Precise Static Analysis of Untrusted Driver Binaries

Johannes Kinder
Technische Universität Darmstadt
Darmstadt, Germany
Email: kinder@cs.tu-darmstadt.de

Helmut Veith
Technische Universität Wien
Vienna, Austria
Email: veith@forsyte.at

***Abstract*—Most closed source drivers installed on desktop systems today have never been exposed to formal analysis. Without vendor support, the only way to make these often hastily written, yet critical programs accessible to static analysis is to directly work at the binary level. In this paper, we describe a full architecture to perform static analysis on binaries that does not rely on unsound external components such as disassemblers. To precisely calculate data and function pointers without any type information, we introduce Bounded Address Tracking, an abstract domain that is tailored towards machine code and is path sensitive up to a tunable bound assuring termination.**

We implemented Bounded Address Tracking in our binary analysis platform Jakstab and used it to verify API specifications on several Windows device drivers. Even without assumptions about executable layout and procedures as made by state of the art approaches [1], we achieve more precise results on a set of drivers from the Windows DDK. Since our technique does not require us to compile drivers ourselves, we also present results from analyzing over 300 closed source drivers.

I. Introduction

Software model checking and static analysis are successful methods for finding certain bugs or proving their absence in critical systems software such as drivers. Source code analysis tools like SDV [2] are available for developers to statically check their software for conformance to specifications of the Windows driver API. For instance, if a driver calls the API method IoAcquireCancelSpinLock, it is required to call IoReleaseCancelSpinLock before calling IoAcquireCancelSpinLock again [3]. The vendors, however, are not forced to use these analysis tools in development, and they are unwilling to submit their source code and intellectual property to an external analysis process. Without source code, certification programs such as the Windows Hardware Quality Labs (WHQL) have to rely on testing only, which cannot provide guarantees about all possible executions of a driver. A solution to this problem is to relocate the static analysis to the level of the compiled binary. If the analysis does not require source code or debug symbols, an analysis infrastructure can be created independently of active vendor support.

Working with binaries poses several specific challenges. In general, code cannot be easily identified in x86 executables such as Windows device drivers. Data can be arbitrarily interleaved with code, and bytes representing code can be interpreted as multiple different instruction streams depending on the alignment at which decoding starts [4]. Therefore, a major challenge in analyzing binaries is to reliably extract those instructions that are actually executed at runtime and to build a control flow graph that accurately represents the possible targets even of indirect jumps. Existing approaches to static analysis of binary executables rely on a preprocessing step performed by a dedicated, heuristics based disassembler such as IDA Pro [5] to produce a plain text assembly listing [6]. This decouples the analysis infrastructure from disassembly itself and makes it difficult to use results from static analysis towards improving the control flow graph. Furthermore, since the analysis builds on an external disassembler, soundness can only be guaranteed with respect to the (error prone) output produced by the disassembler.

To overcome this problem, we propose an architecture for single pass disassembly and analysis, which does not discriminate between disassembly and analysis stages (Figure 1). Its integrative design is based on the following key insight: Following the control flow of a binary in order to decode the executed instructions is already an analysis of reachable locations. This is non-trivial in presence of indirect control-flow and should not be left to heuristic algorithms.

Another challenge in statically analyzing binaries is that the lack of types and the a priori unknown control flow make a cheap points-to analysis impossible. Every dereference of an unknown pointer can mean an access to any memory address, be it the stack, global memory, or the heap. A write access then causes a *weak update* to the entire memory: After the write, every memory location *may* contain the written value, which dramatically impacts the precision of the analysis. Worst of all, weak updates potentially overwrite return addresses stored on the stack (or function pointers anywhere in memory), which can cause spurious control flow to locations that are never executed at runtime. The goal of a sound and precise analysis on binaries is thus to achieve *strong updates* wherever possible: If a pointer can only point to one specific address in a state, the targeted memory location *must* contain the written value after a write access [7].

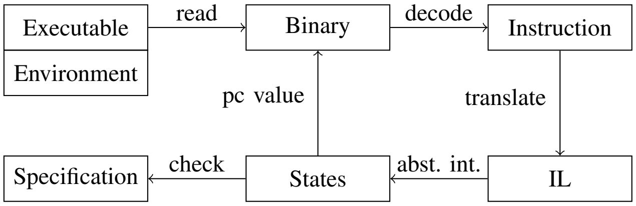


Fig. 1. Disassembly and analysis architecture.

In essence, an analysis capable of dealing with the lack of types in binaries needs to be precise enough to represent addresses without over-approximation that might introduce spurious control flow into non-code regions. On the other hand, high precision analyses are known not to scale to larger programs, so abstraction has to be introduced where possible. In this paper, we present our approach to dealing with these challenges without sacrificing soundness. In particular, our paper makes the following contributions:

- We describe an architecture for checking specifications on binary executables without access to source code and without a heuristics based, separate disassembly step. The control flow of the binary is reconstructed in a single pass with static analysis, following the approach presented in [8]. Abstractions of the execution environment can be written in C and are compiled into a separate module.
- We introduce *Bounded Address Tracking*, an abstract domain based on tracking a selection of register and memory values up to a given bound (inspired by [9]). The path sensitivity of our analysis allows strong updates to allocated heap regions. Since path sensitivity subsumes context sensitivity, we do not require assumptions about a separate call stack or well-structured procedures.
- In our path-sensitive analysis, nondeterminism in the program (e.g., from modeling input) is especially expensive. To address this issue, we offer two different constructs for nondeterminism, *havoc* and *nondet*, which cause explicit enumeration of variable values or their abstraction to an unknown value, respectively.

II. Background

We extended our own iterative disassembler JAKSTAB [10] to implement the integrated analysis architecture for single pass disassembly and static analysis (Figure 1). Using the entry point of the executable as the initial program counter (*pc*) value, our tool decodes one instruction at a time from the file offset that corresponds to *pc*. This instruction is then translated into one or more statements of the intermediate language (IL). Depending on the abstract domain chosen for the analysis, JAKSTAB calculates successor states by interpreting the abstract semantics of the IL. If a newly reached state is an error state according to the specification, an abstract error trace is generated. Otherwise, JAKSTAB concretizes new *pc* values from the states and uses these to decode the next instructions to be interpreted.

A. Low Level Intermediate Language

CISC architectures such as x86 offer very rich instruction sets, in which a single instruction can affect multiple registers and status flags and can even represent non-trivial operation sequences including loops. To avoid dealing with hundreds of different concrete and abstract state transformers when analyzing machine code, we translate each instruction into a sequence of IL statements using specifications of the instruction semantics. For instance, the instruction push eax, which pushes the contents of register eax to the stack and decrements the stack pointer, is specified to translate to the IL code $m[esp] := eax; esp := esp-4$. Note that for simplicity of the exposition, in this paper we assume all memory accesses and all bit vectors to be 32 bit. The actual implementation allows arbitrary word lengths using bit masking expressions.

The IL uses a finite set of bit vector type registers $V = \{v_0, \ldots, v_n\}$, a store $m[\cdot]$, and the program counter pc. The set $\mathbf{Exp}$ of expressions of the IL contains common arithmetic, Boolean, and bit-manipulation operations. All expressions are of the 32-bit bit vector type $\mathbb{I}_{32}$; Boolean *true* and *false* are represented by the bit vectors 1 and 0, respectively. To model input from the hardware, expressions can contain the keyword $nondet$, which nondeterministically evaluates to some bit vector value in its concrete semantics.

A program is made up of IL statements of the form $[stmt]^{\ell}_{\ell'}$, where $\ell \in \mathbb{I}_{32}$ is the address of the statement, $\ell' \in \mathbb{I}_{32}$ is the address of the next statement, and $stmt \in \mathbf{Stmt}$ is one of nine types of statements:

- Register assignments $v := e$, with $v \in V$ and $e \in \mathbf{Exp}$, assign the value of expression e to register v.
- Store assignments $m[e_1] := e_2$, with $e_1, e_2 \in \mathbf{Exp}$, assign the value of expression e_2 to the memory location at the address computed by evaluating e_1.
- Guarded jumps if e_1 jmp e_2, with $e_1, e_2 \in \mathbf{Exp}$, transfer control to the target address resulting from evaluating e_2 if the guard expression e_1 does not evaluate to 0.
- A halt statement terminates execution.
- Allocation statements alloc v, e, with $v \in V$ and $e \in \mathbf{Exp}$, reserve a block of memory of the size determined by evaluating e and write the address to register v.
- Deallocation statements free v release the block of memory pointed to by $v \in V$ for reallocation.
- Statements assume e terminate execution if $e \in \mathbf{Exp}$ evaluates to 0, and do nothing otherwise.
- Assertions assert e are similar to assume statements, but signal an error on termination.
- Statements havoc $v <_u n$, with $v \in V, n \in \mathbb{I}_{32}$, nondeterministically assign a value x with $0 \leq_u x \leq_u n$ to v, where $\leq_u$ denotes unsigned comparison. The same effect can be achieved using $v := nondet$; assume $v \leq_u n$. The point of having two different sources of nondeterminism becomes apparent in Section III-C, where they will be used for selective abstraction.

The statements alloc, free, assert, and havoc are never generated from regular instructions, but are encoded in our abstracted model of the operating system (Section IV-C).

Note that call and return instructions receive no special treatment in our IL but are translated to assignments and jumps. In x86 assembly, these instructions simply store the current program counter on the stack and jump to a target, or read an address from the stack and jump to it, respectively. There is no fixed concept of procedures in x86 assembly, so relying on binary code to respect high level procedural structuring can introduce unsoundness into the analysis.

The concrete IL semantics is defined in terms of states $S = \mathbf{Loc} \times \mathbf{Val} \times \mathbf{Store} \times \mathbf{Heap}$, consisting of the location

$$
\begin{aligned}
\mathbf{post}\llbracket[v := e]^{\ell}_{\ell'}\rrbracket(s) &:= s[v \mapsto \mathbf{eval}\llbracket e\rrbracket(s)][pc \mapsto \ell']\\
\mathbf{post}\llbracket[m[e_1] := e_2]^{\ell}_{\ell'}\rrbracket(s) &:= s[m[\mathbf{eval}\llbracket e_1\rrbracket(s)] \mapsto \mathbf{eval}\llbracket e_2\rrbracket(s)][pc \mapsto \ell']\\
\mathbf{post}\llbracket[\textsf{if } e_1 \textsf{ jmp } e_2]^{\ell}_{\ell'}\rrbracket(s) &:= \begin{cases} s[pc \mapsto \ell'] & \text{if } \mathbf{eval}\llbracket e_1\rrbracket(s) = 0\\ s[pc \mapsto \mathbf{eval}\llbracket e_2\rrbracket(s)] & \text{otherwise}\end{cases}\\
\mathbf{post}\llbracket[\textsf{halt}]^{\ell}_{\ell'}\rrbracket(s) &:= \bot\\
\mathbf{post}\llbracket[\textsf{alloc } v, e]^{\ell}_{\ell'}\rrbracket(s) &:= s[v \mapsto h][pc \mapsto \ell'],\ \text{min. } h > h_0 \text{ s.t.}\\
\forall(h', z') \in s(H).h \geq h' + z' &\vee h + z \leq h', \text{where } z = \mathbf{eval}\llbracket e\rrbracket(s)\\
\mathbf{post}\llbracket[\textsf{free } v]^{\ell}_{\ell'}\rrbracket(s) &:= s[H \mapsto H \setminus (v, \cdot)][pc \mapsto \ell']\\
\mathbf{post}\llbracket[\textsf{assume } e]^{\ell}_{\ell'}\rrbracket(s) &:= \begin{cases} \bot & \text{if } \mathbf{eval}\llbracket e_1\rrbracket(s) = 0\\ s[pc \mapsto \ell'] & \text{otherwise}\end{cases}\\
\mathbf{post}\llbracket[\textsf{assert } e]^{\ell}_{\ell'}\rrbracket(s) &:= \begin{cases} \bot\text{(raise error)} & \text{if } \mathbf{eval}\llbracket e_1\rrbracket(s) = 0\\ s[pc \mapsto \ell'] & \text{otherwise}\end{cases}\\
\mathbf{post}\llbracket[\textsf{havoc } v <_u n]^{\ell}_{\ell'}\rrbracket(s) &:= s[v \mapsto x][pc \mapsto \ell'],\ \text{with some } x \leq n
\end{aligned}
$$

Fig. 2. Concrete semantics of the intermediate language.

valuation $\mathbf{Loc} := \{pc\} \rightarrow \mathbb{I}_{32}$, the register valuation $\mathbf{Val} := V \rightarrow \mathbb{I}_{32}$, the store valuation $\mathbf{Store} := \mathbb{I}_{32} \rightarrow \mathbb{I}_{32}$, and a heap set $\mathbf{Heap} := \mathbb{I}_{32} \rightarrow \mathbb{I}_{32}$, which maps addresses of allocated heap objects to their corresponding sizes. Allocation of heap objects starts above some constant h_0 in the address space. We denote access to parts of the state by $s(pc)$, $s(v_i)$, $s(m[\cdot])$, $s(H(p))$. The syntax $s[\cdot \mapsto \cdot]$ denotes the state obtained by updating part of state s with a new value. The concrete semantics is then given by the concrete $\mathbf{post}$ operator from states and statements to states in Figure 2. It uses the operator $\mathbf{eval} :: \mathbf{Exp} \rightarrow \mathbb{I}_{32}$ to concretely evaluate IL expressions.

B. Control Flow Reconstruction

In [8], we proposed an integrated theoretical framework for building the most precise control flow graph of a low level program while calculating data flow facts, akin to control flow analysis in functional programming languages. The basic idea of the framework is to translate low level statements into edges $(\mathbb{I}_{32} \times \mathbf{Stmt} \times \mathbb{I}_{32})$ of the control flow automaton (a control flow graph where edges instead of vertices carry the statements). The edges over-approximate the concrete control flow of the program, eliminating any indirect jumps.

In particular, every guarded jump $[\textsf{if } e_1 \textsf{ jmp } e_2]^{\ell}_{\ell'}$ is transformed into a set E of edges labeled with assume statements: If $e_1 = 0$, E contains the fall-through edge $(\ell, \textsf{assume } (e_1 = 0), \ell')$. If $e_1 \neq 0$, E also contains all of the possible target edges $\{(\ell, \textsf{assume } (e_1 \neq 0 \wedge e_2 = \ell''), \ell'') \mid \ell'' \in \widehat{\mathbf{eval}}\llbracket e_2\rrbracket(\widehat{\mathbf{post}}\llbracket \textsf{assume } (e_1 \neq 0)\rrbracket(s))\}$, where $\widehat{\mathbf{post}}$ and $\widehat{\mathbf{eval}}$ denote the abstract $\mathbf{post}$ and $\mathbf{eval}$ operator of a suitable abstract domain, respectively. The key feature that allows this approach to produce the most precise control flow automaton is that the conditions for taking a particular edge from a guarded jump, i.e., the jump condition and the jump target, are encoded into the assumption.

As a result, an abstract domain used with this framework only needs to supply implementations of the $\widehat{\mathbf{post}}$ (for statements other than jmp) and $\widehat{\mathbf{eval}}$ operators and does not need to deal specifically with indirect jumps.

III. Precise Pointer and Value Analysis

The translation of guarded jumps to labeled edges requires a precise evaluation of the target expression, otherwise spurious control flow edges can be introduced that point into code or data sections never meant to be executed, causing a cascading loss of precision. Furthermore, the lack of types in binaries prohibits a limited over-approximation of points-to sets. While in regular source based static analysis an unknown pointer may point to all variables of the matching type, an unknown pointer in untyped assembly code may point to any location in the entire memory, including code.

We have therefore devised a highly precise abstract domain for tracking states as valuations of registers and memory locations that supports pointer arithmetic and the ambiguity between integer values and addresses (there is no distinction between pointers and regular values in machine code).

A. Memory Model

The virtual memory available to a process is organized as one large, continuous array. The stack, the heap, and global variables all share this address space. The runtime environment initializes the stack and heap locations to reasonable values such that they do not interfere, and it uses buffer pages between these logical memory regions to detect overflows. Correct implementations of malloc (and its kernel-level equivalents available to drivers) guarantee that allocated memory blocks in the heap do not overlap. Therefore, we use a concrete memory model based on a set $\mathbf{R}$ of separate *memory regions*:

- The global region, containing code, global variables, and static data,
- a single stack, holding local variables, parameters, and return addresses at runtime,
- and zero or more allocated heap regions, which correspond to memory blocks allocated using malloc.

We thus treat every memory address as a pair of memory region and offset from $\mathbf{R} \times \mathbb{I}_{32}$. Pointers into the global region are denoted by $(\textsf{global}, \mathit{offset})$; the stack pointer is assumed to be initialized to a value of $(\textsf{stack}, 0)$. Subsequent modifications to the stack pointer then change the offset, but let it stay within the stack region. In x86, the stack grows downward, so the stack pointer will always have negative offsets within valid code. The number of heap regions is unbounded, and a fresh heap region is created by any call to malloc. A fresh identifier tags the individual heap region, creating pointers such as $(\textsf{alloc}_{id}, \mathit{offset})$.

Strictly speaking, this memory model presents an abstraction of the actual x86 memory layout, since it ignores the relative position of regions to each other. If for whatever reason the memory region model is too imprecise for the kind of code being analyzed, it can be effectively turned off by initializing the stack pointer and any newly allocated memory into the global address space.

Our memory model combines integer and pointer values similarly to Value Set Analysis [6]; it does not make the assumption of separated procedure stack frames, however, but uses a single region for the entire stack instead.

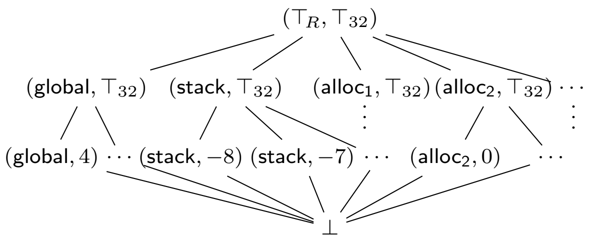
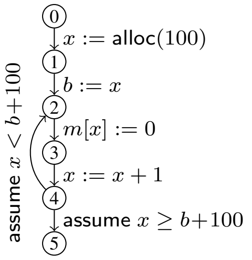


Fig. 3. Diagram of the lattice of abstract addresses $\hat{A}$.



0
$x := \mathsf{alloc}(100)$
1
$b := x$
2
$m[x] := 0$
3
$x := x + 1$
4
$\mathsf{assume}\ x \geq b+100$
5
$\mathsf{assume}\ x < b+100$

ℓ	# x	# b
0	1	1
1	1	1
2	6	1
3	6	1
4	6	1
5	1	1

x, # b: Number of unique values for x and b.

Fig. 4. Example code fragment and final value counts.

B. Bounded Address Tracking

To build our abstract domain, we extend the model of memory addresses to a lattice that includes a top element $(\top_R, \top_{32})$ representing a memory address with the unknown region $\top_R$ and unknown offset $\top_{32}$. We further introduce an intermediate level of pointers with known region but unknown offset of the form $(\mathit{region}, \top_{32})$, which represents the join of different addresses within the same region (e.g., $(r, 4) \sqcup (r, 8) = (r, \top_{32})$). We thus define the set of abstract memory addresses as $\hat{A} = \{(\top_R, \top_{32})\} \cup (\mathbf{R} \times \{\top_{32}\}) \cup (\mathbf{R} \times \mathbb{I}_{32})$. The resulting lattice for $\hat{A}$ is sketched in Figure 3.

Our analysis over-approximates the set of reachable concrete states of the program by calculating a fixpoint over the abstract states. Abstract states form the set $\hat{S} = \mathbf{Loc} \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Store}}$, consisting of an abstract register valuation $\widehat{\mathbf{Val}} := V \rightarrow \hat{A}$ and an abstract store $\widehat{\mathbf{Store}} := \hat{A} \rightarrow \hat{A}$. The initial state at the entry point of the executable is initialized to $(\ell_{\mathbf{start}}, \{\mathit{esp} \rightarrow (\mathsf{stack}, 0)\}, \{(\mathsf{stack}, 0) \rightarrow \ell_{\mathbf{end}}, (\mathsf{global}, a_0) \rightarrow d_0), \ldots, (\mathsf{global}, a_n) \rightarrow d_n)\})$, where $a_0, \ldots, a_n$ denote static data locations in the executable (e.g., initial values for global variables, integer or string constants) and $d_0, \ldots, d_n$ their respective values. Location $\ell_{\mathbf{end}}$ points to a halt statement that catches control flow when the main procedure returns, the esp register is initialized to point to this return address on the stack. All registers and memory locations (including all offsets in all heap regions) not shown are implicitly set to $(\top_R, \top_{32})$.

Our analysis is path sensitive, i.e., it does not join abstract states when control flow recombines after a conditional block or loop. To ensure termination, we introduce bounds on the number of values tracked for each register and memory location (hence the name *Bounded Address Tracking*). In particular, the analysis bounds the number of abstract addresses *per variable per location* that it explicitly tracks and performs widening in two steps. Before calculating abstract successors for a state s at location ℓ, the analysis checks for each register or memory location x whether the total number of unique abstract values for x in all reached states at ℓ exceeds the configurable bound k. If it does, then the value of x is widened to $(r, \top_{32})$, where r is the memory region of x in s. If the number of unique memory regions also exceeds the bound k, then x is widened to $(\top_R, \top_{32})$ (see BOUND rule in Figure 5).

Consider the example code in Figure 4. The single initial abstract state is $(0, \{x \rightarrow (\top_R, \top_{32}), b \rightarrow (\top_R, \top_{32})\}, \emptyset)$, so there is one unique value per variable. We choose to set the bound k to 5. After creating a new abstract heap region and copying the pointer into b, the analysis enumerates states in the loop $2, 3, 4$ while the edge $(4, \mathsf{assume}\ x \geq b + 100, 5)$ remains infeasible. When the state $(2, \{x \rightarrow (\mathsf{global}, 5), b \rightarrow (\mathsf{global}, 0)\}, \{(\mathsf{alloc}_1, 0) \rightarrow (\mathsf{global}, 0), \ldots\})$ is reached, the analysis counts 6 unique values for x in location 2, and widens x to $(\mathsf{alloc}_1, \top_{32})$. This causes a weak update to alloc_1 once x is dereferenced. At the end of the loop, both assume edges are now feasible, and the analysis reaches a fixpoint.

The abstract semantics of Bounded Address Tracking is given using the abstract evaluation operator $\widehat{\mathbf{eval}} :: \mathbf{Exp} \rightarrow \hat{S} \rightarrow \hat{A}$, the bounding operator $\mathbf{bound} :: \hat{S} \rightarrow (V \cup \hat{A}) \rightarrow \hat{S}$, and the abstract transfer function $\widehat{\mathbf{post}} :: \mathbf{Stmt} \rightarrow \hat{S} \rightarrow 2^{\hat{S}}$ from statements and abstract states to sets of abstract states defined in Figure 5. A worklist algorithm extended to apply and adapt precision information [11] (in our case bounds over the number of abstract values) enforces the bound for all registers and memory locations before calculating the abstract transfer function.

Global addresses (global, n) are absolute integers and thus expressions over them are calculated concretely (first case of EVALOP). Addresses for other regions have no statically known absolute value, so only additions of positive or negative integers to their offset can be precisely modeled (second and third case); if pointers to different regions are added or pointers are involved in other types of expressions (including comparisons), the resulting abstract value is safely over-approximated to $(\top_R, \top_{32})$ (fourth case). Other operations (bit extraction, sign extension, etc.) are interpreted analogously. Explicit nondeterminism in expressions evaluates to $(\top_R, \top_{32})$, and memory reads are interpreted by joining the values stored at the addresses in the concretization of the abstract pointer.

A register assignment is interpreted concretely and replaces an existing mapping in the new abstract state. For an assignment to a memory location (i.e., an assignment to a dereferenced pointer), we distinguish three cases depending on the abstract value of the pointer. We can perform a:

- *Strong update*, if both region and offset of the pointer are known. A strong update allows to replace the old value of the memory location in the new state.
- *Weak update to a single region*, if the region of the pointer is known but the offset is $\top_{32}$. Since the precise offset is not known, all memory locations in the region *may* hold the new value, so the existing values have to be joined

$$\textsc{EvalOp} \quad \widehat{\mathbf{eval}}\llbracket e_1 \odot e_2 \rrbracket(s) := \mathsf{let}(r_1, o_1) := \widehat{\mathbf{eval}}\llbracket e_1 \rrbracket(s), (r_2, o_2) := \widehat{\mathbf{eval}}\llbracket e_2 \rrbracket(s) \begin{cases} (\mathsf{global}, o_1 \odot o_2) & \text{if } \odot \text{ not } + \text{ and } r_1 = \mathsf{global} \wedge r_2 = \mathsf{global} \\ (r_1, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_2 = \mathsf{global} \\ (r_2, o_1 + o_2) & \text{if } \odot \text{ is } + \text{ and } r_1 = \mathsf{global} \\ (\top_R, \top_{32}) & \text{otherwise} \end{cases}$$

$$\textsc{EvalNondet} \quad \widehat{\mathbf{eval}}\llbracket nondet \rrbracket(s) := (\top_R, \top_{32})$$

$$\textsc{EvalMem} \quad \widehat{\mathbf{eval}}\llbracket m[e] \rrbracket(s) := \mathsf{let}(r, o) := \widehat{\mathbf{eval}}\llbracket e \rrbracket(s) \begin{cases} s(\hat{m}[r, o]) & \text{if } r \neq \top_R \wedge o \neq \top_{32} \\ \bigsqcup_{i \in \mathbb{I}_{32}} s(\hat{m}[r, i]) & \text{if } r \neq \top_R \wedge o = \top_{32} \\ (\top_R, \top_{32}) & \text{if } r = \top_R \wedge o = \top_{32} \end{cases}$$

$$\textsc{Bound} \quad \mathbf{bound}(s, x) := \begin{cases} s & \text{if } |\{s(x) \mid s \in \{s' | s'(pc) = \ell\}\}| \leq k \\ s[x \mapsto (\top_R, \top_{32})] & \text{if } |\{r \mid (r, o) = s'(x). s' \in \{s'' | s''(pc) = \ell\}\}| > k \\ \mathsf{let}(r, o) = s(x). s[x \mapsto (r, \top_{32})] & \text{otherwise} \end{cases}$$

$$\textsc{AssignReg} \quad \widehat{\mathbf{post}}\llbracket [v := e]^{\ell}_{\ell'} \rrbracket(s) := \left\{ s[v \mapsto \widehat{\mathbf{eval}}\llbracket e \rrbracket(s)][pc \mapsto \ell'] \right\}$$

$$\textsc{AssignMem} \quad \widehat{\mathbf{post}}\llbracket [m[e_1] := e_2]^{\ell}_{\ell'} \rrbracket(s) := \mathsf{let}(r, o) := \widehat{\mathbf{eval}}\llbracket e_1 \rrbracket(s),\ a := \widehat{\mathbf{eval}}\llbracket e_2 \rrbracket(s),\ s' := s[pc \mapsto \ell']$$

$$\begin{array}{r} \textit{strong update} \\ \textit{weak update single region} \\ \textit{weak update all regions} \end{array} \begin{cases} \{s'[\hat{m}[r, o] \mapsto a]\} & \text{if } r \neq \top_R \wedge o \neq \top_{32} \\ \{s'[\hat{m}[r, i] \mapsto s(\hat{m}[r, i]) \sqcup a][\ldots] \text{ for all } i \in \mathbb{I}_{32}\} & \text{if } r \neq \top_R \wedge o = \top_{32} \\ \{s'[\hat{m}[r, i] \mapsto s(\hat{m}[j, i]) \sqcup a][\ldots] \text{ for all } j \in \mathbf{R}, i \in \mathbb{I}_{32}\} & \text{if } r = \top_R \end{cases}$$

$$\textsc{Alloc} \quad \widehat{\mathbf{post}}\llbracket [\mathsf{alloc}\ v, e]^{\ell}_{\ell'} \rrbracket(s) := \left\{ s[v \mapsto (r, 0)][pc \mapsto \ell'] \text{ where } r \text{ is a fresh region identifier} \right\}$$

$$\textsc{Free} \quad \widehat{\mathbf{post}}\llbracket [\mathsf{free}\ v]^{\ell}_{\ell'} \rrbracket(s) := \mathsf{let}(r, o) := s(v),\ s' := s[pc \mapsto \ell'] \begin{cases} \emptyset \text{ (raise error)} & \text{if } r = \top_R \vee o \neq 0 \\ \{s'[\hat{m}[r, i] \mapsto (\top_R, \top_{32})][\ldots] \text{ for all } i \in \mathbb{I}_{32}\} & \text{otherwise} \end{cases}$$

$$\textsc{Assume} \quad \widehat{\mathbf{post}}\llbracket [\mathsf{assume}\ e]^{\ell}_{\ell'} \rrbracket(s) := \begin{cases} \emptyset & \text{if } \widehat{\mathbf{eval}}\llbracket e \rrbracket(s) = (\mathsf{global}, 0) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases}$$

$$\textsc{Assert} \quad \widehat{\mathbf{post}}\llbracket [\mathsf{assert}\ e]^{\ell}_{\ell'} \rrbracket(s) := \begin{cases} \emptyset \text{ (raise error)} & \text{if } \widehat{\mathbf{eval}}\llbracket e \rrbracket(s) = (\mathsf{global}, 0) \\ \{s[pc \mapsto \ell']\} & \text{otherwise} \end{cases}$$

$$\textsc{Havoc} \quad \widehat{\mathbf{post}}\llbracket [\mathsf{havoc}\ v <_u n]^{\ell}_{\ell'} \rrbracket(s) := \left\{ s[v \mapsto (\mathsf{global}, i)][pc \mapsto \ell'] \mid i <_u n, i \in \mathbb{I}_{32} \right\}$$

Fig. 5. Definition of abstract evaluation and abstract post operators for Bounded Address Tracking.

with the new value (with respect to the lattice of abstract addresses shown in Figure 3).

Note that this rule makes the assumption that a memory write to a specific region never exceeds the bounds to write to an adjacent heap regions, since the goal of this work is not to prove memory safety but check API specifications. For full soundness, however, we would have to perform a weak update to all regions.

- *Weak update to all regions*, if neither region nor offset of the pointer are known. All memory locations in all regions have to be joined with the new value.

  In practice, the state becomes too imprecise to continue analysis. In particular, all return addresses will be affected by the weak update. Our implementation thus signals an error for writing to an unknown (possibly also null) pointer in this case.

Besides the fact that region and offset have to be known, there is another prerequisite for performing strong updates: The region of the pointer must not be a *summary region*, i.e., on all execution paths, the abstract region corresponds only to one concrete memory region [7]. Our analysis never creates summary regions, which can be seen from the ALLOC rule in Figure 5. New regions are tagged with fresh, unique identifiers. The only way the abstract region value of a pointer can represent multiple regions is if the number of regions for the pointer exceeds the value bound k and is joined to $\top_R$. In this case, a weak update to all regions will be performed when the pointer is dereferenced, which is a sound over-approximation for an assignment to a summary region.

The abstract post operator for free sets all memory locations in the freed region to $(\top_R, \top_{32})$. The abstract semantics for assume and assert is similar to the concrete case and only adapted to the abstract address model. The abstract post for havoc is the only implementation that returns a non-singleton set: It splits abstract states by enumerating absolute integer values for the given register.

C. Abstraction of Nondeterminism

Abstraction by approximating multiple concrete program states with abstract states is the key to achieving scalability of an analysis. In static analysis, abstraction is introduced by choosing a suitable abstract domain for the program to be analyzed. In software model checking, an iterative refinement finds a suitable abstraction by adding new predicates over program variables. Control flow reconstruction from binaries requires concrete values for jump targets, however, and the lack of types requires precise values for pointer offsets. Therefore, existing mechanisms for abstraction are not well-suited for a precise analysis of binaries. Still, abstraction has to be introduced to make the analysis feasible.

Even though Bounded Address Tracking resembles software model checking in the way that states from different paths are not merged, it allows registers and memory locations to be *unknown*, i.e., set to $(\top_R, \top_{32})$. This is especially useful when representing nondeterminism in the execution environment (e.g., input, unspecified behavior). Setting parts of the state to *unknown* avoids an exponential enumeration of value combinations. When designing the environment model for a program, we often have a good idea of what needs to be precisely modeled and where we can safely over-approximate. For instance, the standard calling convention of the Windows API specifies that upon return the contents of registers eax, ecx, and edx are undefined. Enumerating all possible values for the registers in a full explicit state exploration would require creating 2^{96} states. By abstracting the nondeterministic choice of values to $(\top_R, \top_{32})$ for all three registers, we only need a single abstract state. It is extremely unlikely to produce a spurious counterexample from this abstraction, since code should not depend on undefined side-effects.

On the other hand, there are occasions when abstracting to an unknown value increases the requirements for the abstract domain. Consider the following code, which is a stub for the Windows API function IoCreateSymbolicLink:

```
int choice = nondet32;           mov eax, nondet32
if (choice == 0)                 neg eax
 return STATUS_SUCCESS;          sbb eax, eax
else                             and eax, 0xC0000001
 return STATUS_UNSUCCESSFUL;     ret
```

Here, the compiler replaced the if-statement with bit-manipulation of the return value. Our abstract domain can only deduce that eax is $(\top_R, \top_{32})$ at the return statement, even though eax actually can be only either `0` or `0xC0000001`. Therefore we added the havoc statement to the IL; it causes the analysis to generate multiple successor states with different integer values for a register (HAVOC in Figure 5). With it, we can change the first line of the stub to `int choice; havoc(choice, 1)`. This causes the analysis to create two states; one with eax set to 0, and one with eax set to 1. From these states it can easily compute the two possible states at the return statement: In the first case eax becomes `0`, in the second case `0xC0000001`.

IV. IMPLEMENTATION

We have implemented the architecture and approach described in this paper in our binary analysis platform JAKSTAB (**Ja**va tool**k**it for **stat**ic analysis of **b**inaries). As input, JAKSTAB is able to process Windows PE files (the format used in 32-bit Windows for `.exe`, `.dll`, `.sys`, and more), unlinked COFF object files, and Linux ELF executables. It can load an executable in combination with multiple dynamic libraries and will resolve dependencies between the files.

A. Instruction Sets

JAKSTAB currently supports only the x86 architecture, but can be extended to other architectures by supplying an opcode table and a description of instruction semantics. Instructions are specified using the semantic specification language of the Boomerang decompiler [12], [13]. We used Boomerang's existing x86 specifications as a starting point, which we rewrote and extended heavily.

Our current description of x86 instruction semantics covers over 500 instructions, which includes all instructions that we encountered in the executables analyzed during the experiments. Large parts of the floating point instruction set and the various SSE extensions are supported. The instruction semantics are specified on the level of registers and flags, I/O instructions are specified to read nondeterministic values.

B. Analysis Architecture

JAKSTAB's analysis architecture is based on the Configurable Program Analysis API by Beyer et al. [9], [11], which allows to seamlessly combine state splitting and state joining analyses such as predicate abstraction and interval analysis, respectively. For the work described in this paper, we used only our Bounded Address Tracking domain combined with the trivial location domain that expands the state space of the program to at least one state per IL statement.

C. OS Abstraction and Driver Harness

Executables in general and drivers in particular frequently interact with the operating system. As in source based analyses, we abstract system calls using stubs, which model the relevant side effects such as memory allocation or the execution of callback routines. Following the approach of the source code software model checker SDV [2], we load the driver into JAKSTAB together with a separate *harness* module, that includes system call abstractions relevant to drivers and contains a main function that nondeterministically exercises the driver's initialization and dispatch routines. The harness is written in C and compiled into a dynamic library (DLL) for loading; it is based on SDV's `osmodel.c` and follows SDV's invocation scheme for plug&play drivers. For our experiments, we manually encoded specifications in the harness by inserting state variables and assertions at the locations where SDV places hooks into its specification files.

Several parts of the SDV harness and rules had to be modified to make it suitable for binary analysis. For example, the preprocessor macro IoMarkIrpPending, which sets a bit in the control word of interrupt request packets (IRPs), is intercepted by SDV to change the state for the *PendedCompletedRequest* rule. Since macro invocations are no longer explicit in the binary, we had to modify the rule's assertion to check the bit directly instead of a separate state variable. Furthermore, we replaced SDV's statement for nondeterminism by either havoc or $nondet$, depending on the context.

The IL statements alloc, free, havoc, and assert are exclusively generated by the harness, since they do not correspond to any real x86 instructions. These statements are encoded

	DDA/x86			Jakstab					
Driver	**Instr**	**Time**	**Result**	**k**	**$\mathbf{k_h}$**	**States**	**Instr**	**Time**	**Result**
vdd/dosioctl/krnldrvr/krnldrvr.sys	2824	14s	✓	28	5	378	413	2s	✓
general/ioctl/sys/sioctl.sys	3504	13s	✓	28	5	3947	630	7s	✓
general/tracedrv/tracedrv/tracedrv.sys	3719	16s	✓	28	5	486	439	2s	✓
general/cancel/startio/cancel.sys	3861	12s	✓	28	5	633	759	2s	✓
general/cancel/sys/cancel.sys	4045	10s	✓	28	5	600	780	2s	✓
input/moufiltr/moufiltr.sys	4175	3m 3s	×	28	5	3830	722	9s	×
general/event/sys/event.sys	4215	20s	✓	28	5	663	690	2s	✓
input/kbfiltr/kbfiltr.sys	4228	2m 53s	×	28	5	3834	726	8s	×
general/toaster/toastmon/toastmon.sys	6261	4m 1s	✓	28	25	4853	977	9s	✓
storage/filters/diskperf/diskperf.sys	6584	3m 17s	✓	28	5	19772	1409	46s	✓
network/modem/fakemodem/fakemodem.sys	8747	11m 6s	✓	28	5	13994	1887	24s	$\times_m$
storage/fdc/flpydisk/flpydisk.sys	12752	1h 6m	FP	100	35	186543	1782	39m34s	✓
input/mouclass/mouclass.sys	13380	40m 26s	FP	28	28	3055	1763	8s	FP$_c$
input/mouser/sermouse.sys	13989	1h 4m	FP	28	28	1888	1293	4s	FP
kernel/serenum/SerEnum.sys	14123	19m 41s	✓	28	25	5213	1503	8s	✓
wdm/1394/driver/1394diag/1394DIAG.sys	23430	1h33m	FP	28	28	2181	2426	4s	FP$_m$
wdm/1394/driver/1394vdev/1394VDEV.sys	23456	1h38m	FP	28	28	2837	2872	5s	FP$_m$

Fig. 6. Comparison of experimental results on Windows DDK drivers between DDA/x86 (on a 3GHz Xeon) and Jakstab (on a 3GHz Opteron).

into the compiled harness object file using illegal instructions, which are directly mapped to the corresponding IL statements during disassembly. For instance, an alloc statement can be generated from the C source of the harness by inlining the assembly instruction `lock rep inc eax`.

V. Experiments

For direct comparison with the IDA Pro and CodeSurfer/x86 based binary driver analyzer DDA/x86 described in [1], we ran Jakstab on the same set of drivers from the Windows Driver Development Kit (DDK) release 3790.1830 and checked the same specification *PendedCompletedRequest*. The rule specifies that a driver must not call IoCompleteRequest and return STATUS_PENDING unless it invokes the IoMarkIrpPending macro on the IRP being processed. We compiled the drivers without debug information using default settings. Note that unlike [1], we did not compile and link the driver source code against the harness; our approach is directly applicable to drivers without access to source code.

Our experimental results are listed alongside those reported in [1] in Figure 6. The number of instructions include instructions from the harness in both cases. Note that the tools report very different numbers of instructions for the same binaries; this is due to the fact that Jakstab disassembles instructions only on demand, i.e., if they are reachable by the analysis. In contrast, CodeSurfer/x86 uses IDA Pro as front end, which heuristically disassembles all likely instructions in the executable. Since for DDA/x86 the entire harness was compiled and linked with the driver, IDA Pro disassembled all code from the harness, including code that is unreachable from the driver under analysis. Conversely, it is possible that some driver code is unreachable from the harness. For the experiments we used two value bounds which we determined empirically; k shows the value bound for registers and stack locations, k_h the value bound for memory locations in allocated heap regions.

For `flpydisk.sys`, Jakstab was able to verify the specification, while DDA/x86 found a false positive (FP). This is due to the only limited degree of path sensitivity in DDA/x86, which follows the ESP approach [14] for differentiating paths based on states of a property automaton. In [1], the property automaton is extended to track updates to the variable holding the return value, but it can miss updates due to its heuristic for detecting interprocedural dependencies for the return value.

In `fakemodem.sys`, Jakstab encountered a potentially unsafe memory access (marked as $\times_m$), where an uninitialized value, i.e., $(\top_R, \top_{32})$, is used as the index for a write to an array. We manually confirmed the feasibility of the error trace for the execution environment simulated by the harness. DDA/x86 does not check for memory safety due to the large number of false positives [1], so it did not detect this bug. As mentioned in Section III-B, our analysis signals an error on weak updates to all regions. This amounts to implicitly checking for write accesses to uninitialized pointers, which allows Jakstab to detect the error. As a consequence of building on the SDV harness, which is not designed for checking memory safety and often omits proper pointer allocation, our analysis yielded false positives where the result shows FP$_m$ in Figure 6. In `mouclass.sys`, a switch jump could not be resolved because the switch variable was over-approximated leading to a false positive of invalid control flow (FP$_c$). Currently, we manually investigate abstract error traces and extend the harness if necessary to eliminate false positives. We leave a partial or full automation for future work.

The comparison of execution times should be taken with a grain of salt, since both prototypes were run on different machines. DDA/x86 was run on a 64-bit Xeon 3GHz processor with 4GB of memory per process, while the experiments with

JAKSTAB were conducted on a 64-bit AMD Opteron 3GHz processor with 4GB of Java heap space (we report the average time of 10 runs per driver). Still, it is possible to see that execution times for JAKSTAB appear favorable overall.

We do not have to recompile and link drivers with the harness, so we were able to extend our experiments beyond the Windows DDK. We ran our prototype on all 322 drivers from the `system32\drivers` directory of a regular 32-bit Windows XP desktop system, using $k = 28$ and $k_h = 5$. Besides the *PendedCompletedRequest* rule, we also checked the *CancelSpinLock* rule, which enforces that a global lock is acquired and released in strict alternation. Note that this set of drivers also includes classes of drivers which are not even supported by the SDV harness in source code analysis, such as graphics drivers. Nonetheless, we were able to successfully analyze 28% of these drivers. For 41% of the drivers, analysis failed because of weak global updates, mostly due to missing information about pointer allocation in the harness. In 31% of the cases, the analysis failed due to unknown or erroneous control flow, which can be again caused by unknown side effects of API functions not supported by the harness, or by coarse abstraction of variables used in switch jumps. Two drivers timed out after 1 hour; in three drivers the analysis found potential assertion violations. To our knowledge, this is the first time static analysis was successfully applied to real world, closed source, binary driver executables.

VI. Related Work and Discussion

Bounded Address Tracking was inspired by the Explicit Analysis of Beyer et al. [11], which tracks explicit values of integer variables of C programs up to a certain bound. In their work, explicit analysis is used for cheap enumeration of values for a variable before it is modeled by the computationally more expensive predicate abstraction.

As pointed out already, the CodeSurfer/x86 project is most closely related to our work and faces similar challenges. The major differences in approach are that CodeSurfer/x86 is implemented on top of the heuristics based IDA Pro, and that its analyses (in particular Value Set Analysis (VSA) [6]) are based on more "classic" static analyses such as interval analysis. VSA is path insensitive and thus requires the use of call strings for reasonable results. Call strings, however, are tied to the concept of procedures (which is unreliable in x86 assembly) and assume the existence of a separate call stack. This issue lead us to the design of the bounded path sensitive analysis presented in this paper.

Balakrishnan and Reps generally rely on summary nodes for representing heap objects. They reduce the number of weak updates by introducing a *recency abstraction* [15] of heap nodes. Their approach extends the common paradigm of using one summary node per allocation site (i.e., address of the call to malloc), by splitting this summary node into (i) the region most recently allocated in the current execution path and (ii) a summary node for the remaining regions. In contrast, our approach instead explicitly discriminates allocated regions up to the value bound.

VII. Summary

In this paper, we presented a framework for precise static analysis of driver binaries. Compared to existing approaches, it significantly reduces the sources of unsoundness by eliminating the separate, error-prone disassembly step. We introduced Bounded Address Tracking, an abstract domain which allows strong updates to memory locations on the heap, as long as the number of different pointer values stays below a definable bound. Experiments on several driver binaries confirm the feasibility of our approach on small, but real world code and demonstrate its improved performance compared to state of the art approaches in spite of increased precision. Moreover, we tried our approach on all drivers of a regular desktop system and achieved encouraging results.

For scaling up to larger programs, however, we will attempt to reduce precision where it is not required. One approach is to reduce the value bound individually for variables not involved with control flow or specifications. Starting from a generally low bound, an iterative refinement loop can help to identify memory locations and function stubs in the harness where increased precision is required. Furthermore, we will investigate the use of summaries that do not require assumptions on procedure structure or calling conventions.

Acknowledgments

The authors would like to thank Vlad Levin for discussing SDV, Gogul Balakrishnan for feedback on DDA/x86, and Peter Bokor and the anonymous reviewers for their detailed comments on the paper. This work was supported by CASED.

References

[1] G. Balakrishnan and T. Reps, "Analyzing stripped device-driver executables," in *TACAS*, ser. LNCS. Springer, 2008, pp. 124–140.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *Proc. 2006 EuroSys Conf.* ACM, 2006, pp. 73–85.

[3] Microsoft. Windows Driver Kit documentation. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff557573(VS.85).aspx

[4] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *WCRE*. IEEE Computer Society, 2002, pp. 45–54.

[5] Hex-Rays SA. IDA Pro. [Online]. Available: http://www.hex-rays.com/idapro/ [Accessed: July 26, 2010]

[6] G. Balakrishnan and T. W. Reps, "Analyzing memory accesses in x86 executables." in *CC*, ser. LNCS, vol. 2985. Springer, 2004, pp. 5–23.

[7] D. R. Chase, M. N. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *PLDI*, 1990, pp. 296–310.

[8] J. Kinder, H. Veith, and F. Zuleger, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *VMCAI*, ser. LNCS, vol. 5403. Springer, 2009, pp. 214–228.

[9] D. Beyer, T. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, pp. 504–518.

[10] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 423–427.

[11] D. Beyer, T. Henzinger, and G. Théoduloz, "Program analysis with dynamic precision adjustment," in *ASE*. IEEE, 2008, pp. 29–38.

[12] M. van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery." in *WCRE*. IEEE Computer Society, 2004, pp. 27–36.

[13] C. Cifuentes and S. Sendall, "Specifying the semantics of machine instructions." in *IWPC*. IEEE Computer Society, 1998, pp. 126–133.

[14] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *PLDI*. ACM, 2002, pp. 57–68.

[15] G. Balakrishnan and T. Reps, "Recency-abstraction for heap-allocated storage," in *SAS*, ser. LNCS, vol. 4134. Springer, 2006, pp. 221–239.