different system call numbers and parameters. Third, most binary programs depend on underlying libraries such as *libc* which also depend on the underlying platform.

To decouple such dependencies, we leverage PIEtrace, a system that monitors a program execution and generates a platform independent program called a trace program. The trace program can reproduce the same control flow path and data dependencies of the original execution in a platform independent fashion. To facilitate discussion, we introduce a low-level language to model normal and trace programs. The syntax is presented in Fig. 7.

$$
\begin{array}{lll}
\textit{Program} & P ::= & s \\
\textit{Stmt} & s ::= & s_1;\ s_2 \mid r :=^{\ell} e \mid r :=^{\ell} \mathbf{R}(r_a) \mid \mathbf{W}^{\ell}(r_a, r_v) \mid \\
 & & \quad \mathbf{goto}^{\ell}(\ell_1) \mid \mathbf{if}\ (r^{\ell})\ \mathbf{then\ goto}(\ell_1) \mid \\
 & & \quad \mathbf{call}^{\ell}(\ell_1) \mid \mathbf{ret}^{\ell} \mid \mathbf{syscall}^{\ell}() \mid \mathbf{depinst}^{\ell}() \\
\textit{TraceProgram} & \hat{P} ::= & \hat{s} \\
\textit{TPStmt} & \hat{s} ::= & \hat{s}_1;\ \hat{s}_2 \mid r :=^{\ell} e \mid r :=^{\ell} \mathbf{R}(\mathbf{VA}(r_a)) \mid \\
 & & \quad \mathbf{W}^{\ell}(\mathbf{VA}(r_a), r_v) \mid \mathbf{goto}^{\ell}(\mathbf{VL}(\ell_1)) \mid \\
 & & \quad \mathbf{if}\ (r^{\ell})\ \mathbf{then\ goto}(\mathbf{VL}(\ell_1)) \mid \mathbf{call}^{\ell}(\mathbf{VL}(\ell_1)) \mid \\
 & & \quad \mathbf{ret}^{\ell} \mid \mathbf{concretize}^{\ell}() \mid \mathbf{skip} \\
\textit{Expr} & e ::= & r \mid c \mid a \mid r_1\ \mathbf{op}\ r_2 \mid r\ \mathbf{op}\ c \\
\textit{Operator} & op ::= & + \mid - \mid * \mid / \mid \ldots \\
\textit{Register} & r ::= & \{sp, r_1, r_2, r_3, \ldots\} \\
\textit{Const} & c ::= & \{true, false, 0, 1, 2, \ldots\} \\
\textit{Addr} & a ::= & \{0, 1, 2, \ldots\} \\
\textit{Label} & \ell ::= & \{\ell_1, \ell_2, \ell_3, \ldots\}
\end{array}
$$

$$
\begin{array}{l}
VL \in \textit{VirtLabelMap}: Label \rightarrow Label \\
VA \in \textit{VirtAddrMap}: Addr \rightarrow Addr
\end{array}
$$

Figure 7: Language.

Language P is a normal program consists of statements s. Statements include memory read (**R**()) and write (**W**()). Conditional or loop statements can be modeled using **goto** and **guarded goto**. Function invocations and returns are denoted by **call** and **ret**. P depends on the underlying platform as it has **syscall**() and **depinst**(). They represent system calls and platform dependent instructions that are only supported on particular platforms (e.g., SSE/MMX).

$\hat{P}$ denotes a platform independent trace program. It is a sequence of *TPStmts* (Trace Program Statements). Since different platforms have different address space layouts, it virtualizes memory address space, a, including instruction address space, ℓ, so that $\hat{P}$ can run on platforms with different memory layouts. **VA**(a) translates the given address a in the original execution to the virtualized address space. **VL**(ℓ) maps labels (ℓ) of an original program P to the corresponding labels in the trace program $\hat{P}$. **concretize**$^{\ell}$() reproduces the same effect of **syscall**() and **depinst**() by using concrete values observed during execution. Therefore, $\hat{P}$ has no dependencies on the underlying platform.

The definition of **concretize**$^{\ell}$() is shown in Fig. 8. It writes concretized values to memory during the execution of the trace program. In particular, it maintains a hit-count of each ℓ which is **HC**(ℓ). *LS* is a log storage which contains all concretized values which can be accessed by *LogId* which is a unique identifier for each log entry. When a trace program is running, a concretized value e that matches with the current ℓ and its hit-count is applied through **W**(**VA**($e.a$),$e.v$).

$$
\begin{array}{l}
id \in \textit{LogId} : < \ell, z > \\
e \in \textit{LogEntry} : < a, v > \\
HC \in \textit{Instance} : Label \rightarrow \mathbb{Z} \\
LS \in \textit{LogStorage} : LogId \rightarrow LogEntry \\
\\
\mathbf{concretize}^{\ell}() ::= \\
\quad \mathbf{HC}(\ell)\text{++}; \\
\quad e := \mathbf{LS}(< \ell, \mathbf{HC}(\ell) >); \\
\quad \mathbf{if}\ e \neq \varnothing\ \mathbf{then} \\
\quad\quad \mathbf{W}(\mathbf{VA}(e.a), e.v);
\end{array}
$$

Figure 8: Definition of concretize$^{\ell}$().

$$
\begin{array}{lll}
\mathbf{PIEtrace}(\mathbf{Exec}(P, i)) & ::= & \hat{P}_i \\
\mathbf{Exec}(P, i) & ::= & < D_i, V_i, C_i, PD_i > \\
\mathbf{Exec}(\hat{P}_i, \varnothing) & ::= & < D_i, V_i, C_i, \varnothing > \\
\\
\textit{Input} \quad i & ::= & \overline{c} \\
\textit{Values} \quad V & ::= & \overline{c} \\
\textit{PlatformDepOp} \quad dop & ::= & \mathbf{depinst}() \mid \mathbf{syscall}() \\
\textit{Exception} \quad ex & ::= & \hat{s}^{\ell}
\end{array}
$$

$$
\begin{array}{ll}
\textit{Exec} & : \textit{Program} \times \textit{Input} \rightarrow < DD, V, C, PD > \\
\textit{PIEtrace} & : \textit{Exec} \rightarrow \textit{TraceProgram} \\
D \in \textit{DataDep} & : \textit{Register} \rightarrow \textit{Register} \\
C \in \textit{CtrlFlowPath} & : \textit{Label} \rightarrow \textit{Label} \\
PD \in \textit{PlatformDep} & : \textit{PlatformDepOp} \rightarrow \textit{Const}
\end{array}
$$

Figure 9: Definition of Normal and Trace Program Exec.

Note that system calls and platform dependent instructions, **syscall**() and **depinst**(), are concretized in a trace program. Hence, a trace program does not take any inputs. Memory allocations are also concretized and a trace program does not have **malloc**() or **free**(). Specifically, all memory addresses returned by **malloc**() are concretized and all accesses to the allocated memory blocks are redirected to the virtualized address space via **VA**(). PIEtrace traces into the library calls and generates corresponding platform independent statements and concretized values too. *sp* denotes the stack pointer register (e.g., `esp` on x86) and we assume all platforms have such a register.

In Fig. 9, **Exec**(P, i) represents an execution of a given program P with an input i. The input i is defined as a sequence of constants $\overline{c}$. The output of the execution includes data dependencies (D_i), values (V_i), control flow path (C_i), and platform dependencies (PD_i), where PD represent invocations of *PlatformDepOp*.

Exec($\hat{P}_i$, $\varnothing$) represents an execution of a trace program $\hat{P}_i$. It does not take any input and reproduces the same data dependencies (D_i), values (V_i), control flow path (C_i) as those produced by the original execution (**Exec**(P, i)). Note that it is platform independent as it does not have platform dependencies (PD). We further define PIEtrace as a function **PIEtrace**() which takes an *execution of a normal program* P and generates a platform independent *trace program* $\hat{P}$.

3.2 Problem Definition

While a trace program $\hat{P}$ is platform independent, it can only reproduce a single execution and cannot handle new inputs. Therefore, our goal is to synthesize a *merged program* $\hat{P}_m$, which can accept multiple inputs, by merging multiple trace programs $\hat{P}$. Ideally, the merged program can take any possible input i and produce the same outputs (i.e., data dependencies (D_i), values (V_i), and control flow path (C_i)) as the original program P does. Theoretically, we can obtain such ideal merged program by merging trace programs generated under all possible inputs. However, it is difficult to cover

all possible inputs as reasoning about all inputs is extremely difficult in practice.

Problem Statement: Instead, we aim to synthesize a merged program $\hat{P}_m$ that can process all the inputs $\{i_1, i_2, ..., i_n\}$ observed in the trace programs, and a set of new inputs $\{i_{n+1}, i_{n+2}, ..., i_{n+m}\}$. $\hat{P}_m$ can correctly produce corresponding outputs (D_i, V_i and C_i), as shown in Fig. 10.

$\textbf{Exec}(\hat{P}_m, i') ::= < D_{i'}, V_{i'}, C_{i'}, \varnothing > \mid ex$
where $\hat{P}_m$ is a result of merging $\{\hat{P}_{i_1}, \hat{P}_{i_2}, \ldots, \hat{P}_{i_n}\}$,
$i' \in Input ::= \{i_1, i_2, ..., i_n\} \cup \{i_{n+1}, i_{n+2}, ..., i_{n+m}\}$, and
$\textbf{Exec}(P, i') ::= < D_{i'}, V_{i'}, C_{i'}, PD_{i'} >$

Figure 10: Problem Statement: Merged Program in Practice.

Note that input i' also includes a set of inputs $\{i_{n+1}, i_{n+2}, ..., i_{n+m}\}$ that are not observed in the trace programs. These unseen inputs should not drive the program to uncovered statements or different resource states. In fact, we observe many of such new inputs which can be handled by merged programs (Sec. 5.1) and show the empirical results in the evaluation (Sec. 5). However, it is possible the merged program may take an input beyond this scope. In this case, $\textbf{Exec}(\hat{P}_m, i')$ should throw an exception *ex* to indicate $\hat{P}_m$ cannot handle the input, rather than continuing the execution and producing an incorrect outcome. In other words, we *guarantee* $\hat{P}_m$ *is sound*: $\textbf{Exec}(\hat{P}_m, i')$ either produces the correct outputs as the original program does or throws an exception indicating a merged program cannot handle a given input. Since supporting all possible inputs is hard or even impossible, we take a practical approach and synthesize merged programs on demand. In particular, a merged program produces a correct output or an exception with an explanation which indicates how to collect additional trace program to merge. For instance, if an exception is caused, CPR suggests collecting another trace program with the input that caused the exception. Later, when the new trace program is merged, CPR checks whether the exception is resolved or not, guarantees the new merged program handles on the previous exception.

4 CPR DESIGN

In this section, we formally discuss how we overcome the challenges of combining trace programs.

4.1 Merging Statements

To merge statements correctly, we first need to identify unique statements across trace programs. While each statement is associated with a unique label (ℓ) in each trace program, it is not unique across different trace programs. Thus, we introduce *Symbolic Label* to unify labels across trace programs. Specifically, *Symbolic Label* is in the form of < *Module*, *Offset* > in x86 Windows/Linux. It is used instead of a concrete instruction address (ℓ) to identify unique statements (instructions) across trace programs.

Fig. 11 shows a procedure **MergeStatements**() for statement merging. It merges a new statement $\hat{s}_i^{\ell}$ into the trace program $\hat{P}_i$, where $\hat{s}_i^{\ell}$ represents a statement $\hat{s}$ in a trace program $\hat{P}_i$ with a label ℓ. As label ℓ cannot be used to index a statement across trace programs, we introduce **SL**() to map label ℓ to its *Symbolic Label*. For example, if two statement labels $\hat{s}^i \in \hat{P}_x$ and $\hat{s}^j \in \hat{P}_y$ satisfy $\textbf{SL}_x(i) \equiv \textbf{SL}_y(j)$, the two statements are same ($\hat{s}_x^i \equiv \hat{s}_y^j$) even though they may have different labels (i.e. $\textbf{VL}_x(\hat{s}_x^i) \neq \textbf{VL}_y(\hat{s}_y^j)$).

Once statements across trace programs are unique identified through *Symbolic Label*, merging is straightforward: we add a statement $\hat{s}^{\ell}$ to a merged program $\hat{P}_m$ if the statement is not yet included.

Module $m ::= \{m \mid m$ is a unique hash value for each module.$\}$
Offset $o ::= \mathbb{Z}$
SymLabel $sl ::= \{< m_1, o_1 >, < m_2, o_2 >, \ldots\}$
$SL \in SymLabelMap : Label \rightarrow SymLabel$

MergeStatements($\hat{P}_m$, $\hat{s}_i^{\ell}$)::=
for each $\hat{s}^j \in \hat{P}_m$ **do** where j represents a label of each statement
if $SL_m(\ell) \equiv SL_i(j)$ **then**
return $\hat{P}_m$;
$VL(\ell)$= **GetLabel**(ℓ);
$SL_m(\ell) = SL_i(j)$;
$\hat{P}_m := \hat{P}_m \cdot \hat{s}_i^{\ell}$;
return $\hat{P}_m$;

Figure 11: Algorithm for Merging Statements.

4.2 Merging Concretized Values

As discussed in Sec. 2.2, we also handle the challenges introduced by *Address Correspondences* and *Resource Driven Concretized Values*. In particular, we introduce *Symbolic Address* which provides unique addresses for variables across runs and deploy an abstraction layer of *external resources* to unify the concretized values caused by different resource states across trace programs.

4.2.1 Handling Address Correspondence. Similar to *Symbolic Label* (Sec. 4.1) for statements, we introduce *Symbolic Address*, a relative address that is stable across executions. We replace absolute addresses in concretized values with the symbolic addresses. Specifically, an address of a stack variable is translated into a pair of a function and an offset (e.g., $\langle Func, 0x10 \rangle$). An address of a global variable is also translated into a pair of a module and an offset (e.g., $\langle main.exe, 0x1000 \rangle$). Handling heap variables is more challenging as heap memory addresses are non-deterministic. We observe that the concretized values on heap variables are always caused by external resources (e.g., through system calls). Hence, we identify and encode the current state of external resources and associate the state with the concretized values. Details can be found in Sec. 4.2.3.

4.2.2 Handling External Resources. We introduce a novel concept that abstracts away platform-specific interfaces or internal semantics of external resources. As interactions between a program and external resources are done through system calls, values from external resources are concretized in a trace program. For example, file contents returned by `ReadFile()` become concretized values in a trace program. As shown in Fig. 6 in Sec. 2.2.2, different trace programs may have different concretized values even from a same resource (e.g., a same file), depending on the state of the resource. To abstract such state of external resources, we introduce *Resource Index (RI)* which abstracts states of external resources based on interactions between a program and external resources. A key observation of RI is that a state of a resource is essentially a history of previous interactions on the resource, which represents how

previous system calls affected the state of the resource. In other words, a state of a resource is deterministic as long as the resource has gone through a same set of operations (e.g., system calls). Note that there are some exceptions such as a state of a external server. In this paper, we only focus on local resources such as files, memory, and local libraries.

Essentially, we first annotate concretized values with RIs, and merge them by associated RIs. A merged program also contains routines to update the current RI. Note that while a merged program does not execute system calls, it will update the current RI accordingly. When a concretized value is required, a concretized value associated with the current RI is returned to the program in order to faithfully reproduce the access in a normal program. If the concretized value required is not present, the merged program throws an exception to indicate an unobserved RI state is reached. Thus, we guarantee that the merged program handles an input correctly if no exception is observed.

4.2.3 Resource Index. When generating a trace program, a concretized value is produced only when a program reads a value which is not computed by the program itself. To simplify the discussion, we assume that a program can only access the external resources through system calls. Note that, however, other methods such as interrupts or instructions like `in` and `out` can be handled in the same way.

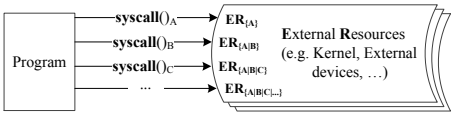


Figure 12: Capturing the State of External Resource. $ER_{\{x\}}$ represents that an External Resource with state x.

The key property that makes external resource accesses different from memory accesses is that it is stateful. For example, a file read may return completely different results depending on previous accesses. Fig. 12 shows an example. The return of **syscall**()$_C$ may depend on previous system calls (**syscall**()$_A$ and **syscall**()$_B$). We argue that *the same request must return the same values if the external resources are in the same state* and we propose *Resource Index (RI)* to represent the operation history and identify states of external resources.

As state changes are driven by system calls, we model the state of the external resources as a sequence of system calls that mutate their states. Hence, RI is essentially a sequence of executed system calls on each resource at a point. Note that RI does not include system calls which do not affect the state of the external resources such as `printf()`. A system call invocation is represented as a *Resource ID* (*ResID*) which contains a system call number and parameters of the system call (i.e., $ResId := \langle SysNum,\ r_1,\ r_2,\ \ldots,\ r_n \rangle$, where r_i denotes the arguments). RI is essentially a sequence of the *Resource ID* of each resource. In other words, *ResID* is added to the current RI on every system call invocation. Note that different resources (e.g., different files) maintain separate *ResID* so that a state of a resource (e.g., $ResID_A$) does not interfere with a state of another resource (e.g., $ResID_B$).

4.2.4 Concretized Value ID. Similarly, to merge concretized values, we need to identify unique concretized values across trace programs. We introduce *Concretized Value ID (CID)* for this purpose, which is defined in Fig. 13.

As discussed in Sec. 4.2.3, PIEtrace generates concretized values only when they are not computed by the program itself. Interestingly, it turns out that there are only two types of sources of concretized values. First, constant values can be accessed during the execution. Second, they are from external resources (e.g., file system, network, timer, motherboard, kernel and etc.). For the first case, constant values are not dependent on external resources and can be easily inlined. Hence, aligning them only requires *Symbolic Address*. For the second case, we need to identify places where the concretized values are generated and the states of the external resources at the time of the generation of the concretized values. As a result, *CID* consists of two elements: *SymbolicAddress* and *ResourceIndex*, where *SymbolicAddress* aligns the constant values and *ResourceIndex* represents the current states of external resources.

Fig. 13 shows the updated definition of the concretized value. Compared to those of a trace program (Fig. 8), the differences are listed as follows. *LogId* is replaced by *CID* and $LogStorage'$ which maps *LogId* to a concretized value is also updated accordingly. More importantly, **concretize**$^{\ell}(a)$ takes as input an address of the concretized value (a).

In particular, **concretize**$^{\ell}(a)$ first gets the corresponding CID of a given address via **GetCID**(a). Note that the same algorithm is also used to obtain a CID from an address during the original execution when PIEtrace generates a trace program. Then, CID is used to look up the $LogStorage'$. Since CID includes a RI value, the look up operation is sensitive to the states of external resources . Finally, the returned value from LS' ($LogStorage'$), which is v, is applied to the given address a.

$SyscallNumber$ $SysNum ::= \mathbb{Z}$
$ResourceIndex$ $ResID ::= < SysNum, \overline{c} >$
$ConcretizedValueID$ $CID ::= < SymAddr(a), ResID >$
$LS' \in LogStorage'$ $: CID \rightarrow Value$

SymAddr(a) returns a symbolic address of the given address.
GetCID(a) returns a ConcretizedValueID from the given address.
SM(a) returns a value on the given address from a shadow memory.
M(a) returns a value on the given address from an actual memory.

LogRead(ℓ, a)::=
  if SM(a)≠ M(a) **then**
    LS'(GetCID(a)):= M(a);
  SM(a):= M(a);

concretize$^{\ell}(a)$::=
  $v := LS'$(GetCID(a));
  if $v \neq \emptyset$ **then**
    W($VA(a)$, v);

Figure 13: Definitions for symbolization

4.3 Safety Net

A merged program should throw an exception *ex* (Fig. 10) if it cannot handle an input correctly. This is the key to making the design sound. Besides, exception is a good indicator that more trace programs are needed as more execution states should be covered. In this section, we explain the design of *Safety Net*, which guarantees that a merged program cannot report incorrect execution results.

4.3.1 Missing Statements. A merged program may miss statements compared to the original program, as our technique is based on dynamic traces. To be safe, we replace such statements with exception triggering statements. Fig. 14 shows an example. Lines

3-4 in the original program are missing in the merged program (Lines 13-14). We add statements which throw exceptions instead of the original statement so that we can detect attempts to execute the missing statement.

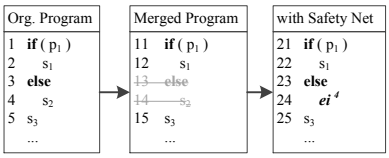


Figure 14: Replacing Missing Statements with Exceptions.

4.3.2 Missing Concretized Values. Even a merged program includes all statements from the original program, missing concretized values can still lead to incorrect operations and erroneous outputs. Thus, we monitor how concretized values are requested and used by a merged program. Intuitively, we check if a merged program requests unobserved parts of external resources at runtime.

In particular, a merged program should throw an exception when it requests an unobserved RI value, as we do not know the value corresponding to the uncovered state. Besides, it may also request a partially concretized buffer. For example, the merged program for Fig. 6 may want to read `buffer[10]` which we do not know its value. To handle such scenarios, we mark the whole buffer *Invalid* first and only update the concretized portion *Valid*. This allows us to throw an exception if an *Invalid* portion is requested. In Fig. 6, only `buffer[0,3]` have concretized values while `buffer[4,1023]` are not. Hence, `buffer[4,1023]` is marked *Invalid*. If a merged program reads `buffer[4,1024]`, an exception will be thrown.

5 EVALUATION

We apply CPR on 12 Windows applications to reuse them on 3 different platforms, Linux (x86 and x64), Raspberry PI and Cisco IOS. We also apply CPR on 3 vulnerable Windows applications to show that the extracted merged programs are resilient to attacks.

Table 1 shows the benchmark programs and the components to be extracted. The first two columns show program name and size of the binary. The third column shows # of trace programs needed to handle all test inputs. The next column represents # of inputs we prepared for each program. The next four columns show # of functions, # of instructions, # of concretized values extracted. The coverage means how many instructions are included in a merged program within the extracted functions. Note that this briefly captures the coverage of the extracted component. The overhead column shows runtime performance and size of the merged programs. We compare the execution time of the original programs and the merged programs. The average performance overhead is 142.2% meaning that the merged programs are approximately 2.4 times slower than the original programs. For the size, we measure size of compiled binary of the merged programs. We measure the overhead on the same machine (Intel Pentium 2.70GHz, 8GB RAM, x86 Windows 7) for fair comparison. We use Visual Studio 2010 to compile the merged programs. The results show that the size of merged programs are fairly small (mostly less than 100KB) which we believe reasonable. Note that trace programs generated by PIEtrace are very slow (more than 20x times overhead) and large (often more than few megabytes). Hence, we develop and apply several optimizations on the merged program to reduce the overhead in both runtime and size. In particular, we inline constant concretized values and translate virtualized instructions into corresponding high-level language idioms. Note that a merged program after such optimizations is still platform independent. By applying some of platform dependent optimizations (e.g., replacing commonly used logics with available library calls on a specific platform), we may reduce the size and runtime overhead even more. The last column shows the target component/algorithm we extracted from each program. `Alzip` is a file archiver. It supports the *egg* file format and it can select an optimum compression algorithm depending on file contents. We extract its decompression component with several decompression algorithms. `AutoHotKey` and `SAScript` are scripting language engines. We use CPR to extract and reuse the language engines so that we can run scripts on a different platform. We want to reuse the file parsing components in both `JPlayer` and `OutlookAddressBookView`. `JPlayer` displays videos recorded by a dash-cam device. `OutlookAddressBookView` extracts recipients information from a Microsoft Outlook Address Book file. We extract the steganography algorithms in `PngSteg` and `HideIt` that are used to hide information in images. `Pngrewrite` compresses png files using various tricks and we want to extract the compression component. `ScriptDecoder` is a source code deobfuscator for Visual Basic Scripts and `Strings` identifies any string inside an executable file. `NggolekiGinambaran` is an image retrieval program that compares two images and shows their similarity score. We extract the image retrieval algorithms in the program. `Entropy` calculates entropy of an executable file to determine if it is malicious. In addition, we have 3 vulnerable programs, `CastRipper`, `FreeAmp` and `PowerTabEditor`. They are vulnerable to memory corruption issues when reading input files. We use CPR to extract components that parse input files, and show that the extracted components are no longer vulnerable.

5.1 Case Studies

5.1.1 Alzip. We collect a set of trace programs by running Alzip on 100 input files. Table 2 shows the file types.

Fig. 15 shows how many trace programs are needed to handle 100 different egg files. X-axis shows the number of trace programs merged and Y-axis represents the number of egg files it can handle. The result shows that after we merge 16 trace programs, the merged program can handle all 100 egg files. The number of trace programs (and inputs) needed is far smaller than collected trace programs. This result also echoes the problem definition in Sec. 3.2 that is merging some of trace programs (16 trace programs) can handle new inputs which were not observed during the merged executions.

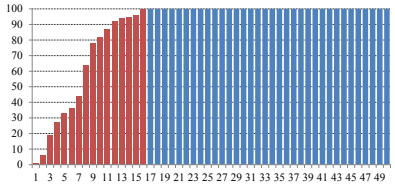


Figure 15: # of Inputs handled by Merged Programs.

To reuse a merged program, we scan the concretized values and match them with original input egg files' contents. We replace the matched concretized values with a new buffer variable so that it can

Table 1: Evaluation Results.

Program	Size	# of trace programs	# of Inputs	Merged Programs				Overhead		Target
				# of Func.	# of Inst.	# of Concr. Val.	Coverage	Runtime	Size	Component/Algorithm
Alzip	2.8 MB	16	100	15,350	483,840	39,477	73.2%	193%	241 KB	Decompression Algos.
AutoHotKey	895 KB	9	50	11,154	368,154	35,900	75.8%	235%	132 KB	Script Language Engine
SAScript	88 KB	11	40	5,223	213,012	6,617	84.0%	223%	81 KB	Script Language Engine
JPlayer	5.1 MB	4	20	7,708	347,260	10,274	71.5%	121%	78 KB	File Format Parsing
OutlookAddressBookView	102 KB	3	10	4,341	131,340	2,956	84.7%	187%	69 KB	File Format Parsing
Pngrewrite	171 KB	12	100	6,318	227,442	15,044	78.0%	133%	50 KB	Image Size Optimization
PngSteg	129 KB	7	100	3,392	120,016	5,009	82.2%	93%	48 KB	Steganography Algos.
HideIt	28 KB	5	100	2,592	82,330	2,823	85.9%	84%	50 KB	Steganography Algos.
ScriptDecoder	44 KB	7	100	2,296	102,042	1,691	88.4%	198%	48 KB	Script Decoding Algos.
Strings	21 KB	4	100	2,047	67,230	9,048	83.1%	102%	30 KB	String Identification
NggolekiGinambaran	976 KB	9	100	5,138	127,036	10,764	87.3%	167%	52 KB	Image Retrieval Algos.
Entropy	26 KB	7	100	1,685	78,374	1,812	93.1%	82%	24 KB	Entropy Calculation
CastRipper	1.4 MB	4	50	2,862	127,245	2,690	91.7%	92%	28 KB	(Vulnerable) File Parsing
FreeAmp	2.2 MB	3	50	2,744	129,578	3,404	95.4%	117%	28 KB	(Vulnerable) File Parsing
PowerTabEditor	2.2 MB	4	20	3,590	142,732	3,136	83.8%	106%	30 KB	(Vulnerable) File Parsing

Table 2: Sample input files for different traces.

Category	Extension	Samp.	Category	Extension	Samp.
Text	.txt, ini, .reg, xml	12	Source code	.c, .h, .cpp, .htm	12
Binary	.dat, .bin, .img, .vbe	13	Executable	.exe	12
Library	.sys, .dll, .a	12	Document	.pdf, .docx, .xlsx,	13
Image	.bmp, .jpg, .png	13	Multimedia	.wav, .mp3, .avi, .mpg	13

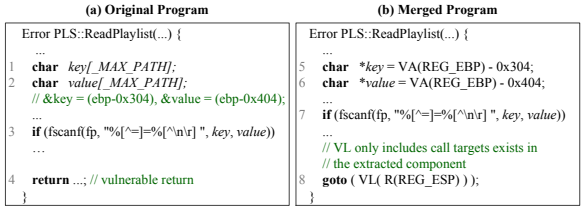
provide new input, and feed the buffer to `ReadFile()`. Similarly, we log all outputs during the execution and identify the output from the log. In particular, we capture the parameter of `NtWriteFile()` in order to get outputs from the merged program.

5.1.2 AutoHotKey and Texter. We use `AutoHotKey` to show how CPR can extract and reuse its script language engine. `AutoHotKey` is an interpreter that automates many tasks. The program is listed as one of the top 10 useful Windows applications that should be ported to Mac by lifehacker.com [26]. There are many projects using `AutoHotKey`. In particular, `Texter` [19] is a text substitution program that replaces abbreviations with user-defined phrases. We use CPR to extract the script language engine and reuse it on Linux to run `Texter`. Specifically, we first pick 50 different AutoHotKey script programs that process keyboard inputs on the Internet [1, 7, 25]. We get 50 trace programs from them and we merge the traces. Interestingly, as shown in Table 1 that we found merging 9 trace programs is enough to run the 50 different script programs. To reuse the extracted component, we replace the API calls related to keyboard and messages (e.g., `keybd_event`, `SetKeyboardState`, and `PostMessage`) as well as a file read operation (e.g., `ReadFile`). To this end, we successfully ran the `Texter` program on the merged program on Linux.

5.1.3 Reusing and Fixing Vulnerable Component. We use CPR to extract a file parsing component from `FreeAmp` which has a memory corruption vulnerability to show that the extracted component is resilient to attacks. `FreeAmp` is a music player and it accepts several types of playlist files. However, a function that processes playlists is vulnerable to code injection attacks. We use CPR to extract the vulnerable file parsing function and check if the extracted component is vulnerable to similar exploits.

First, we get 50 trace programs. Note that we use fairly big benign inputs (>100KB) to cover most instructions. It turns out that we reached the fixed point after merging 3 trace programs. Then, we feed exploits to the merged program and compare the execution with the original program.

Fig. 16 (a) and Fig. 16 (b) show how the original and merged program behave. Note that the `return` statement is replaced with



(c) Stack of the Merged Program (Benign)

VA(Address)	Value	VL(Value)
0x025FFD0C	0x03AE2680	Undefined
0x025FFD10	0x025FFCFC	Undefined
...	...	...
0x025FFE28	0x120415F4	*Func_415F4*

(d) Stack of the Merged Program (Attack)

VA(Address)	Value	VL(Value)
0x025FFD0C	0x03AE2729	Undefined
0x025FFD10	0x025FFD80	Undefined
...	...	...
0x025FFE28	0x025FFCA4	*Undefined*

Figure 16: Merged Program under the Buffer Overflow.

goto with **VL** (Fig. 7). Fig. 16 (c) shows the stack during the benign execution. In particular, the fourth row in Fig. 16 (c) shows a return address. The first two columns show the stack addresses and values stored on the stack. The last column shows whether the value stored on the stack can be translated to another label (e.g., call/jump target). Note that in Fig. 16 (c), the return address (e.g., 0x120415F4) actually points to a valid code block (e.g., *Func_415F4()*).

On the other hand, Fig. 16 (d) shows the stack under the attack. Note that the fourth row of the stack (the return address) is different. Specifically, the value stored on the stack is 0x025FFCA4 which points to injected payload. Since such code never exists while we collect the trace programs, **VL** of the value is *Undefined*. Since the jump target cannot be resolved, the attack is not successful.

In addition, we further construct ROP attacks and test them on the both original and merged programs. The result shows that the merged program is not vulnerable to most ROP attacks as it rejects most of ROP gadgets as they are not part of the merged program anymore.

5.1.4 Extracting Component from Infected Programs. Attackers often inject malicious payloads into a useful benign program and allure users to run the infected program. In this case study, we use CPR to extract a component from an infected application and show that the extracted component is free of malicious payloads.

We pick `OutlookAddressBookView` [27] that can extract recipient info from the Microsoft Outlook address book files. We use `Metasploit` to inject a malicious payload (Reverse TCP) into the program using `msfvenom`. Then, we collect several trace programs and merge them. To reuse the extracted component, we replace file related functions with the corresponding functions available in the

target platform. Since we want to make sure there is no remote attack while collecting trace programs, we disabled the network.

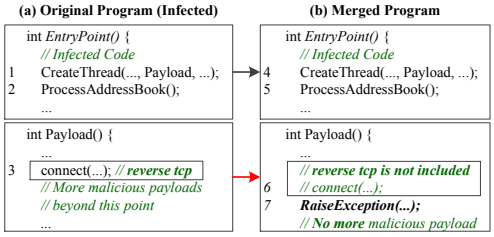


Figure 17: Merged Program under the Infection

Fig. 17 (a) shows the infected original program. It creates a thread that executes the malicious payload at Line 1. The malicious payload decodes the encrypted payload and eventually calls `connect()` to establish a reverse TCP connection.

Fig. 17 (b) shows the extracted component from the infected original program. At Line 6, `connect()` function that opens a reverse TCP connection does not exist. This is because trace programs do not have platform dependent system calls including network APIs.

Moreover, even if it includes the `connect()`, the attack cannot be launched. This is because the merged program does not have any malicious code beyond the `connect()` as the network connection was never established. Hence, any code beyond the function is not added to any trace programs. Note that we disabled the network connection while collecting trace programs. Thus, any further execution will throw exceptions (Line 7).

6 RELATED WORK

Binary Extraction and Reuse Inspector gadget [51] and BCR [48] extracts a part of a program for security analysis and code reuse. The extracted gadgets and components can take inputs to perform certain functionalities. However, they are platform dependent hence cannot be reused on different platforms. TOP [73] is a dynamic analysis based technique that decompiles a binary to C program. The resulting C program can also take inputs as the original binary does. However, it requires the presence of the same environment such as the same set of libraries and kernel interfaces, hence it is platform dependent. Virtuoso [42] automatically generates introspection tools for security applications. The generated tools are python programs hence platform independent. However, it requires a driver application in order to specify functions and variables to extract. In practice, writing such a driver for real-world binary programs due to the lack of semantic information. In contrast, CPR synthesizes a platform independent program without any knowledge on the target software, components, and platform.

Decompilation/Disassembler Decompilers [14, 59, 65] reconstruct program source code from the lower-level languages [38]. However, most of them are based on static analysis, and they often fail to recover non-trivial indirect jump/call targets such as function pointers. Moreover, they have limited capacity to handle obfuscation techniques, hardened binaries or optimized binaries as such techniques often break common compiler idioms. When they cannot find a high-level language representation of the given binary code, they often generate *platform dependent assembly code.* While most disassemblers [14, 39, 59] focus on producing code for analysis purpose meaning that it cannot be directly recompiled, there have been several attempts to generate IR (e.g., LLVM IR) from binaries [11, 22, 31]. However, these tools are not mature enough to generate recompilable IR code. Recently, Wang et al. propose a reassemblable disassembler [68]. However, the disassembled program still have platform dependencies such as system calls. In contrast, CPR uses dynamic analysis and synthesizes platform independent programs even from obfuscated/hardened/optimized binaries.

Software Transplantation Recently, Harman et al. [47] have introduced the idea of software transplantation based on Genetic Improvement (GI) [32, 45, 46, 53, 63, 70] which improves existing systems by manipulating program code. There are also some attempts [56, 69] to transplant code from one location to another within the same system (e.g., different versions of a system). Recently, Earl Barr et al. [35] transplanted code from one system to different systems. It combines dynamic and static analysis to identify program dependencies of the transplanted functionality. However, they require source code which is often not available in practice. More importantly, even if source code is available, a program may use closed sourced libraries and/or up-to-date language features (e.g., lambda expressions), which are challenging to handle for source code based approaches. Instead, CPR works directly on binary programs to automatically identify and decouple platform dependencies even inside the closed source libraries.

Execution Replication and Replay Execution replication and replay has been widely studied [30, 37, 40, 43, 44, 55, 58, 60, 62, 66, 72]. However, most of them do not support cross platform or require knowledge on target platforms (e.g., system call interface). Moreover, they can only replay a specific execution path and do not allow new inputs. In contrast, CPR synthesizes a platform independent open program which can take new inputs. Xu et al. [71] proposed a compiler based technique to generate two instrumented versions of a Java program for logging and replay. However, it assumes the availability of source code and the same set of libraries. Instead, CPR directly works on binaries and does not require any libraries.

Virtual Machine, Emulation, and Runtime Support Virtual Machines [34, 54, 67] and Emulators [36, 41, 57, 64] can run applications from different platforms. However, they often require hardware support or an entire software stack including OS. Moreover, to reuse a program inside the VM, one needs to know interfaces of target OS and VM. Docker [54] is fast and handy to run an application on different platforms. However, it currently supports only few desktop OSes. WINE [29] provides a compatibility layer between Windows applications and POSIX compliant OSes. However, it can only run programs on the same processor. CPR does not rely on any knowledge on the target platforms and the synthesized programs can be reused on any platform even on different processors.

7 CONCLUSION

In this paper, we proposed a novel approach CPR that can generate a platform-independent program from a platform-dependent program. In particular, we overcome technical challenges introduced by un-unified differences across executions and merge a set of platform independent trace programs generated by PIEtrace. The merged program is representative of the observed program runs. Our evaluation on 15 real-world Windows programs shows that CPR is highly effective and the generated programs are often more secure.

REFERENCES

[1] 1 hour software by skrommel - donationcoder.com. http://www.donationcoder.com/Software/Skrommel/.

[2] 10 enterprise internet of things deployments with actual results. http://www.networkworld.com/article/2848714/cisco-subnet/10-enterprise-internet-of-things-deployments-with-actual-results.html.

[3] 11 amazing success stories to prove that internet of things (iot) is not just a verbal tic. https://www.linkedin.com/pulse/11-amazing-success-stories-prove-internet-things-iot-just-sambhani.

[4] 404 - file or directory not found. http://www.altools.com/al/downloads/egg_module/unegg_v0.5.tar.bz.

[5] Alzip - cute & easy file compression program - altools. http://www.altools.com/altools/alzip.aspx.

[6] Arduino. https://www.arduino.cc/.

[7] Autohotkey script showcase. https://autohotkey.com/docs/scripts/.

[8] Binary executable transforms (bet). https://opencatalog.darpa.mil/BET.html.

[9] Bypassing malware defenses. https://www.sans.org/reading-room/whitepapers/testing/bypassing-malware-defenses-33378.

[10] Cisco ios technologies. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html.

[11] Dagger. http://dagger.repzret.org/.

[12] Dronecode. https://www.dronecode.org/.

[13] Findfirstfile behaves differently on vista. http://www.yqcomputer.com/1147_3324_1.htm.

[14] Hex-rays. ida pro disassembler. https://www.hex-rays.com/idapro.

[15] Intel(r)-based drone technology pushes boundaries. http://www.intel.com/content/www/us/en/technology-innovation/aerial-technology-overview.html.

[16] Intel(r) galileo gen 2. http://www.intel.com/content/www/us/en/embedded/products/galileo/galileo-overview.html.

[17] Intel(r) iot platform. http://www.intel.com/content/www/us/en/internet-of-things/infographics/iot-platform-infographic.html.

[18] Internet of things: Why iot is here to stay within the enterprise. http://blogs.air-watch.com/2015/11/internet-things-iot-enterprise/#.V79OzlsrJUQ.

[19] Lifehacker code: Texter (windows). http://lifehacker.com/238306/lifehacker-code-texter-windows.

[20] Linux cross reference - inflate.c source code. http://lxr.free-electrons.com/source/lib/inflate.c.

[21] mbed iot device platform. https://www.arm.com/products/internet-of-things-solutions/mbed-IoT-device-platform.php.

[22] Mc-semantics. .https://github.com/trailofbits/mcsema.

[23] Onr baa announcement # n00014-17-s-b010. https://www.onr.navy.mil/-/media/Files/Funding-Announcements/BAA/2017/N00014-17-S-B010.ashx.

[24] Raspberry pi. https://www.raspberrypi.org/.

[25] Scripts and functions - autohotkey community. https://autohotkey.com/boards/viewforum.php?f=6.

[26] Top 10 windows applications that should be on macs. http://lifehacker.com/5567174/top-10-windows-applications-that-should-be-on-macs.

[27] View / export the address book of ms-outlook. http://www.nirsoft.net/utils/outlook_address_book_view.html.

[28] Why does this code work on windows 7, but doesn't on windows xp? http://stackoverflow.com/questions/12638698/why-does-this-code-work-on-windows-7-but-doesnt-on-windows-xp.

[29] Winehq - run windows applications on linux, bsd, solaris and mac os x. https://www.winehq.org/.

[30] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 193–206, New York, NY, USA, 2009. ACM.

[31] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.

[32] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168, June 2008.

[33] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, Oct. 2010.

[34] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[35] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM.

[36] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[37] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[38] P. T. Breuer and J. P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5):1613–1647, Sept. 1994.

[39] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[40] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[41] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, 22(1):128–137, May 1994.

[42] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[43] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.

[44] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.

[45] C. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, Sept. 2013.

[46] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 1–14, New York, NY, USA, 2012. ACM.

[47] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for reverse engineering. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 1–10, Oct 2013.

[48] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society.

[49] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu. Dual execution for on the fly fine grained execution comparison. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 325–338, New York, NY, USA, 2015. ACM.

[50] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1128–1139, New York, NY, USA, 2014. ACM.

[51] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 29–44, Washington, DC, USA, 2010. IEEE Computer Society.

[52] Y. Kwon, X. Zhang, and D. Xu. Pietrace: Platform independent executable trace. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering*, pages 48–58, Nov 2013.

[53] Langdon and W. B. Mark Harman. Optimising Existing Software with Genetic Programming. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 19, NO. 1, FEBRUARY 2015.*

[54] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[55] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005.

[56] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement & code transplants to specialise a c++ program to a problem class. In *In 17th European Conference on Genetic Programming (EuroGP*, 2014.

[57] J. Polley, D. Blazakis, J. Mcgee, D. Rusk, and J. S. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE SECON '04*, 2004.

[58] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05*, 2005.

[59] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 353–368, Berkeley, CA, USA, 2013. USENIX Association.

[60] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. *SIGPLAN Not.*, 44(3):37–48, Mar. 2009.

[61] E. H. Spafford. Extending mutation testing to find environmental bugs. *Software Practice and Principle*, 20(2):181–189, February 1990.

[62] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[63] J. Swan, M. G. Epitropakis, and J. R. Woodward. Geno-fix: An embeddable framework for dynamic adaptive genetic improvement programming. 2014.

[64] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[65] M. Van Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 27–36, Washington, DC, USA, 2004. IEEE Computer Society.

[66] A. Vasudevan, N. Qu, and A. Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *Proceedings of the 2011 44th Hawaii International Conference on System Sciences*, HICSS '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[67] B. Walters. Vmware virtual platform. *Linux J.*, 1999(63es), July 1999.

[68] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 627–642, Berkeley, CA, USA, 2015. USENIX Association.

[69] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[70] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *Trans. Evol. Comp*, 15(4):515–538, Aug. 2011.

[71] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE*, 2007.

[72] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, May 2003.

[73] J. Zeng, Y. Fu, K. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *CCS '13*, 2013.