I/O Interaction Analysis of Binary Code

Konstantin Scherer
Technische Universität Berlin
Berlin, Germany
k.scherer@campus.tu-berlin.de

Tobias Pfeffer
Technische Universität Berlin
Berlin, Germany
tobias.pfeffer@tu-berlin.de

Sabine Glesner
Technische Universität Berlin
Berlin, Germany
sabine.glesner@tu-berlin.de

***Abstract*—The increasing use of closed-source software in everyday life leads to a need for privacy verification of binary code. Unfortunately, analyses of machine code behavior are often imprecise, do not scale, or require expert interaction. We present a novel and fast algorithm that is capable of soundly approximating I/O behavior of binary code. The key idea of our algorithm is that overall I/O behavior can be approximated by analyzing input and output operations. Since most function parameters are defined close to their use, our backward symbolic execution algorithm can quickly recover most meaningful parameters. We show the applicability and performance of our approach by analyzing the coreutils binaries.**

***Index Terms*—binary analysis, confidentiality, symbolic execution**

I. Introduction

Computing has gained an increasing presence in our everyday life, leading to software gaining access to much of our personal data. Software often shares the collected data in order to personalize provided services. While the sharing of information can improve provided services, information flow control is necessary to ensure that sensitive data is not leaked to unauthorized parties. Hence, it is desirable to validate software conformance with confidentiality constraints. Since the source code is not always available, the validation needs to work on binary code. In addition, the sheer amount of software involved in our everyday life makes it infeasible to rely on expert input during analysis. This drives the need for fully automatic confidentiality analyses on binary code.

Unfortunately, static analysis of binary code is known to be hard [1]. Due to the heavy use of pointers, the flat memory model, and the dynamic control flow, most source-code analyses are not applicable to binary code. Consequently, while practical information flow analyses exist for high-level languages [2], similar approaches on binary code typically do not scale [3]. The problem is that binary-code analyses are usually program-wide and semantics-based. Program execution traces have been used as an easy alternative to analyze I/O interactions [4]. Recorded system interactions are ordered by time of occurrence and contain concrete parameters. Typically, only a limited number of different runtime behaviors is taken into account. This is insufficient since even small differences in distinct executions can result in the transfer of information. To achieve sound results, it would be necessary to enumerate all possible program inputs. Instead, symbolic execution can be used to enumerate all paths [5]. The constraints produced by such a simulation can be used to recover the parameters of I/O interactions. However, symbolic execution often suffers from path explosion and the complexity of constraint solving operations.

In this paper, we describe a binary analysis technique to aid the automated validation of software confidentiality conformance. The key idea is that software needs to access system resources in order to gain access or distribute information. Usually, access needs to be granted explicitly by an underlying operating system. Thus, the overall software behavior can be approximated by analyzing only a subset of interactions with the OS (i.e. system calls). However, the behavior of I/O interactions depends on the type of interaction that is performed and corresponding runtime parameters. Hence, it is necessary to recover relevant parts of the runtime program state for any analysis of system interactions to be meaningful.

We propose an algorithm for recovering partial program states at any targeted instruction in binary code. The key idea of our algorithm is that many parameters are defined in close proximity to the instruction they are used in. Our algorithm is based on symbolic execution but simulates instructions backwards instead of forwards. We simulate instructions starting at targeted I/O interactions until sufficient program state information has been recovered. Because we recover the program state progressively, we can terminate the algorithm after a bounded number of instructions is simulated. We handle unresolved symbolic memory accesses by introducing a queue of pending operations.

In the next section, we outline related work. Then, we give a brief overview of the intermediate representation (IR) used for our algorithm definition. In Section IV, we present the proposed backward symbolic execution algorithm. We extend the algorithm description by presenting our symbolic memory model. Section V evaluates our algorithm using our implementation. Finally, we present our conclusion.

II. Related Work

Detection of malicious programs using system call traces has long been the subject of research (e.g. [4], [6]). However, most approaches use either the software source code or execution traces to locate and analyze system calls. The static analysis of Java Byte Code for Android devices described in [7] is similar to our approach in that analysis is focused on well-known functions that can perform potentially malicious system interactions. Recently, binary analysis techniques have become increasingly popular giving rise to multiple binary analysis platforms. Hex Rays' IDA Pro [8], for example, is a well-known proprietary interactive disassembler. BAP [9] is another binary analysis platform, which is open and features its own intermediate representation called BAP IL. BAP has been used to verify information flow properties for small code segments [3] using symbolic execution. However, this kind of verification does not scale well and is not feasible for the verification of an entire program. angr [10], which we have used to implement the proposed algorithm ourselves, is another binary analysis platform. It uses the VEX intermediate representation which was designed for Valgrind [11], a well-known framework for dynamic program analysis. Yet another binary analysis platform is BitBlaze [12]. To analyze binary code, a control flow graph is often needed. Fast and accurate control flow recovery is also an active research field [10], [13], [14]. We rely on the correctness of the transformation from assembly code to the intermediate representations. As shown in [15], the correctness of IRs is difficult to guarantee and even often used IRs are prone to contain bugs. Thus, basing algorithms on established and well-tested IRs is a very reasonable choice.

III. VEX

We use the VEX intermediate representation [11] for our definition and implementation of the backward symbolic execution algorithm. The VEX intermediate representation is structured into instruction superblocks (IRSB). Each IRSB has exactly one entry point but can have multiple points of exit. An IRSB contains a list of statements which may modify the program state and ends with an exit statement describing the succeeding IRSB location. VEX heavily uses temporary variables to hold values that are only used inside one IRSB. Temporary variables are not written more than once. The most common statements are `WrTmp`, `Put`, and `St` to write temporary variables, registers, and memory. Statements use expressions for calculations. The most common expressions are `RdTmp`, `Get`, and `Ld` to read temporary variables, registers, and memory. Unary and binary operations are also often used in calculations. Figure 1 displays a constructed example of an I/O interaction in C and VEX. The C code was compiled with clang and the VEX code was generated using PyVEX, which is a part of angr [10]. Because there are no branches in the code, its translation only takes up one IRSB. We use the `volatile` keyword to demonstrate memory load and store operations on the program stack.

C
```c
int open_for_read(const char* fname) {
    volatile const int flags = O_RDONLY;
    return open(fname, flags);
}
```

VEX
```
——— IMark(0x401130, 8, 0) ———
t6 = GET:I64(rsp)
t5 = Add64(t6,0xfffffffffffffffc)
STle(t5) = 0x00000000
PUT(rip) = 0x0000000000401138
——— IMark(0x401138, 4, 0) ———
t7 = Add64(t6,0xfffffffffffffffc)
t10 = LDle:I32(t7)
t9 = 32Uto64(t10)
PUT(rsi) = t9
——— IMark(0x40113c, 2, 0) ———
PUT(cc_op) = 0x0000000000000013
PUT(cc_dep1) = 0x0000000000000000
PUT(cc_dep2) = 0x0000000000000000
PUT(rax) = 0x0000000000000000
——— IMark(0x40113e, 5, 0) ———
NEXT: PUT(rip) = 0x0000000000401030
```

Fig. 1. Constructed I/O interaction in C and VEX

IV. I/O Interaction Analysis

Our goal is to provide a fast algorithm to approximate the I/O behavior of binaries. The results can be used to indicate attractive operations for further analyses. We focus on the analysis of I/O interactions and the recovery of their parameters. Since the parameters are most often defined in close proximity to the function call, we can avoid program-wide analysis. Additionally, since we perform backwards analysis, our algorithm can be terminated at any point and still produces sound results. This makes our approach especially well-suited as an early stage in a broader binary analysis toolchain. In this section, we describe our proposed algorithm to recover partial program states. We start by giving a quick overview and continue by describing the syntax and semantics we use to define our backward symbolic execution algorithm. After the basic algorithm has been defined, we present our symbolic memory model. I/O interaction parameters recovered by our algorithm can be used to approximate the overall program environment interactions. We work under the assumption that system calls are the only way for a program to interact with the environment. However, most programs issue system calls only indirectly by using external library functions. Because of this, we also target functions from the C standard library that are wrappers for system calls. System call addresses and most function call addresses are defined by the syntax and can therefore be recovered without extensive data-flow analysis. Hence, we focus on recovering dynamic function parameters, to provide additional details about the program-specific use of the I/O interface.

As an example, we consider the `open` system call (or C-function, respectively). While it may seem like `open` is in itself not an output instruction, it can become one if the file creation flag is provided. If this is the case, `open` will create a file with the provided pathname. Thus, `open` can be abused to communicate with a different process via the file system. Conversely, `open` can be used as an input instruction, e.g. to check for the presence of files. Knowledge about the parameters can also guide subsequent analyses that track the use of the returned file descriptor. If all calls to `open` with the file-reading flag target unclassified channels, it can be inferred

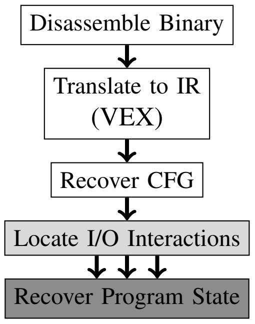

Fig. 2. General process

that the program is confidential. Similarly, if all calls with file-writing flag target authorized channels, it can also be inferred that the program is confidential. Therefore, fast analysis of the specific use of the I/O interface enables early detection and focused post-processing. In the following, we give a general overview of our technique and provide our analysis algorithm in detail.

A. Overview

Binary code is inherently unstructured. Hence, we rely on a recovered control flow graph to locate I/O interactions and determine the preceding instructions. Control flow recovery of binary code is a complicated operation and we use already proposed CFG recovery algorithms (i.e. [10], [14]). We locate target instructions by scanning the CFG for I/O interactions. We then use backward symbolic execution for every found target instruction to partially recover the corresponding run-time program states. Figure 2 illustrates this behavior. Instead of using raw assembly code, we use the VEX intermediate representation. Backward symbolic execution simulates VEX statements in reverse order than in normal program execution. Each statement is simulated using a symbolic program state. The program state contains a set of constraints describing all of the recovered register and memory values. Each statement simulation adds constraints, refining the overall program state. Because constraints do not get removed once they are added, the information contained in the simulated program state increases continuously. The first statement to be simulated is the statement right before the targeted I/O interaction. After that, the simulation progresses by simulating the preceding statement next. If there are multiple predecessors, each of the preceding statements gets simulated separately using a copy of the simulation state.

B. Backwards Symbolic Execution

In this section, we formally describe the backward symbolic execution algorithm using inference rules. Our syntax is based on the formal definition of forward symbolic execution in [5]. We store the current program state in a tuple of the form $\langle R, T, C, pc \rangle$. The state consists of the register mapping R, the temporary variable mapping T, the set of constraints C, and the current statement index pc. The register and temporary variable mapping map the names of variables and registers

$$\frac{\text{computation}}{\langle\text{state}\rangle, \text{expr} \vdash \text{expr'}}\ \text{Expr} \qquad \frac{\text{computation}}{\langle\text{state}\rangle, \text{stmt} \rightsquigarrow \langle\text{state'}\rangle}\ \text{Stmt}$$

$$\frac{\text{computation}}{\langle\text{substate}\rangle, \text{stmt}, \text{guard} \rightsquigarrow \langle\text{substate}'\rangle}\ \text{Pending Op.}$$

Fig. 3. Inference rule syntax

to their symbolic value. The set of constraints is a set of conditions about variable values that are all satisfied in the program state (e.g. $\{\texttt{rax} == 3\}$). We call a variable concrete if it can be evaluated to exactly one value using the state constraints. At any point, all symbolic variables must correspond to at least one concrete value, otherwise, the program state is unsatisfiable and we discard the corresponding simulation. pc denotes the index of the statement in the IRSB that is currently being simulated and is not related to the instruction pointer. We give the first statement of an IRSB a statement index of 1 and index succeeding statements incrementally. As such, the simulation of one block starts with the statement having the highest statement index and finishes once pc reaches 0. Additionally, we use Σ to denote a constant mapping from the statement indices to the statements. The variable $i \in \mathbb{N}$, which is not stored in the program state for convenience, is used to index variables. x_i^n denotes a variable with a size of n bits. i is always incremented after being used, leading to x_i^n always denoting an unused unconstrained variable. Mappings are accessed using square brackets (e.g. $R[\texttt{rip}]$). We use $M[k/v]$ to denote a new mapping, which is a copy of M but with $M[k]$ replaced by v. If k is a variable, we evaluate it to a constant value using the state constraints. We write $e \Downarrow e'$ to evaluate the expression e to e'. The evaluation implicitly uses the current program state and constraints. Once an IRSB has been entirely simulated, simulation can continue with the preceding instruction block(s). For each preceding instruction block, the simulation state is copied and the temporary variable mapping, as well as the statement index, are initialized anew. Additionally, Σ is updated in each simulation. Statements and expressions are defined in the form shown in Figure 3. Expressions are evaluated (denoted $\vdash$) recursively until no further expression evaluation rule can be applied. Only statements can change the program state (denoted $\rightsquigarrow$). A problem arises when memory operations make use of symbolic addresses. To solve this, we introduce pending operations in the next section.

Figure 4 shows the inference rules for statements and expressions. The displayed statements and expressions are the most commonly occurring operations in VEX code and support for most other operations can be added in a straight forward way[1]. The inference rules are identical for each variable size. We use $\diamond_u$ and $\diamond_b$ as a placeholders in `UnOp` and `BinOp` for the corresponding operations (e.g. $+, -$). In

[1]The presented combination of statements and expressions covers approx. 97% of all instructions in the GNU core utility binaries.

$$\frac{}{\langle R,T,C,pc\rangle, v \vdash v}\ \text{Const} \qquad \frac{}{\langle R,T,C,pc\rangle, \text{Get}(r) \vdash R[r]}\ \text{Get} \qquad \frac{}{\langle R,T,C,pc\rangle, t \vdash T[t]}\ \text{RdTmp}$$

$$\frac{v \Downarrow v'}{\langle R,T,C,pc\rangle, \text{UnOp}(v) \vdash \diamond_u v'}\ \text{UnOp} \qquad \frac{e \Downarrow e', \quad C' = C \bigcup \{R[r] == e'\}, \quad R' = R[r/x_i], \quad i \leftarrow i+1}{\langle R,T,C,pc\rangle, \text{Put}(r) = e \rightsquigarrow \langle R',T,C',pc-1\rangle}\ \text{Put}$$

$$\frac{a \Downarrow a', \quad b \Downarrow b'}{\langle R,T,C,pc\rangle, \text{BinOp}(a,b) \vdash a' \diamond_b b'}\ \text{BinOp} \qquad \frac{e \Downarrow e', \quad C' = C \bigcup \{T[t] == e'\}}{\langle R,T,C,pc\rangle, t = e \rightsquigarrow \langle R,T,C',pc-1\rangle}\ \text{WrTmp}$$

Fig. 4. Expression and statement semantics

the `WrTmp` statement, we simply add a constraint for the temporary variable that is written. Temporary variables in VEX are not written more than once, hence, we do not have to care about retrieving the wrong version of the variable. The `Put` statement writes registers values. Because registers can be overwritten, we need to ensure that register read operations retrieve the value of the correct register version. We achieve this by adding a constraint to the program state that directly influences previously simulated register read operations once a value has been written to a register. Since register values are unknown before they are written, we replace the symbolic register variable in the register mapping with a new unconstrained variable after register write operations. We demonstrate the evaluation of a `WrTmp` statement in Figure 5.

C. Symbolic Memory

Having introduced the basic algorithm, we present our method of handling symbolic memory. In contrast to register operations, memory operations often do not have concrete targets. Additionally, memory can be overwritten. Therefore, we need to make sure that loaded values correspond to the correct stored values. Since we perform our analysis backwards, target addresses of memory operations might be unknown when first encountered by the simulation. To handle this, we present a very simple memory model that is sound and easy to implement. We introduce a queue of pending memory operations that are executed in order, once target memory addresses are known. Executing memory operations in order allows us to use the same memory versioning we use for registers. Because our backward symbolic execution algorithm is intended to simulate a bounded number of instructions, we decided in favor of simplicity and result soundness. However, especially the simulation of a large number of statements and operations with memory addresses that are themselves loaded from memory can lead to incomplete results. We extend the program state with a memory mapping M and a queue for pending memory operations P. The updated program state tuple is $\langle R,T,C,pc,M,P\rangle$. We write $P + [\texttt{op}]$ to append an operation to the queue. We use $M_e^n[a]$ to access n contiguous bits from the memory mapping, starting at address a and using the endianness e. We introduce guard conditions to indicate if pending operations can be executed. Pending operations are only executed when their guard evaluates to true. For memory load and store operations, the guard indicates if the target address is concrete. We write $|a|_C == 1$ to check if a is concrete using the set of constraints C. Relying on a guard allows us to use the same pending operations queue for more complex conditions, if the need arises (e.g. evaluating system functions once all parameters are concrete). The inference rules for pending operations are defined similar to statements, except that they feature a guard and operate on a substate, which is is subset of the complete program state (Figure 3). Because target memory addresses can be symbolic, all memory operations make use of the pending operations queue and do not execute their operation immediately. Instead, they enqueue a pending operation which performs the actual operation at a later time. Our inference rules for memory operations are shown in Figure 6. The load expression always evaluates to a new unconstrained variable. Once the corresponding pending load operation is executed, a constraint is added to simulate the actual memory load. Similarly, the store statement does not store the value immediately. The value gets written to memory only when the corresponding pending store operation is performed. The pending store works much like the `Put` statement. A constraint for the stored value is added and the memory mapping gets updated with a new unconstrained value of given size. Note, that even though pending operations can modify the program state, the pending load does not update any variable mapping. We demonstrate the evaluation of a `Store` statement in Figure 5. The simplest way to execute the pending memory operations is to simulate and remove the first pending operation while the guard evaluates to true (Figure 7). In our implementation, we retry the pending operations after a new operation was appended and when the IRSB was completely simulated. In order to improve result completeness, we implemented additional mechanics for pending memory operations to be evaluated. Since write-protected memory regions cannot be overwritten by store operations, we allow out of order execution of memory loads that target write-protected memory. Additionally, we try all memory load operations in the queue until the first store operation is encountered, even if some load operations do not have concrete target addresses. This is possible because load operations are order independent amongst themselves. To simulate the program stack, we start the initial simulation with a concrete memory address in the stack pointer register. This ensures, that operations using the stack pointer have concrete addresses and can be executed. By starting the simulation with a concrete stack pointer, the

$$\dfrac{\dfrac{}{\langle R,T,C,pc\rangle\,,\text{Get:I64}(\texttt{rsp}) \vdash R[\texttt{rsp}]}\,\text{Get}, \quad C' = C \bigcup \{T[t_6] == T[\texttt{rsp}]\}}{\langle R,T,C,pc\rangle\,, t_6 = \text{Get:I64}(\texttt{rsp}) \rightsquigarrow \langle R,T,C',pc-1\rangle}\,\text{WrTmp}$$

$$\dfrac{\dfrac{}{\langle R,T,C,pc\rangle\,, t_5 \vdash T[t_5]}\,\text{RdTmp}, \quad \dfrac{}{\langle R,T,C,pc\rangle\,, \text{0x0} \vdash \text{0x0}}\,\text{Const}, \quad P' = P + [\text{PStore}^{64}_{\text{le}}(M^{64}_{\text{le}}[T[t_5] = \text{0x0}])]}{\langle R,T,C,pc,M,P\rangle\,, \text{Store}^{64}_{\text{le}}(t_5) = \text{0x0} \rightsquigarrow \langle R,T,C,pc,M,P'\rangle}\,\text{Store}$$

$$\dfrac{C' = C \bigcup \left\{M^{64}_{\text{le}}[T[t_5]] == \text{0x0}\right\}, \quad M' = M^{64}_{\text{le}}[T[t_5]/x^{64}_i], \quad i \rightarrow i+1}{\langle R,T,C,M\rangle\,, \text{PStore}^{64}_{\text{le}}(M^{64}_{\text{le}}[T[t_5]] = \text{0x0}), |T[t_5]|_C == 1 \rightsquigarrow \langle R,T,C',M'\rangle}\,\text{PStore}$$

Fig. 5. Evaluation of line 2 and 4 of the VEX code in Figure 1

$$\dfrac{v = x^n_i, \quad i \leftarrow i+1, \quad a \Downarrow a', \quad P' = P + [\text{PLoad}^n_e(v == M^n_e[a'])]}{\langle R,T,C,pc,M,P\rangle\,, \text{Load}^n_e(a) \vdash v, P'}\,\text{Load}$$

$$\dfrac{v \Downarrow v', \quad a \Downarrow a', \quad P' = P + [\text{PStore}^n_e(M^n_e[a'] = v')]}{\langle R,T,C,pc,M,P\rangle\,, \text{Store}^n_e(a) = v \rightsquigarrow \langle R,T,C,pc,M,P'\rangle}\,\text{Store}$$

$$\dfrac{C' = C \bigcup \{v == M^n_e[a]\}}{\langle R,T,C,M\rangle\,, \text{PLoad}^n_e(v == M^n_e[a]), |a|_C == 1 \rightsquigarrow \langle R,T,C',M\rangle}\,\text{PLoad}$$

$$\dfrac{C' = C \bigcup \{M^n_e[a] == v\}, \quad M' = M^n_e[a/x^n_i], \quad i \leftarrow i+1}{\langle R,T,C,M\rangle\,, \text{PStore}^n_e(M^n_e[a] = v), |a|_C == 1 \rightsquigarrow \langle R,T,C',M'\rangle}\,\text{PStore}$$

Fig. 6. Memory operation semantics

```
procedure TryPendingOperations
    while $|P| > 0$ do
        $\sigma$, guard $\leftarrow P[0]$
        if $\neg$Evaluate(guard) then break end if
        Simulate($\sigma$)
        $P \leftarrow P \backslash P[0]$
    end while
end procedure
```

Fig. 7. Pseudocode for trying the pending memory operations

simulation is able to completely recover the *flag* parameter in the constructed example shown in Figure 1. While our memory model features simplicity and result soundness, results can be incomplete. The pending operations queue can, for example, be stalled by store operations that have a symbolic target address. Memory addresses that get loaded from memory themselves can be difficult for the simulation to handle. However, we expect that binary code, especially for architectures that use registers for most function parameters (e.g. x86-64, ARM), can be sufficiently simulated for a bounded number of instructions using our proposed algorithm.

V. Evaluation

To demonstrate the feasibility, efficiency, and accuracy of our algorithm, we provide a reference implementation based on the angr [10] binary analysis framework. We have evaluated the algorithm using all binaries from the GNU core utilities[2] that contained relevant external function calls. We chose the `open`, `read` and `write` functions from the C standard library for our more detailed examination. The `open` function is similar to the identically named system call. Figure 8 shows the average symbolic bits of the *flags* and *pathname* parameters to the `open` function (35 binaries were analyzed). We forcefully terminated the simulation after the simulation of 7 instruction blocks. However, some simulations encountered external library calls that could not be simulated and were terminated early. We ran the analysis on a PC equipped with an Intel(R) Xeon(R) E3–1231 v3 (4 cores up to 3.80 GHz) processor and 8GB of RAM in a single thread. Contrary to the *flag* parameter bitfield, we counted *pathname* as either fully symbolic or concrete. It can be seen that after the simulation of only four to five IRSBs, almost no new information could be recovered. In accordance with our initial hypothesis, some parameters could be recovered with great accuracy by only

[2]GNU core utilities version 8.30 compiled with gcc 8.2.0 for x86-64 Ubuntu 18.10.

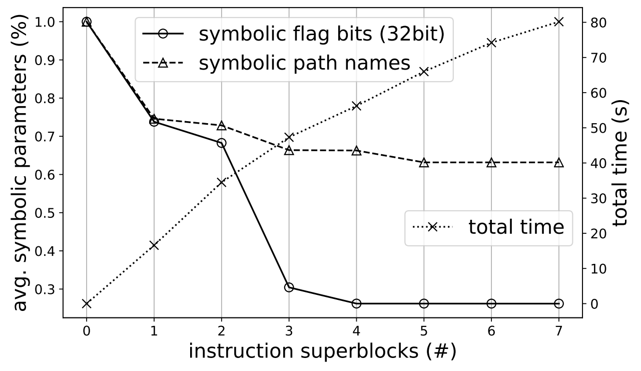

Fig. 8. Symbolic parameters to `open`

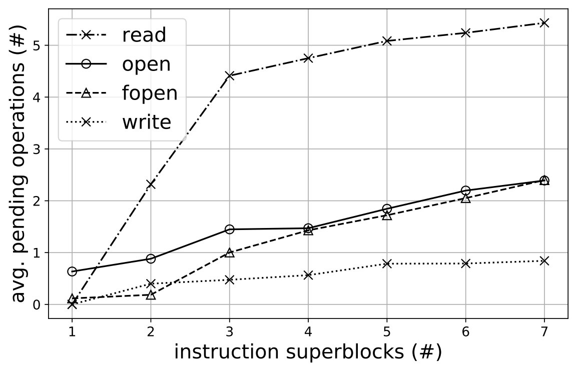

Fig. 9. Number of pending operations

simulating very few instructions preceding the targeted function call. Our algorithm successfully recovered partially symbolic parameters. It is interesting to note that many parameters were not entirely concrete but instead had few symbolic bits left. This happened because some flag bits depend on function parameters that could not be recovered. Figure 9 shows the average number of pending operations for the analysis of various functions. Except for the `read` function, the number of pending operations only slowly increased after each IRSB. This indicates that our memory model is sufficient for the simulation of most functions but struggles in some scenarios.

VI. Conclusion

In this paper, we have presented our novel algorithm for efficient recovery of I/O operation parameters in binary code. Our algorithm uses backward symbolic execution and is capable of recovering partial program states at any instruction. Additionally, we have defined the algorithm with and without support for symbolic memory operations. The recovered partial program states are sound and are shown to facilitate recovery of function parameters to I/O interactions. In the evaluation section, we have shown that many function parameters to I/O interactions are, in fact, defined close to their use. Because our presented algorithm is sound, its results can be reused for further analysis to validate software privacy guidelines. We expect that future work is able to improve our techniques and provide more complete results. Symbolic memory could, for example, be improved by implementing various different memory resolution techniques (e.g. [16]). Another possibility is the use of heuristics to greatly increase execution speed and completeness at the cost of result soundness. We expect that our algorithm and the results gathered can be used to aid binary privacy validation algorithms.

References

[1] X. Meng and B. Miller, "Binary code is not easy," Tech. rep., Computer Sciences Department, University of Wisconsin, Madison, Tech. Rep., 2015.

[2] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, "Checking probabilistic noninterference using joana," *it-Information Technology*, vol. 56, no. 6, pp. 280–287, 2014.

[3] M. Balliu, M. Dam, and R. Guanciale, "Automating information flow analysis of low level code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1080–1091.

[4] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, 2001, pp. 156–168.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[6] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 45–60.

[7] C. Mann and A. Starostin, "A Framework for Static Detection of Privacy Leaks in Android Applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 1457–1462.

[8] C. Eagle, *The IDA pro book*. No Starch Press, 2011.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[11] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[12] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Information Systems Security*, R. Sekar and A. K. Pujari, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25.

[13] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universität Darmstadt, 2010.

[14] T. Pfeffer, P. Herber, L. Druschke, and S. Glesner, "Efficient and Safe Control Flow Recovery Using a Restricted Intermediate Language," in *VSC Track at the IEEE International Conference on Enabling Technologies (WETICE 2018)*. IEEE Computer Society, 2018.

[15] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing Intermediate Representations for Binary Analysis," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 353–364.

[16] A. Romano and D. R. Engler, "symMMU: Symbolically Executed Runtime Libraries for Symbolic Memory Access," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 247–258.