Universidad Nacional de Rosario

Facultad de Ciencias Exactas,
Ingeniería y Agrimensura

Tesina de Licenciatura en Ciencias de la Computación
Agosto 2012

Tipógrafo: Infiriendo el grafo de tipos de un programa binario

Autor:
Gustavo Grieco
Legajo G-3488/6
Universidad Nacional de Rosario
Rosario, Argentina

Supervisor:
Dr. Juan Caballero

Instituto IMDEA Software
Madrid, España

Esta página se dejó intencionalmente en blanco.

Agradecimientos

A mi director, Juan Caballero, cuya extensa dedicación y paciencia con este trabajo (¡y conmigo!) me proporcionaron una ayuda invaluable para esta tesina de grado. También a mi querida familia, por su apoyo y contención, sin los cuales nunca hubiera tenido la oportunidad de elegir mi carrera universitaria sin presiones ni compromisos.
Además, a los docentes y alumnos de la carrera, cuyas incontables horas en el pequeño pero familiar Departamento de Ciencias de Computación me inspiraron en un amplio rango de temas académicos y no tan académicos. Y finalmente, a mi amigos, locales y extranjeros que me apoyaron en distintos ámbitos de mi vida durante estos años de carrera y esta tesina.

Índice

1. Introducción

Los tipos de datos son fundamentales para entender cómo un programa funciona: sus entradas, salidas y pasos intermedios. Su análisis tiene un amplio rango de aplicaciones en depuración de programas [13], optimización [17,19,20] y seguridad [1,11,12,22]. Todas estas aplicaciones requieren conocer las propiedades básicas de las estructuras de datos definidas y utilizadas por un programa: sus campos, estructura y cómo éstos se relacionan entre si.

Si el código fuente de un programa está disponible, el análisis de sus estructuras de datos puede ser una tarea relativamente sencilla. Sin embargo, los programas son usualmente ensamblados en código binario. Cuando esto sucede, el compilador remueve prácticamente toda la información que se necesita para realizar el análisis de las estructuras de datos y optimiza el código ensamblador ofuscando la naturaleza real de cómo el programa utiliza sus estructuras de datos.

Desafortunadamente, algunas veces el análisis de programas binarios es la única posibilidad debido a que no todos los programas tienen su código fuente disponible. Por ejemplo, un analista externo no tiene acceso al código fuente de un programa porque el fabricante desea proteger la propiedad intelectual que contiene. Además, para los programas como el malware, la fuente rara vez está disponible para alguien que quiera conocer qué hacen estos programas malignos.

La recuperación manual de la información de tipos de un programa binario es una tarea difícil, lenta y susceptible a errores [15]. Para poder definir un procedimiento que realiza esta tarea automáticamente debemos combinar metódicamente cada pieza de información disponible para poder obtener una imagen global de los tipos de datos definidos y utilizados en un programa.

Contribuciones. Contribuimos al estado del arte con un formalismo y una implementación de un conjunto de técnicas dinámicas que nos permiten recuperar la información de las estructuras de datos, desde sus campos individuales hasta estructuras de datos complejas sin acceso al código fuente o información de depuración. Proponemos tres algoritmos de tipado dinámico para recuperar tipos primitivos y los comparamos con nuestra implementación de un algoritmo [23] del estado del arte de la recuperación de tipos.

Nuestro formalismo se define de manera abstracta con un número minimal de suposiciones claramente establecidas. A pesar de que los ejemplos y resultados se muestran utilizando x86 cómo la arquitectura elegida, los conceptos fundamentales de este formalismo son abstraídos permitiéndonos modificarlos fácilmente para cambiarlos o extenderlos si fuera necesario.

Diseñamos e implementamos Tipógrafo, una herramienta para recuperar estructuras de datos utilizadas por un programa. Tipógrafo trabaja de extremo a extremo analizando una o más trazas de ejecución y combinando la información de tipos deducida de ellas.

Evaluamos Tipógrafo usando dos programas de ejemplo y un complejo mensajero instantáneo de código abierto, llamado Pidgin [24]. Utilizamos estos programas porque tiene su código fuente disponible para evaluar el resultado de la recuperación de tipos. Nuestros algoritmos muestran una mejora de hasta 60 % en la evaluación sobre cuánta memoria reservada es tipada, comparados con el trabajo previo, eliminando casi totalmente los conflictos de tipos detectados en las ejecuciones de los programas evaluados. Además, los grafos de tipos inferidos se corresponden casi totalmente con los tipos definidos en sus códigos fuentes.

Finalmente, sugerimos algunas extensiones y mejoras para continuar desarrollando la inferencia del grafo de tipos además de algunas aplicaciones concretas usando nuestro formalismo.

2. Visión general

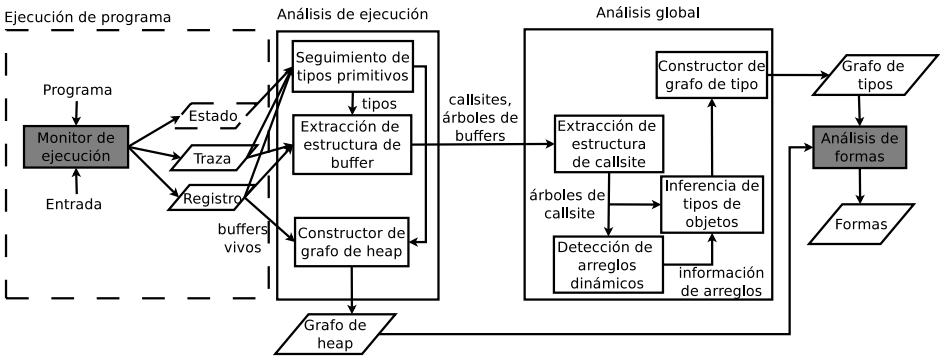


Figura 1: Visión general del proceso de recuperación del grafo de tipos.

Un *grafo de tipos* es un grafo con nodos como tipos de datos y aristas cómo punteros entre esos tipos de datos. Estos grafos capturan la esencia sobre cómo los tipos del programa se relacionan entre sí.

El proceso de inferencia del grafo de tipo se compone de tres fases: la *ejecución del programa*, el *análisis de ejecución* y el *análisis global*. Durante la fase de ejecución del programa, se recolectan datos de bajo nivel sobre la ejecución del programa. En la próxima fase, el análisis de ejecución, se realiza una recuperación estructural de cada buffer reservado durante la ejecución. Finalmente, en la fase de análisis global se recupera el grafo de tipos. Es importante notar que las dos primeras fases pueden ser realizadas múltiples veces utilizando diferentes entradas en el programa analizado. El grafo de tipo producido por la implementación puede acumular la información de múltiples trazas de ejecución.

2.1. Ejecución de Programa

El proceso de recuperación de tipos de datos comienza con información de bajo nivel sobre la ejecución de un programa dada una entrada. En la primera fase, *ejecución de programa*, esta información es recolectada usando un conjunto de herramientas provistas por la plataforma de análisis binario de BitBlaze [29]. El componente de análisis dinámico en BitBlaze es un emulador completo llamado TEMU [29]. TEMU utiliza un plugin llamado Tracecap [8] para obtener una traza de ejecución, una copia del estado de la CPU y un registro de reserva de memoria. La traza de ejecución contiene información sobre las instrucciones ejecutadas y el contenido de sus operandos. La copia del estado de la CPU almacena el contenido del espacio de memoria del proceso y los registros de CPU en un punto de la ejecución. El registro de reserva de memoria contiene la información manejo de memoria de un proceso cuando este reserva o libera memoria, capturando qué regiones de memoria son asignadas por el sistema operativo a un proceso en algún punto de su ejecución.

Buffers y objetos. En este trabajo, un punto de partida para recuperar tipos de datos de cada objeto creado y usado por un programa será saber dónde están almacenados estos objetos en la

memoria. Un programa sólo puede almacenar objetos durante su ejecución en regiones contiguas de memoria reservada. Estas regiones se conocen como *buffers*. Usando este sencillo concepto, un principio clave en el presente trabajo será que cada objeto, primitivo o compuesto se almacena en un solo buffer.

2.2. Análisis de Ejecución

La segunda fase de este proceso se denomina *análisis de ejecución*, en la cual la información de tipos primitivos y estructura de buffer se reconstruye de una traza de ejecución. Esta fase puede ser realizada múltiples veces sobre diferentes trazas de un mismo programa.

Un primer paso para inferir los tipos de los objetos almacenados en los buffers es la detección de tipos primitivos en estos buffers incluyendo enteros con signo, enteros sin signo y flotantes. Un tipo primitivo es una propiedad de una región de un buffer determinada por cómo el programa opera con los bytes de esa región. Por ejemplo a través de operaciones aritméticas, flotantes o de indirección. Cada tipo primitivo detectado en un buffer puede ser interpretado como un campo en un objeto contenido en dicho buffer. Los tipos primitivos son detectados en el módulo de *seguimiento de tipos primitivos*.

Otro importante paso en esta etapa es realizado en el módulo de *extracción de estructura de buffer*. Cada vez que un buffer se libera, su estructura es reconstruida insertando los tipos primitivos inferidos como campos en una estructura jerárquica llamada *árbol de buffer*.

2.3. Análisis Global

La tercera fase se denomina *análisis global*. En esta fase, la herramienta agrupa los árboles de buffer que representan objetos del mismo tipo.

En la fase de *ejecución de programa*, el monitor de ejecución identificó cada llamada para reservar buffers y su callsite, o sea la dirección de memoria del llamado de la función que reserva memoria. Un callsite puede proveer una definición laxa de tipos ya que asumiremos que dos buffers reservados en el mismo callsite contienen objetos del mismo tipo. Esto nos permite combinar los árboles de buffers correspondientes a buffers que tienen el mismo callsite en una estructura análoga al árbol de buffers pero, para cada callsite, llamada *árbol de callsite*. De todas maneras, objetos del mismo tipo todavía podrían ser reservados en distintos callsites, por lo tanto el módulo de *inferencia de tipos de objetos* agrupará los árboles de callsites de acuerdo a la información provistas por sus punteros, produciendo *árboles de tipos*.

El *análisis global* también identifica arreglos dinámicos analizando árboles de buffers del mismo callsite pero diferente tamaño. Finalmente, el *constructor de grafo de tipo* toma los árboles de tipos y genera un grafo de tipos de la ejecución de un programa acumulando la información inferida de múltiples trazas.

El grafo de tipos inferidos puede ser usado como entrada para herramientas de análisis de formas, las cuales pueden ser capaces de identificar estructuras de alto nivel como árboles o listas.

3. Trabajos relacionados

La recuperación de tipos y estructura de un programa sin acceso a su código fuente o información de depuración es un área relativamente joven. Los primeros enfoques para abordar este problema utilizan análisis estático para recuperar información que pueda ser usadas para detectar los tipos de un programa binario. En uno de los primeros trabajos, Ramalingam et al. definen un algoritmo llamado Indentificación de Estructuras Agregadas [26] que utiliza análisis estático para detectar elementos atómicos en estructuras agregadas (por ejemplo, registros, arreglos), basándose en los patrones de acceso de un programa COBOL.

Luego, Balakrishnan y Reps introducen una técnica llamada análisis de conjunto de valores [2] (o VSA por sus siglas en inglés). Esta técnica permite determinar un superconjunto de todos los valores posibles de una variable en un programa binario. Los autores implementan dicha técnica en una herramienta llamada CodeSurfer/x86.

Cozzie et al. presentan Laika [12], un formalismo para recuperar estructuras de datos de copias de memoria de un programa. Sus técnicas infieren tipos básicos para cada byte en la memoria basado en su valor. También utiliza los valores de punteros inferidos para estimar dónde los objetos comienzan. Esta información es usada como entrada de un clasificador bayesiano no supervisado para inferir las estructuras de datos usadas. Los autores de este trabajo también muestran como esta técnica puede ser usada para detectar virus polimórficos. Desafortunadamente, Laika sólo funciona con la copia de la memoria en un instante. Debido a que solamente los valores de la memoria están disponibles, la precisión de la recuperación de tipos es bastante limitada. Además, no disponen de un procedimiento para combinar la información de varias imágenes de memorias.

Algunas técnicas de análisis dinámico también han sido propuestas para extraer información de tipado e información estructural analizando las operaciones realizadas por un programa durante su ejecución. Caballero et al. en su herramienta llamada Dispatcher [10] presentan una técnica dinámica para extraer el formato de un buffer descomponiéndolo en un árbol formado por otros buffers usados para generar su contenido. También presentan un procedimiento para inferir los tipos privados de un programa usando tipos públicos. Por ejemplo, los usados en las funciones de la API de Windows. Los autores también muestran como usar Dispatcher para recuperar el formato de mensajes de protocolos no documentados como los usados por la botnet de MegaD. Dispatcher solamente recupera la estructura de los buffers que contienen mensajes para ser enviados por la red.

Un enfoque similar fue usado por Zhiqiang et al. en REWARDS [23]. Ellos analizan secuencialmente instrucciones de ensamblador en una traza de ejecución para tratar de inferir los tipos de sus operandos. También, la propagación de los tipos inferidos se realiza en cada paso de la traza. REWARDS expande la detección dinámica de tipos para incluir el espacio entero de memoria usado por un programa, en vez de sólo algunos de sus buffers. Los autores utilizan REWARDS para análisis forense de memoria y para descubrimiento automático de vulnerabilidades. En la tesis proponemos dos algoritmos y los comparamos con REWARDS.

Más recientemente, Slowinska et al., presentan Howard [28], otro formalismo basado en análisis dinámico. Howard puede inferir punteros y recuperar campos de estructuras observando como los accesos a las estructuras se realizan. También presentan nuevas técnicas para detectar arreglos. Los autores presentan dos aplicaciones usando Howard: generación automática de símbolos de depuración y defensa contra ataques de desbordamiento de buffer.

Estas dos últimas herramientas, Howard y REWARDS no identifican tipos de estructuras de datos o arreglos del mismo tipo.

La información de tipos y estructural puede ser recuperada estáticamente del código binario de

un programa. Recientemente, Lee et al. presentan TIE [21], un sistema formal para inferir tipos creando y resolviendo un sistema de restricciones de tipos de memoria y registro de CPU. TIE puede analizar estáticamente un programa binario, y sus trazas de ejecución. El enfoque dinámico realizado por este formalismo trabaja convirtiendo las instrucciones de ensamblador de las trazas de ejecución a un lenguaje formal intermedio llamado BIL [4]. Debido a que este lenguaje no es tipado, los autores definen un látice de tipos básicos y generan restricciones de tipos usando variables en el código convertido. Esta técnica no puede usar más de una traza y presenta problemas cuando las inconsistencias de tipos aparecen porque el sistema de restricciones se vuelve insatisfacible.

Otro enfoque relacionado es el tipado de memoria de un programa en un momento determinado. La diferencia con los otros enfoques es que en este las definiciones de tipos del programa están disponibles y el objetivo es asignar estos tipos a a cada uno de los objetos en memoria. `whatsat`, presentado por Polishcuk et al. [25], resuelve un sistema de restricciones para verificar si los tipos de los objetos en memoria de un programa en un punto particular de su ejecución son consistentes o no.

Influencias en nuestro formalismo. Tratando de seguir la metáfora de *pararse en los hombros de gigantes*, el formalismo de nuestro trabajo incorpora y mejora técnicas presentadas en trabajos previos.

Tomamos prestada la idea usar *árboles de formato* para representar las estructuras de datos de Dispatcher. También estamos fuertemente influenciados por REWARDS al proponer nuevo algoritmos utilizados como componentes principales de nuestro sistema de inferencia de tipos. El objetivo de nuestros algoritmos también es deducir y seguir la utilización de tipos primitivos durante la ejecución de un programa.

Nuestro trabajo también incorpora la idea de un látice de TIE con algunos cambios. De todas maneras, no convertimos el código de ensamblador usando un lenguaje intermedio o creamos restricciones de tipos porque la creación de la mismas puede llevar a un sistema insatisfacible si el programa analizado utiliza datos de una manera inconsistente. Si esto ocurre, la detección de tipos fallaría completamente. En cambio, buscamos la posibilidad de detectar conflictos de tipos sin interferir con el tipado de datos usados de manera consistente durante la ejecución de un programa binario.

En `whatsat`, Polishcuk et al. presentan una novedosa idea para definir los tipos en los programa binarios. Utilizan un látice de tipos, pero definidos a nivel de cada byte. Nosotros creemos que este enfoque es más adecuado debido a que datos que ya tienen tipos asignados (por ejemplo constantes o variables) se copian de una locación a otra por bytes.

4. Análisis de Ejecución

La primera fase en el proceso de inferencia se llama *análisis de ejecución*. Comienza con definiciones fundamentales sobre como los tipos son inferidos a partir de las instrucciones de ensamblador. Luego, se explica como los tipos y las estructuras de cada buffer son reconstruidas a partir de una traza de ejecución.

4.1. Definiciones

4.1.1. Locaciones de programa

La formalización de este formalismo para deducir estructuras de datos de alto nivel comienza con la definición de máquina de computación abstracta. Esta máquina abstracta tendrá memoria principal, registros multipropósito y diferentes instrucciones. Debido a que las instrucciones usan la memoria principal, registros o constantes en ellas para leer o escribir bytes que formarán las estructuras de datos, necesitamos formalizar estas posibilidades. Definimos *locaciones de programa* para representar bytes que pueden ser leídos o escritos por un programa. Una locación de programa puede ser:

- Un byte en la memoria: $M(x)$ donde x es una dirección de memoria.
- Un byte en un registro: $R(reg, x)$ donde reg es un nombre de registro y x es un offset de byte.
- Un byte en un valor constante: $I(i, x)$ donde i es una dirección de memoria donde una instrucción es almacenada y x es un offset de byte.

Algunos ejemplos pueden ser útiles para entender la sintaxis. En una abstracción de una CPU x86, tendríamos:

- Un entero de 16-bit almacenado en la dirección de memoria $0x0034dd20$ estaría compuesto por dos locaciones de programa: $M(0x0034dd20)$ y $M(0x0034dd21)$.
- Si tenemos la siguiente instrucción de ensamblador en la dirección $0x00002012$:

  $$\text{addl } \$3, \%\mathbf{ecx}$$

  Entonces el valor constante en esta suma de 32-bit está compuesto por las locaciones $I(0x00002012, 0)$, $I(0x00002012, 1)$, $I(0x00002012, 2)$ y $I(0x00002012, 3)$ y el registro ECX está compuesto por cuatro locaciones de programa: $R(ecx, 0)$, $R(ecx, 1)$, $R(ecx, 2)$ y $R(ecx, 3)$.

Ya que los registros, direcciones de memoria o valores constantes en las instrucciones son usadas en las instrucciones ensamblador como operandos pero no realmente a nivel de bytes individuales, debemos usar usar una función definida sobre el conjunto de los operandos, llamado $\mathbb{O}$, al conjunto de conjuntos de todas las posibles locaciones de programas, llamado $\mathbb{L}$.

$$L : \mathbb{O} \to \mathcal{P}(\mathbb{L})$$

Un ejemplo puede ayudar a entender como se usa esta función. Si tenemos el registro de procesador ECX representado este operador como $ECX \in \mathbb{O}$ entonces:

$$L_{ECX} = \{R(ecx, 0), R(ecx, 1), R(ecx, 2), R(ecx, 3)\}$$

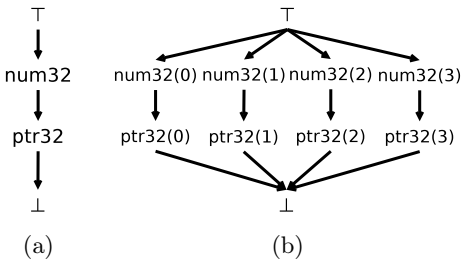


Figura 2: Látice de tipos primitivos

4.1.2. Tipos Primitivos

Las estructuras de datos de alto nivel son usualmente construcciones complejas que involucran datos en diferentes offsets tales como números, caracteres, flotantes, o punteros a otras estructuras de datos. Un primer paso para recuperar estas estructuras es identificar tipos de datos primitivos en locaciones de programa. Definimos el concepto de tipo primitivo para poder formalizar la reconstrucción de estructuras de datos complejas.

Un tipo primitivo es una propiedad de una región de un buffer determinada por como el programa opera con los bytes almacenados en dicha región. Los tipos primitivos pueden darnos una idea, por ejemplo, si el programa está operando en ciertas locaciones de programa, en un instante determinado de su ejecución con un número entero o con un número flotante, o incluso, si el número entero tiene signo o no. Los tipos primitivos son definidos considerando las operaciones que un conjunto de instrucciones de determinada arquitectura realiza, de acuerdo a la documentación provista.

4.1.3. Árboles de Rango

Para formalizar estructuras de datos con campos y estructura jerárquica, vamos a definir el concepto de *árbol de rango*. Un árbol de rango es un árbol con nodos representando estructura interna o campos almacenados entre dos offsets de bytes, o sea un rango. Una arista dirigida existe si y sólo si el rango del nodo de destino está completamente contenido en el rango del nodo fuente. El solapamiento entre los rangos de los nodos no está permitido. El nodo raíz define el tamaño del árbol de rango, teniendo su rango entre cero y el tamaño del árbol menos uno.

Cada nodo en un árbol de rango tiene tres atributos: un rango, un tipo y, opcionalmente, datos. El tipo de un nodo puede ser *estructura*, *arreglo*, *arreglo dinámico*, o *primitivo*.

Los nodos representando arreglos, arreglos dinámicos y primitivos, tienen en su campo de datos un tipo primitivo. Los nodos internos del árbol de rangos no pueden ser tipo primitivo y sólo la raíz puede tener tipo arreglo dinámico.

En el presente trabajo, los árboles de rango son usados para representar estructuras jerárquicas y campos de tipos de datos, desde buffers individuales reservados por el programa, hasta tipos completos de los objetos en un programa.

4.2. Infiriendo Tipos Primitivos

El tipado de locaciones de programa será realizado usando un algoritmo de inferencia. Dicho algoritmo utilizará diferentes *reglas de inferencia de tipos* aplicadas en diferentes momentos mientras

se analiza una traza de ejecución para tipar locaciones de programa. El tipado dependerá de cual instrucción de ensamblador estemos procesando o si encontramos una llamada o un retorno de una función conocida.

Las reglas de tipado pueden también clasificarse respecto a si dan información de tipo sobre locaciones de programa que se leen o se escriben por la CPU. Las llamaremos *sumidero de tipo de lectura* y *sumidero de tipo de escritura* respectivamente. Ambas reglas de inferencia son formalizas utilizando los conceptos propuestos en este trabajo.

4.2.1. Un Ejemplo Motivador

En este ejemplo vamos a mostrar la intuición detrás de las reglas de inferencia de tipos y los tipos primitivos.

Empezaremos describiendo dos tipos primitivos: $num32$ y $ptr32$ representando respectivamente números de 32-bit y punteros de 32-bit en un conjunto de instrucciones con una semántica similar a la del conjunto de instrucciones de x86. Estos tipos primitivos están relacionados por una relación de refinamiento, debido a que los punteros puede ser interpretados como números. El hecho que los punteros refinan a los números puede ser expresado formalmente como:

$$ptr32 \sqsubseteq num32$$

La relación de refinamiento define una noción fundamental en los tipos primitivos, la compatibilidad: Dos tipos primitivos T_1 y T_2 se dicen compatibles si y sólo si $T_1 \sqsubseteq T_2 \vee T_2 \sqsubseteq T_1$.

Otra importante propiedad de los tipos primitivos es su tamaño. Todo tipo primitivo puede ser asociado a su tamaño en bytes cuando es almacenado en memoria. En este caso, $ptr32$ y $num32$ tiene ambos 4 bytes de tamaño.

Si agregamos dos tipos primitivos especiales, $\top$ y $\bot$ como elementos máximos y mínimos en este relación de refinamiento podemos introducir la idea de tener un látice de tipos primitivos. En este ejemplo en particular, el látice puede verse en la Figura 2a.

El conjunto que contiene todos los posibles tipos primitivos en un látice se conoce como $\mathbb{P}$. Las operaciones básicas de látice, ínfimo y supremo son notadas como $\downarrow$ y $\uparrow$ respectivamente.

Ahora vamos a mostrar como los tipos primitivos son deducidos usando reglas de inferencia de tipos en una secuencia de instrucciones de ensamblador. Supongamos que un programa de x86 ejecuta las siguientes instrucciones de ensamblador:

```
1 incl %ecx
2 xorl %eax, %eax
3 movl %eax, (%ecx)
4 movw %cx , %bx
```

Si no tenemos ninguna información adicional, comenzaremos asumiendo que todas las locaciones de programa sólo contienen $\top$. Luego de ejecutar la linea 1, podemos deducir que ECX contiene un tipo primitivo al menos tan refinado como $num32$. Cuando la CPU ejecuta la linea 2, simplemente no sabemos que tipo primitivo puede ser leído o almacenado en las locaciones de programa de EAX debido a que la instrucción de disjunción exclusive es sólo lógica y puede ser usada en muchos contextos. Entonces, EAX contiene el tipo primitivo $\top$. En la instrucción siguiente, en la linea 3, refinamos el tipo primitivo de ECX como $ptr32$ porque este registro se usó como un puntero.

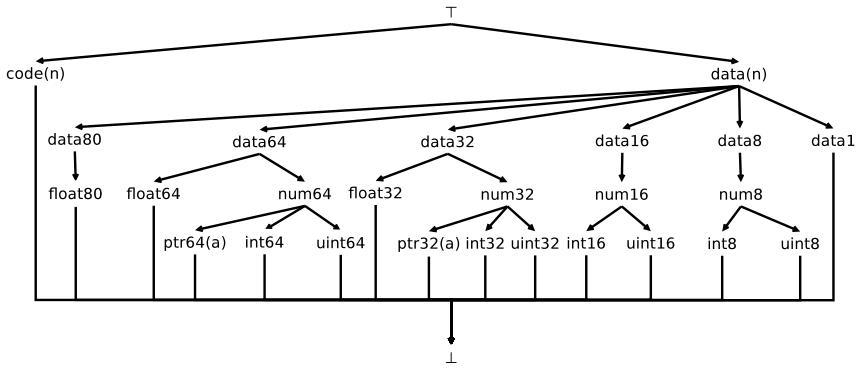


Figura 3: El látice de tipos primitivo completo.

Finalmente, una pregunta interesante surge luego de ejecutarse la última instrucción debido a que se copian los dos bytes más bajos de un puntero al registro BX, por lo tanto, dicho registro no contiene un puntero sino la mitad de él. Podríamos considerar los dos bytes más bajos de un $num32$ como un $num16$, pero no está claro qué deberían ser los dos bytes más bajos de un $ptr32$.

Nuestro formalismo debería ser lo suficientemente flexible como para manejar esta clase de comportamiento. Por lo tanto, introducimos un nuevo látice de tipos primitivos por bytes correspondientes a los tipos primitivos con un índice, para poder llevar cuenta de cada byte de cada tipo primitivo usado, tal como se ve en la Figura 2b. Este índice estará acotado por el tamaño en bytes de cada tipo primitivo. El conjunto de todos los posibles tipos primitivos de bytes se denota $\mathbb{P}_i$. Una relación de refinamiento similar se define entre estos tipos. Adicionalmente, será importante verificar la consistencia en el tipado por bytes porque no toda secuencia de elementos de $\mathbb{P}_i$ garantiza la existencia de un tipo primitivo de $\mathbb{P}$.

Para recuperar estructuras de datos de alto nivel genéricas vamos a usar el látice de tipos primitivos de la Figura 3 usando los tipos inspirados en un subconjunto de los tipos de C.

4.2.2. Reglas de Inferencia de Tipos

El objetivo de esta formalización es deducir el tipo primitivo más refinado para cada locación de programa en diferentes instrucciones de un traza. La deducción del tipo primitivo más refinado requiere definir algunas reglas formales y aplicarlas dependiendo de la instrucción ejecutada.

Algunas definiciones previas son requeridas para saber exactamente sobre qué estamos deduciendo con nuestro formalismo. Debido a que las locaciones de programa son usadas durante la ejecución de un programa para diversos propósitos, una función registrará el tipo primitivo por bytes de cada locación de programas en cada instrucción de la ejecución de un programa. Esta función tomará dos parámetros, el tiempo de ejecución como un número natural y una locación de programa, para retornar un tipo primitivo por bytes. Esta función es llamada $\dot{\tau}$. Formalmente:

$$\dot{\tau} : \mathbb{N} \times \mathbb{L} \to \mathbb{P}_i$$

Usando esta función, podemos definir otra importante función que nos indica que el tipo primitivo de un conjunto de locaciones de programa consecutivas. Esta función también toma un número natural como el parámetro de tiempo y un conjunto de locaciones de programa consecutivas para retornar un tipo primitivo.

$$\tau : \mathbb{N} \times \mathcal{P}(\mathbb{L}) \to \mathbb{P}$$

Definimos esta función porque queremos ser capaces de usar un conjunto de locaciones de programa de cada operador involucrado en una instrucción ejecutada por la CPU. También, esta función realiza una importante verificación de la consistencia de los tipos primitivos por bytes porque éstos deben estar ordenados en las locaciones contiguas. Cuales locaciones de programa son contiguas depende de las locaciones que hayan sido definidas y, por lo tanto, de que arquitectura de computador concreta se está analizando. Estos detalles son abstraídos por τ. Por ejemplo, podemos tener inconsistencias en los índices de un $ptr32$ en EAX en el momento t:

$$\begin{aligned} &\dot{\tau}_t(R(eax,0)) = ptr32(0) \quad \wedge \\ &\dot{\tau}_t(R(eax,1)) = ptr32(1) \quad \wedge \\ &\dot{\tau}_t(R(eax,2)) = ptr32(3) \quad \wedge \\ &\dot{\tau}_t(R(eax,3)) = ptr32(2) \quad \nRightarrow \quad \tau_t(L_{EAX}) = ptr32 \end{aligned}$$

Las reglas formales de inferencia de tipos pueden ser clasificadas de acuerdo a la clase de instrucciones sobre la cual ellas operan.

4.2.3. Movimiento de Memoria.

Una de las operaciones más importantes definidas en casi cualquier arquitectura de computador es la instrucción que simplemente mueve bytes de memoria de un operador fuente (src) a un operador de destino (dst). En x86 existen varias instrucciones que involucran un movimiento directo de memoria (por ejemplo mov, movs, push, pop). Para obtener un formalismo simple y unificado, formalizamos esta operación definiendo el movimiento de memoria desde src a dst simplemente moviendo byte a byte con la siguiente notación:

$$dst \leftarrow src \quad = \quad \forall i \in \{0, \ldots, size(dst) - 1\} \bullet dst_i \leftarrow src_i$$

Con esta definición equivalente de movimiento de memoria podemos hacer que los tipos primitivos por bytes se preserven entre las locaciones de programa fuente y destino.

$$\forall i \in \{0, \ldots, size(dst) - 1\} / l_{src} \in L_{dst_i}, l_{dst} \in L_{dst_i}$$

$$\dot{\tau}^{t+1}(l_{dst}) = \dot{\tau}^{t}(l_{src}) \tag{1}$$

4.2.4. Reuso de Memoria

Algunas locaciones de programa son comúnmente utilizadas para almacenar valores temporarios. Los ejemplos típicos son los registros del procesador. Cada vez que se escriben bytes en un registro, la información de los tipos primitivos anterior es inútil porque los registros pueden contener cualquier clase de valor. Sin embargo, otras locaciones deberían mantener tipos consistentes durante un lapso de tiempo. Dichas locaciones de programa son aquellas que contienen objetos. Una suposición clave que hacemos en este formalismo es:

Todo objeto se almacena en un sólo buffer (2)

dónde un buffer es una región reservada de memoria continua.

Estamos particularmente interesados en las locaciones de programa que representan la memoria que contienen estos objetos. Por lo tanto, en estas locaciones de buffer, los tipos primitivos deberían refinarse o preservarse a través de las sucesivas escrituras. Formalizamos esta posibilidad con un pequeño cambio en la regla de movimiento de memoria descripta en la Fórmula 1 cuando el operador *dst* corresponde a las locaciones que preservan memoria.

$$\forall i \in \{0, \ldots, size(dst) - 1\} / l_{src} \in L_{dst_i}, l_{dst} \in L_{dst_i}$$

$$\dot{\tau}^{t+1}(l_{dst}) = \dot{\tau}^{t}(l_{src}) \downarrow \dot{\tau}^{t}(l_{dst}) \tag{3}$$

4.2.5. Las reglas de inferencia de x86

Como establecimos anteriormente, las reglas de inferencia de tipos pueden dividirse en dos clases:

- Reglas de inferencia de tipos asociadas a una instrucción de ensamblador:

  Esta clase de reglas de inferencia de tipos están asociadas con una instrucción de ensamblador en particular. Con ellas, tipos primitivos acotan los operadores de dichas instrucciones.

  Ejemplos de estas reglas de inferencia de tipos incluyen las instrucciones aritméticas (de propósito general, con signo y sin signo), las instrucciones que manipulan números flotantes y las que manipulan punteros entre otras.

- Reglas de inferencia de tipos asociadas a una llamada o un retorno de una función:

  Esta clase de reglas de inferencia de tipos están asociados con una función conocida detectada durante una ejecución del programa. Con ellas, tipos primitivos acotan los argumentos y los valores de retorno. Por supuesto, las locaciones que se les infieren el tipo depende directamente de la convención de llamada a función que utiliza el programa. Ejemplos de estas reglas incluyen las funciones de reservado de memoria provistas por el sistema operativo o las funciones de manipulación de cadenas estándar entre otras.

En el presente trabajo, el uso de estas reglas de inferencia no requiere conocimiento previo del tipo de otras locaciones de programa durante la ejecución. Llamamos a esta característica como *sumidero de tipo no condicional*. Usando condiciones en los sumideros de tipo, podríamos describir un sistema completo de inferencia de aritmética de punteros. Por ejemplo, una de estas reglas de tipado de punteros podría describir la suma realizada con un operador tipado como $ptr32$ y otro tipado como $num32$ resultando un $ptr32$.

Operaciones aritméticas. Las operaciones aritméticas son ampliamente utilizadas por las instrucciones de ensamblador. Estas instrucciones se utilizan para aritmética de enteros y punteros. Por ejemplo, adiciones y substracciones a través de las instrucciones *add* y *sub* en x86 realizan tanto operaciones sobre enteros con signo como sin signo y punteros. Por lo tanto, inferimos el tipo primitivo de los operadores usando el tamaño de los operadores que manipulan estas instrucciones como $num32$, $num16$ o $num8$. Para la multiplicación, *imul* opera sobre enteros con signo, por eso

sus operadores son tipados como $int32$ y mul opera sobre los enteros sin signo y sus operadores son tipados como $uint32$. La Tabla 1 resume las operaciones aritméticas de 32-bit.

Instrucción abstracta	Inferencia
$dst \overset{t}{\leftarrow} addl(src_1, src_2)$	$num32 \sqsubseteq \tau^t(L_{src_1})$ $num32 \sqsubseteq \tau^t(L_{src_2})$ $num32 \sqsubseteq \tau^{t+1}(L_{dst})$
$dst \overset{t}{\leftarrow} subl(src_1, src_2)$	$num32 \sqsubseteq \tau^t(L_{src_1})$ $num32 \sqsubseteq \tau^t(L_{src_2})$ $num32 \sqsubseteq \tau^{t+1}(L_{dst})$
$dst \overset{t}{\leftarrow} imul(src_1, src_2)$	$int32 \sqsubseteq \tau^t(L_{src_1})$ $int32 \sqsubseteq \tau^t(L_{src_2})$ $int32 \sqsubseteq \tau^{t+1}(L_{dst})$
$dst \overset{t}{\leftarrow} mul(src_1, src_2)$	$uint32 \sqsubseteq \tau^t(L_{src_1})$ $uint32 \sqsubseteq \tau^t(L_{src_2})$ $uint32 \sqsubseteq \tau^{t+1}(L_{dst})$

Tabla 1: Reglas de inferencia para algunas operaciones aritméticas de 32-bit.

Indirección. El uso de números como direcciones de memoria a través de la indirección es una muy importante operación en la arquitectura x86. Si una instrucción utiliza un operador como un puntero, las locaciones de este operador son tipadas como puntero. Otro sumidero de tipos es definido usando el valor de retorno de las funciones que reservan memoria, capturadas en el registro de reserva de memoria. Podemos deducir que las locaciones que contienen el valor de una nueva reserva de memoria contienen un puntero. También debemos recuperar dónde los punteros apuntan: cada vez que un puntero es detectado usando alguno de estos sumideros de tipo, será necesario buscar dónde apunta el valor de este puntero en la memoria y registrar si apunta a algún buffer. Esta información será fundamental en el proceso de construcción de un grafo de tipos preciso, tal como lo explicaremos en el próximo capítulo.

Instrucción abstracta	Inferencia
$dst \overset{t}{\leftarrow} *(src)$	$ptr32 \sqsubseteq \tau^t(L_{src})$
$ret(alloc)$	$ptr32 \sqsubseteq \tau^t(L_{eax})$

Tabla 2: Reglas de inferencia para punteros de 32-bit.

Los tipos de datos inferidos por nuestros algoritmos para instrucciones que manipulan operadores de 32-bit y 64-bit se muestran en la Tabla 3. Los sumideros de tipos de punteros se muestran en la Tabla 2 como también los sumideros de tipos de retorno de las funciones de reserva de memoria, que son los únicos sumideros asociados con una función que se utilizan.

4.3. Algoritmo de Inferencia de Tipos

En la segunda fase del proceso de recuperación de tipos, el módulo de *seguimiento de tipos* utiliza las reglas de inferencia para recuperar tipos de las variables en memoria y los registros en

cada instrucción de una traza. Esta fase se repite varias veces para múltiples trazas. Los tipos primitivos resultantes, cuando son situados en los buffers representan campos de los objetos que contienen.

El seguimiento de tipos utiliza el mismo enfoque dinámico de análisis de flujo de datos empleado por REWARDS [23], pero con diferencias significativas que incluyen un látice bien definido, distintas opciones de unificación restringida o sin unificación y punteros como tipos parametrizados por otro tipo. Además nos inspiramos de TIE [21] (por ejemplo, para nuestro látice de tipos primitivos) pero preferimos un enfoque dinámico porque resolver un sistema de restricciones cada vez que se libera un buffer es demasiado costoso computacionalmente. Adicionalmente, recuperar el tipo al que apuntan los punteros es más sencillo usando técnicas de análisis dinámico.

Nuestros algoritmos de inferencia de tipos puede ser utilizados en paralelo con la ejecución del programa como un algoritmo de tipado en linea, ya que no requiere ningún tipo backtracking.

4.3.1. Los algoritmos TIPO

Una contribución clave presentada en este trabajo es un algoritmo de tipado de locaciones de programa. La principal diferencia entre nuestros algoritmos y REWARDS es el manejo de la unificación. La unificación sucede en las instrucciones de movimiento de memoria. En estas instrucciones $dst \leftarrow src$, y dst hereda los tipos de src. Con unificación, src también hereda los tipos que dst tenía antes de ser sobrescrito. La unificación resulta problemática cuando un conjunto de locaciones se utiliza para almacenar variables de distinto tipo. Por esta razón, REWARDS inhabilita la unificación cuando se escribe en un registro, ya que los registros se utilizan para almacenar temporariamente variables de distinto tipo. Sin embargo, la memoria de stack también se utiliza para almacenar y se convierte en una fuente de conflictos de tipos cuando se utiliza REWARDS. Nuestros tres algoritmos desactivan o restringen la unificación para evitar estos problemas. Comparados con REWARDS, nuestros algoritmos recuperan más tipos en locaciones de programa y tienen menos conflictos.

Nuestros algoritmos se agrupan bajo el nombre, en inglés, *Type Inferences using Primitive Operations*, o por las siglas *TIPO* y se denominan TIPO conservador (TIPO-C), TIPO agresivo (TIPO-A) y TIPO unificación limitada (TIPO-LU).

Nuestros algoritmos funcionan procesando cada instrucción secuencialmente en un traza de ejecución. Éstos ejecutan una función llamada *instrument_ insn* con una instrucción de ensamblador

Tipo Primitivo	**Instrucciones**
$Num32$	add, sub, inc, dec, lea
$UInt32$	mul, not
$Int32$	imul, neg, fild
$Ptr32(Code)$	ijmp, icall
$Float64$	fst, fstp, fld, fadd fsub, fmul, fdiv
$\top$	*Todas las demás*

Tabla 3: El tipo primitivo inferido por los sumideros de tipos de TIPO por instrucciones que manejan operadores 32-bit y 64-bit.

insn como parámetro. Para cada instrucción procesada, los algoritmos TIPO realizan cuatro pasos fundamentales:

1. La deducción de los tipos de las locaciones de programa usadas por la instrucción usando los sumideros de tipos definidos en 4.2.2. Los algoritmos refinan el tipo de las locaciones de programa leídas por la instrucción usando el ínfimo entre el tipo actual y el nuevo de acuerdo al látice. Y sobrescriben el tipo de las locaciones de programas escritas por la instrucción con el tipo deducido por los sumideros de tipo (por ejemplo, las líneas 3–5 en el Algoritmo 1)

2. El manejo de instrucciones de movimiento de memoria. Nuestros algoritmos difieren solamente en este paso.

3. Para instrucciones que no mueven memoria, a todas las locaciones de dicha instrucción analizadas que no fueron tipadas con un sumidero de tipos, se asigna su tipo a $\top$ (por ejemplo, las líneas 11–13 en el Algoritmo 1).

4. El manejo de bytes reservados o liberados asignando $\top$ a los tipos primitivos de esas locaciones de programa:

   a) Cuando la instrucción es una llamada o un retorno de una función de reserva de memoria, la información de tipos cada byte contenido en el nuevo buffer reservado se asigna a $\top$ (por ejemplo, líneas 15–18 en el Algoritmo 1)

   b) Cuando la instrucción es una llamada a una función para liberar memoria, se crea el árbol de rango con la información de tipado del buffer, para el rango memoria liberada. Luego, la información de tipos de cada byte contenido en el buffer liberado se asigna a $\top$ (por ejemplo, líneas 20–24 en el Algoritmo 1).

Agrupación de locaciones de programa. Cuando movemos datos en un programa y sobrescribimos o unificamos locaciones, podemos crear grupos de locaciones que comparten el mismo tipo primitivo. Dichos grupos definen una partición sobre $\mathbb{L}$ y nos permiten:

- Si el tipo de una locación de programa debe ser refinado, actualizar los tipos primitivos correspondientes a todas las locaciones en su grupo.

- Si se mueve memoria, combinar los grupos de locaciones correspondientes a las locaciones de programa de fuente y de destino.

Una función especial, llamada γ nos indica que locaciones de programa comparten el mismo tipo primitivo. Por ende, $\gamma(l)$ retorna el conjunto de locaciones compartiendo el mismo tipo primitivo de la locación l. Si no poseemos información adicional, comenzaremos asumiendo que cada locación de programa tiene un tipo primitivo distinto. Esto puede formalizarse de la siguiente manera: $\forall l \in \mathbb{L} \bullet \gamma(l) = \{l\}$.

Adicionalmente, dos operaciones son necesarias para manipular los grupos de locaciones que comparten el mismo tipo: una operación para combinar grupos de locaciones que comparten el mismo tipo llamada *merge_ groups* y una operación para reemplazar directamente el grupo de locaciones que comparten el mismo tipo de una locación específica llamada *replace_ group*.

Algoritmo 1 TIPO-C

```
1  def instrument_insn(insn) {
2    for (l_src,T) in read_type_sinks(insn)
3      refine(l_src,T)
4    for (l_dst,T) in write_type_sinks(insn)
5      overwrite(l_dst,T)
6
7    if (moves_data(insn))
8      for (l_src,l_dst) in read_write_locations(insn)
9        overwrite(l_dst τ̇(l_src))
10   else
11     for l_dst      in write_locations(insn) and
12         (l_dst,_) not in write_type_sinks(insn)
13       overwrite(l_dst,⊤)
14
15   if is_alloc_call(insn) or is_alloc_ret(insn)
16     (addr,size) = get_alloc_range(insn)
17     for a in [addr,addr+size)
18       overwrite(memloc(a),⊤)
19
20   if is_dealloc_call(insn)
21     (addr,size) = get_dealloc_range(insn)
22     create_buffer_tree(addr,size)
23     for a in [addr,addr+size)
24       overwrite(memloc(a),⊤)
25 }
26
27 def refine(l,T) {
28   τ̇(l) ← τ̇(l) ↓ T
29 }
30
31 def overwrite(l,T) {
32   τ̇(l) ← T
33 }
```

TIPO-C. La versión conservadora de TIPO (Algoritmo 1) nunca utiliza unificación ni crea grupos de locaciones que comparten el mismo tipo, por eso, en las instrucciones de movimientos de datos *dst* hereda la información de tipo de *src*, pero *src* no hereda ninguna tipo de *dst*. TIPO-C maneja el movimiento de datos sobrescribiendo la información de tipos de las locaciones de *dst* con los tipos de las locaciones de *src* siguiendo la Fórmula 1.

TIPO-C utiliza *refine* y *overwrite* para actualizar los tipos primitivos de las locaciones de programa en $\dot{\tau}$. Por simplicidad, $\dot{\tau}$ no estará parametrizado por el tiempo, sino que se actualizará la función cada vez que una instrucción se procesa y se detecta nueva información de tipo para locaciones de programa.

TIPO-LU. TIPO-LU (Algoritmo 2) trata las instrucciones de manejo de memoria ($dst \leftarrow src$), unificando *src* con *dst* sólo si *dst* es heap debido a que las locaciones de memoria en la heap preservan su tipo en sucesivas escrituras. Si *dst* son locaciones de stack o de registros, el algoritmo sobrescribe los tipos debido a que la memoria de stack y los registros son frecuentemente utilizados como almacenamiento temporario. Definimos que locaciones de memoria preservan sus tipos cuando son escritas con la función booleana *preserve_ type*. La unificación se realiza en las líneas 11–14, refinando *src* y *dst* y combinando los grupos correspondientes. Si *dst* almacena solamente información de manera temporaria, sus tipos y locaciones agrupadas debe ser sobrescritas con las de *src* y las locaciones de *dst* deben ser agregadas al grupo de locaciones compartiendo el mismo tipo que *src*. Esto se realiza en las líneas 16–17.

TIPO-LU también utiliza *refine* y *overwrite* para actualizar $\dot{\tau}$, pero estas funciones además manipulan las locaciones agrupadas: cuando un tipo primitivo de una locación de programa debe ser refinado, *refine* actualiza todas las locaciones que comparten ese mismo tipo. Además, cuando el tipo de una locación debe ser sobrescrito, *overwrite* también reinicia el grupo de locaciones que comparte el tipo con esa locación.

TIPO-A. La versión agresiva de TIPO (Algoritmo 3) funciona agrupando locaciones de programa y actualizando sus tipos usando *refine* y *overwrite* como TIPO-LU pero sin realizar unificación. TIPO-A se encarga de las operaciones de movimiento de memoria sobrescribiendo los tipos de *dst* y las locaciones de memoria agrupadas con las de *src* y agregando las locaciones de *dst* al grupo de locaciones compartiendo el mismo tipo que *src*. Esto se realiza en las líneas 10–11.

TIPO-A trabaja de una manera muy similar a TIPO-LU, pero considera que ninguna locación de programa preserva el tipo en las sucesivas escrituras. Sería como si la función preserve_type siempre retornara falso y la unificación nunca se llevara a cabo. Sorprendentemente, los resultados muestran que TIPO-A y TIPO-LU tienen una precisión similar detectando tipos primitivos. Sin embargo, existe una importante diferencia entre ellos: la implementación de TIPO-A es considerablemente más simple y rápida.

4.4. Creando Árboles de Buffers

Los objetos son elementos centrales de nuestro análisis. Por esta razón, usamos árboles de rango para hacer un seguimiento de la jerarquía y la disposición de los campos durante la ejecución. Debido a esto, cada vez que un nuevo buffer se reserva, un árbol de rango es creado. Particularmente, estos árboles de rango reciben el nombre de *árboles de buffer*.

Algoritmo 2 TIPO-LU

```
1  def instrument_insn(insn) {
2
3    for (l_src,T) in read_type_sinks(insn)
4      refine(l_src,T)
5    for (l_dst,T) in write_type_sinks(insn)
6      overwrite(l_dst,T)
7
8    if (moves_data(insn))
9      for (l_src,l_dst) in read_write_locations(insn)
10       if (preserve_types(l_dst))
11         inf = τ̇(l_src) ↓ τ̇(l_dst)
12         refine(l_src, inf)
13         refine(l_dst, inf)
14         merge_groups(l_src,l_dst)
15       else
16         overwrite(l_dst, τ̇(l_src))
17         merge_groups(l_src,l_dst)
18   else
19     for l_dst       in write_locations(insn) and
20       (l_dst,_) not in write_type_sinks(insn)
21       overwrite(l_dst, ⊤)
22
23   if is_alloc_call(insn) or is_alloc_ret(insn)
24     (addr,size) = get_alloc_range(insn)
25     for a in [addr,addr+size)
26       overwrite(memloc(a),⊤)
27
28   if is_dealloc_call(insn)
29     (addr,size) = get_dealloc_range(insn)
30     create_buffer_tree(addr,size)
31     for a in [addr,addr+size)
32       overwrite(memloc(a),⊤)}
33
34 def refine(l,T) {
35   for l_i in γ(l):
36     τ̇(l_i) ← τ̇(l_i) ↓ T
37 }
38
39 def overwrite(l,T) {
40   τ̇(l) ← T
41   replace_group(l,{l})
42 }
```

Algoritmo 3 TIPO-A

```
1  def instrument_insn(insn) {
2
3    for (l_src,T) in read_type_sinks(insn)
4      refine(l_src,T)
5    for (l_dst,T) in write_type_sinks(insn)
6      overwrite(l_dst,T)
7
8    if (moves_data(insn))
9      for (l_src,l_dst) in read_write_locations(insn)
10       overwrite(l_dst, τ̇(l_src))
11       merge_groups(l_src,l_dst)
12   else
13     for l_dst       in write_locations(insn) and
14         (l_dst,_) not in write_type_sinks(insn)
15       overwrite(l_dst, ⊤)
16
17   if is_alloc_call(insn) or is_alloc_ret(insn)
18     (addr,size) = get_alloc_range(insn)
19     for a in [addr,addr+size)
20       overwrite(memloc(a),⊤)
21
22   if is_dealloc_call(insn)
23     (addr,size) = get_dealloc_range(insn)
24     create_buffer_tree(addr,size)
25     for a in [addr,addr+size)
26       overwrite(memloc(a),⊤)}
27
28 def refine(l,T) {
29   for l_i in γ(l):
30     τ̇(l_i) ← τ̇(l_i) ↓ T
31 }
32
33 def overwrite(l,T) {
34   τ̇(l) ← T
35   replace_group(l,{l})
36 }
```

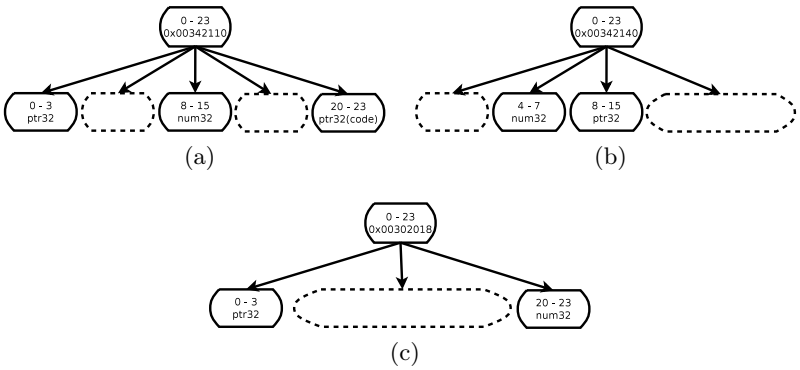


Figura 4: Un ejemplo de tres árboles de buffers inferidos de una estructura sqlite3_rtree_geometry.

4.4.1. Ejemplo Motivador

Vamos a explicar e ilustrar los árboles de buffers usando un ejemplo tomado directamente del código fuente de Sqlite [30]. La Figura 5a muestra una estructura de C que se encuentra en el código fuente llamada *sqlite3_rtree_geometry*. Esta estructura contiene la información necesaria para definir un r-tree [18]. Un árbol de buffer con cada campo en esta estructura se muestra en la Figura 5b. Esta es sola una de muchas posibilidades, ya que la disposición exacta es determinada por el compilador y puede variar.

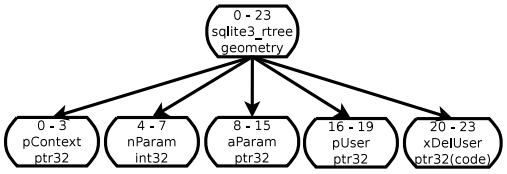
Ahora, vamos a enfocarnos visualizar un árbol de buffer etiquetado con la información de tipos inferida. Supongamos que durante la traza de ejecución de sqlite se reservan tres buffers para crear una estructura sqlite3_rtree_geometry en cada uno. También supongamos que nuestro módulo de seguimiento de tipos, al final de la traza detecta algunos campos en cada buffer como se muestran en la Figura 4. Como podemos ver, los buffers contienen sólo algunos de los campos originales. Los campos faltantes en estos buffers no fueron usados (leídos o escritos) durante la ejecución. Además, los campos inferidos podrían no ser tan refinados como en el árbol de rango de referencia creado con el código fuente. Por ejemplo, en la Figura 4a, el campo $aParam$ es inferido como $num32$ y no como $ptr32$.

4.4.2. Extrayendo las Estructuras de los Buffers

Luego de inferir los tipos primitivos de cada buffer en una traza de ejecución, reconstruimos la estructura del buffer. Cada vez que un buffer se libera, un *árbol de buffer* se crea. Dicho árbol contendrá los campos creados usando los tipos primitivos inferidos con sus tamaños como la longitud de cada campo.

```
struct sqlite3_rtree_geometry {
  void    *pContext;
  int     nParam;
  double  *aParam;
  void    *pUser;
  void    (*xDelUser)(void *);
};
```

(a) Estructura de C.



(b) Árbol de buffer.

Figura 5: Una estructura llamada sqlite3_rtree_geometry hallada en el código fuente de Sqlite y su correspondiente árbol de buffer.

4.4.3. Resultados del Análisis de Ejecución

Árboles de Buffers El resultado primario de esta fase es una lista de árboles de buffers, uno por cada buffer reservado por el programa, incluyendo los liberados. Estos árboles de buffers contienen los campos recuperados usando el módulo de seguimiento de tipos.

Grafo Concreto de Heap Usando los buffers que ser reservaron durante la ejecución de un programa y la información de punteros, el módulo constructor del grafo de heap puede generar el grafo de heap en cualquier punto de la ejecución de un programa.

5. Análisis Global

La tercera fase de la recuperación de tipos usa los árboles de buffer para tratar de deducir que buffers tienen el mismo tipo. Un procedimiento para combinar la información en los árboles de buffers será definido de manera que sea posible obtener una aproximación del grafo de tipos usado en el programa analizado. Esta fase combina información de árboles de buffers de múltiples trazas del mismo programa. Adicionalmente, arreglos dinámicos pueden ser identificados en ciertos callsites. Primero introduciremos algunos conceptos requeridos para explicar esta fase.

5.1. Información de Callsites

La dirección de una instrucción que invoca la función de reserva de memoria de un buffer se denomina callsite. Todo buffer tiene su callsite debido a que debió ser reservado en algún momento de la ejecución del programa. El callsite da una idea sobre el tipo del objeto en el buffer porque suponemos que no se cambiará de tipo (casting) a un objeto en un buffer a lo largo de su ejecución. Desafortunadamente, no podemos usar la información provista por el callsite como un reemplazo preciso del tipo porque algunos buffers con diferentes callsites podrían contener objetos del mismo tipo.

Tampoco podemos olvidar que los callsites en una traza de ejecución dependen fuertemente de como el compilador fue utilizado en la creación de ese programa binario. Una técnica muy común de optimización conocida como inlining puede producir callsites diferentes cuando una función reserva un buffer: si esta función se compila inline, cada llamada a la misma será traducida en un conjunto distinto de instrucciones y por lo tanto, un callsite distinto. De todas maneras, estos callsites distintos corresponderían a objetos del mismo tipo.

Los callsites y los tamaños de buffer también están relacionados. Los buffers con el mismo tipo deberían tener el mismo tamaño. Por ende, si los callsites reservan buffers del mismo tipo, estos buffers deben ser del mismo tamaño. Un callsite con solamente buffers del mismo tamaño se conoce como *callsite estable*. Si dos buffers de distinto tamaño tiene el mismo callsite, dicho callsite se dice *inestable*. Más adelante en esta sección, presentaremos un método para detectar arreglos dinámicos en *callsite inestables*.

5.1.1. Ejemplo Motivador

En el código fuente de sqlite, la función deserializeGeometry puede reservar buffer que contienen estructuras de sqlite3_rtree_geometry:

```
static int deserializeGeometry
 (sqlite3_value *pValue, RtreeConstraint *pCons){
  RtreeMatchArg *p;
  sqlite3_rtree_geometry *pGeom;
  int nBlob;

  ...

  pGeom = (sqlite3_rtree_geometry *)malloc(
      sizeof(sqlite3_rtree_geometry) + nBlob
  );
  if( !pGeom ) return SQLITE_NOMEM;

  ...

  return SQLITE_OK;
}
```

En esta función, todos los buffers reservados tienen el mismo callsite. A nivel de binario, el compilador genera código ensamblador que invoca a malloc en $0x00421230$ y los correspondientes árboles de buffer son reservados usando deserializeGeometry. Los buffers representados en las Figuras 4a, 4b y 4c tendrá este callsite.

5.2. Detectando Arreglos Dinámicos

El sencillo proceso de detección de tipos puede ser ampliado con varias técnicas para detectar otros tipos datos comúnmente usados. La detección de arreglos dinámicos es una de estas técnicas. El procedimiento para detectar arreglos dinámicos en nuestro formalismo puede resumirse así:

1. Se identifican los buffers reservado en el mismo callsite pero con diferente tamaño.
2. Se busca el máximo común divisor entre los tamaños en bytes de los buffers.
3. Se segmentan los buffers en subbuffers del tamaño del máximo común divisor.
4. Se crean los árboles de buffer por cada subbuffers, verificando compatibilidad entre ellos y combinando su información.

Si alguno de estos pasos falla para algún callsite, simplemente se lo deja como inestable. El resultado de este procedimiento es un árbol de rango con su raíz de tipo arreglo dinámico y con un único elemento igual al tipo del arreglo dinámico detectado. Una desventaja de esta técnica es que requiere al menos dos buffers de distinto tamaño, de otra manera, el callsite no será considerado como un posible arreglo dinámico. Este problema se mitiga combinando la información generada por múltiples trazas del mismo programa.

5.3. Combinando Árboles de Buffers

En cada callsite estable, suponemos que los árboles de buffer representan objetos del mismo tipo. Naturalmente, nos gustaría poder combinar esta información para obtener información de tipo más

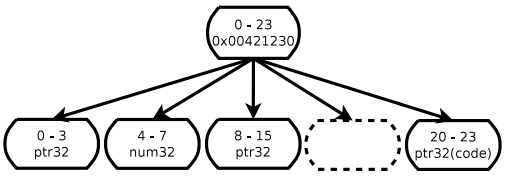


Figura 6: El árbol de callsite luego de realizar la combinación.

completa. Por esta razón, necesitamos un procedimiento para reconstruir la información de tipos a partir de la información de tipos de cada árbol de buffer. Dicho procedimiento debe ser consistente y seguro.

5.3.1. Un Procedimiento para Combinar Árboles

Se definirá un proceso para combinar la información de los árboles de rango. No todos los árboles de rango pueden ser combinados. Si el procedimiento falla, decimos que los árboles que se intentar combinar son incompatibles y no pueden ser combinados. El proceso de combinación requiere insertar todos los nodos de un árbol de rango en el otro. Si el rango de un nodo para ser insertado ya existe en el otro árbol de rango, se debería verificar las compatibilidad de acuerdo a la siguiente tabla.

tipo de nodo	compatible con	nodo resultante
estructura	estructura	estructura
primitivo(T_1)	primitivo(T_2)	primitivo($T_1 \downarrow T_2$)
arreglo(T_1)	arreglo(T_2)	arreglo($T_1 \downarrow T_2$)
arreglo_dinámico(T_1)	arreglo_dinámico(T_2)	arreglo_dinámico($T_1 \downarrow T_2$)

Si existe un solapamiento de nodos a insertar, el procedimiento falla. Adicionalmente, en el árbol de rango resultante, todos los nodos primitivos deben ser hojas y sólo los nodos que son arreglos dinámicos sólo pueden ser la raíz, o el procedimiento fallará.

5.3.2. Combinando por Callsite

Si un callsite es estable, podemos combinar la información de sus árboles de buffers. El árbol de rango resultante será llamado *árbol de callsite*. En nuestro ejemplo, luego de combinar los árboles de buffer del callsite $0x00421230$, podemos observar el árbol de callsite resultante en la Figura 6.

5.4. Combinando Árboles de Tipo

Comenzamos combinando la información de los árboles de buffers usando sus callsites, pero podemos ir un paso más allá si usamos la información provistas por los punteros inferidos en los buffers. El resultado de esta combinación de árboles de callsite es otra clase de árboles de rango, con la información provista por los callsite. Estos árboles son llamados *árboles de tipo*.

5.4.1. Combinando por Punteros

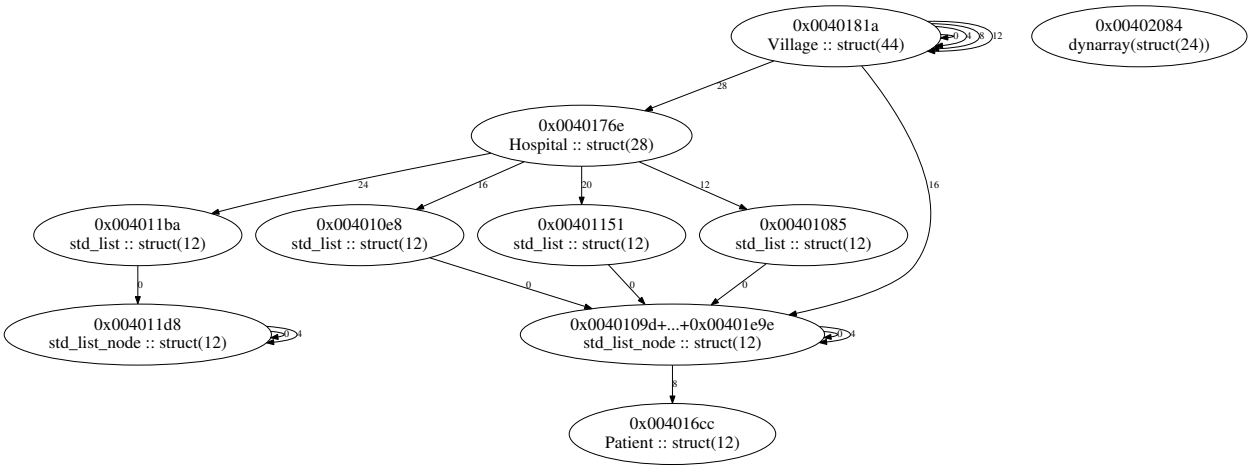
Un interesante enfoque para combinar los árboles de callsite correspondiente al mismo tipo se deriva de la siguiente suposición:

Los campos de punteros en un buffers del mismo tipo apuntan siempre al mismo tipo (4)

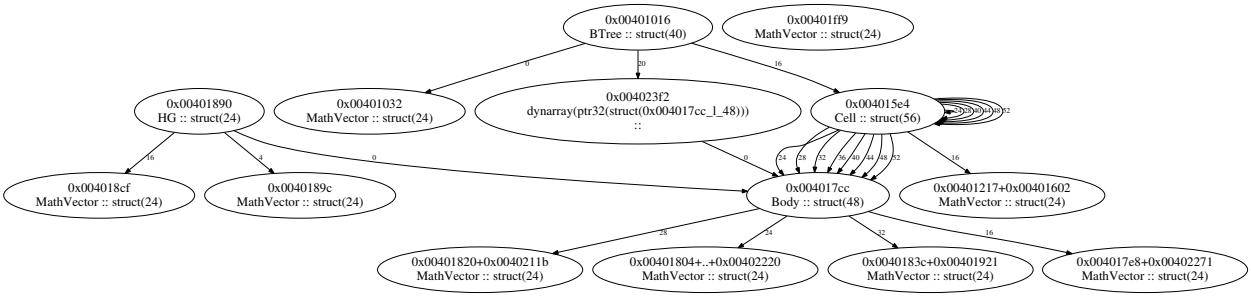
De este principio, un procedimiento para combinar árboles de callsite comienza considerando a cada árbol de callsite como un árbol de tipo distinto. Si x e y son punteros detectados en el mismo offset en árboles de buffer en una clase de callsites asociadas a un árbol de tipos, x apunta al tipo T_x e y apunta al tipo T_y, entonces T_x y T_y son de hecho, el mismo tipo, y deben ser combinados en un mismo árbol de tipos. Este procedimiento continua hasta que no es posible encontrar punteros x e y como los descriptos anteriormente. En ciertas circunstancias, la combinación de dos árboles de tipos detectados como pertenecientes al mismo tipo con este procedimiento puede fallar. Un caso interesante de estas situaciones se da si en un programa existen diversos subtipos que heredan de un tipo de dato particular y un puntero que apunta a distintos subtipos. La combinación por el tipo del puntero puede fallar porque no hay garantías de que los distintos subtipos sean compatibles entre sí. No hemos explorado todavía la posibilidad de detectar herencia de tipos en el grafo de tipos reconstruido, pero esta situación parece un punto de partida prometedor para investigaciones futuras al respecto.

5.4.2. Resultado del Análisis Global

El resultado primario de esta fase es un grafo de tipos creados usando la información de los árboles de callsites. Este grafo tiene conjuntos de callsites como nodos y usa los punteros inferidos en los buffers para determinar como los tipos de datos se apuntan entre sí. A pesar de que se obtiene un grafo de tipos, este no contendrá nombres para los tipos de datos en el programa, sino conjuntos de callsite. Adicionalmente, algunos nodos pueden ser marcados como posibles arreglos dinámicos. Los grafos de tipos de dos programas evaluados en este trabajo se muestran en la Figura 7. Estos grafos se obtuvieron usando TIPO con unificación limitada. Los grafos reconstruidos no tienen nombres para los tipos. Agregamos sus nombres usando la información de depuración para mostrar la precisión del grafo de tipos reconstruido.



(a) HealthBmk



(b) BHBmk

Figura 7: Grafos de tipos reconstruidos de los dos programas de ejemplo usados en la evaluación.

6. Evaluación

Para la evaluación del formalismo presentado en este trabajo, se realizarán experimentos basados en el análisis de dos programa de ejemplo: BHBmk y HealthBmk, y un complejo mensajero instantáneo de código abierto, llamado Pidgin [24].

Estas pruebas se realizaron debido a diferentes razones. Por un lado, el uso de pequeños programas de ejemplo nos permite determinar que tan preciso puede ser el grafo de tipos reconstruido y detectar claramente sus limitaciones. Por otro lado, el uso de programas como Pidgin nos permite evaluar la practicidad del formalismo cuando se usa para encontrar estructuras de datos específicas en un programa con gran número de tipos de datos.

El primero de los programas de ejemplo, llamado BHBmk es una implementación del algoritmo de Barnes-Hut N-Cuerpos [5] y el segundo, llamado HealthBmk es una implementación del benchmark de Olden simulando el sistema de salud colombino [16].

Estos programas fueron escritos en C++ y sus binarios fueron compilados para un entorno Win32 usando Visual Studio 2011 en modo release con las optimizaciones por defecto. Debido a que tenemos acceso al código fuente de BHBmk y HealthBmk, podemos extraer la información de depuración de un archivo de base de datos de programa, o PDB por sus siglas en inglés. Por supuesto, esta información no será utilizada durante el proceso de inferencia de tipos definido en el presente trabajo, pero necesitamos dicha información para evaluar nuestros resultados. La información de depuración nos dirá exactamente la disposición de los campos de cada buffers y la información de tipo definida en el código fuente correspondiente. Estos dos programas de ejemplo contienen una rica estructura de clases con diversos tipos primitivos en sus campos.

En HealthBmk, los objetos representando a los pueblos (Village), hospitales (Hospital) y pacientes (Patient) son creados. Los Villages contienen listas de pacientes. Un árbol con cuatro ramas, también llamado quadtree, con Village en sus nodos se implementa usando un arreglo de cuatro punteros a Village. Cada Village contiene un puntero a un Hospital, con cuatro listas específicas usadas para simular el sistema de salud colombino. El resultado se almacena en objetos de tipo Result.

En BHBmk, los objetos representando árboles de Barnes-Hut (BTree), sus nodos (Cell), los cuerpos en la simulación (Body) y vectores tridimensionales (MathVector) son las estructuras de datos principales usadas en esta implementación del algoritmo de N-Cuerpos de Barnes-Hut.

Los objetos de tipo Node representan los campos comunes para implementar nodos y cuerpos. De hecho, Cell y Body heredan de Node. Finalmente, el tipo HG contiene la información necesaria para realizar la computación de la gravedad.

En la Figura 8 se pueden ver los grafos de tipos de referencia de HealthBmk y BHBmk usando la información extraída directamente de los archivo PDB. Adicionalmente, estos ejemplos usan la biblioteca estándar de C++ de Microsoft y por eso las estructuras internas de los vectores y listas de dicha librería son visibles en el grafo.

Implementación. La implementación de los algoritmos TIPO para su evaluación se realizó dentro del formalismo de análisis binario llamado BitBlaze [7]. Dicho formalismo se presenta en forma de librería de Ocaml e implementa herramientas básicas de análisis binario, tanto para la captura de información de bajo nivel de programas como para el análisis simbólico de dicha información. Tipógrafo utiliza la mayor parte de las interfaces provistas por BitBlaze en Ocaml para implementar distintas variantes de nuestros algoritmos. Adicionalmente, REWARDS fue reimplementado junto a

nuestras herramientas utilizando el mismo formalismo. Esto fue posible gracias al acceso al código fuente, amablemente provisto por sus creadores.

Actualmente, nuestra implementación no forma parte oficial de BitBlaze, pero esperamos que el código sea revisado e incorporado pronto.

6.1. Inferencia de Tipo de Objetos

Vamos a comparar nuestros algoritmos con REWARDS. Para poder comparar estos algoritmos consistentemente, vamos a utilizar siempre los mismo sumideros de tipos. Esto se debe a que lo qué realmente queremos comparar es como las decisiones de diseño detrás de estos algoritmos impactan en la precisión de los tipos recuperados. Para abordar este problema, reimplementamos la versión en linea de REWARDS en nuestro formalismo. Para evitar confusión con la implementación original, la llamamos REWARDS*.

Para realizar estos experimentos, los programas se corren en un monitor de ejecución para obtener una traza de ejecución y su registro de reserva de memoria. Cada programa se evaluará utilizando 5 trazas de ejecución distintas. Además se recolectarán estadísticas relevantes como el número de buffers y callsites. Luego, el módulo de seguimiento de tipos se utiliza para obtener tipos primitivos. Los tipos son inferidos usando los sumideros de tipos descriptos en 4.2.2. No todas las locaciones de programa son evaluadas en estas pruebas, sino solamente aquellas que son reservadas en la heap durante la ejecución. Cuando un buffer se libera, sus locaciones de programa se inspeccionan para evaluar qué tan efectivo fue el tipado del módulo de seguimiento de tipos. Si la traza termina y hay buffers sin liberar, los bytes de dichos buffers también son considerados en la evaluación.

6.1.1. Número de bytes con tipos detectados

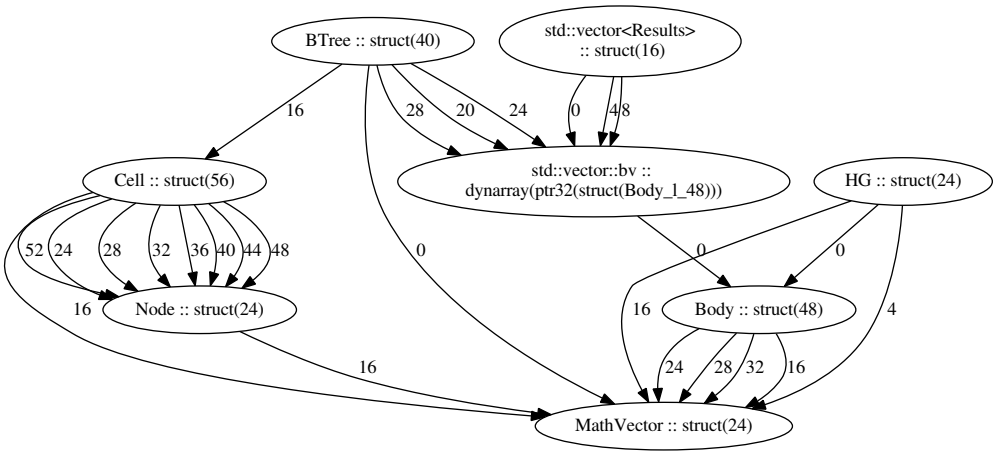
La más simple evaluación de como este formalismo lleva a cabo el tipado usando el módulo de seguimiento de tipos se obtiene con el número de bytes en buffer que se puede tipar. Los resultados por byte se resumen en la Tabla 4. Para cada algoritmo de tipado, mostramos tres columnas. La primera columna muestra el número de bytes tipados en los buffers. Esta columna nos muestra una medida cuantitativa de qué tan efectivo fue el tipado a nivel de bytes. La segunda columna muestra el número bytes en los buffers que no fueron tipados (su tipo es $\top$). La tercera columna muestra el número de bytes en buffers con conflictos de tipo (su tipo es $\bot$).

Estos experimentos muestran una mejora significativa de hasta un 60 % en el tipado de buffers los programa evaluados usando las variantes de TIPO comparando con REWARDS*. Estas mejoras son resultados de la reducción de conflictos de tipos. También, como se esperaba, la versión conservadora de TIPO que trabaja sin agrupamiento de locaciones que comparten el mismo tipo, minimiza los conflictos de tipos pero tiene una efectividad menor a TIPO-A y TIPO-LU.

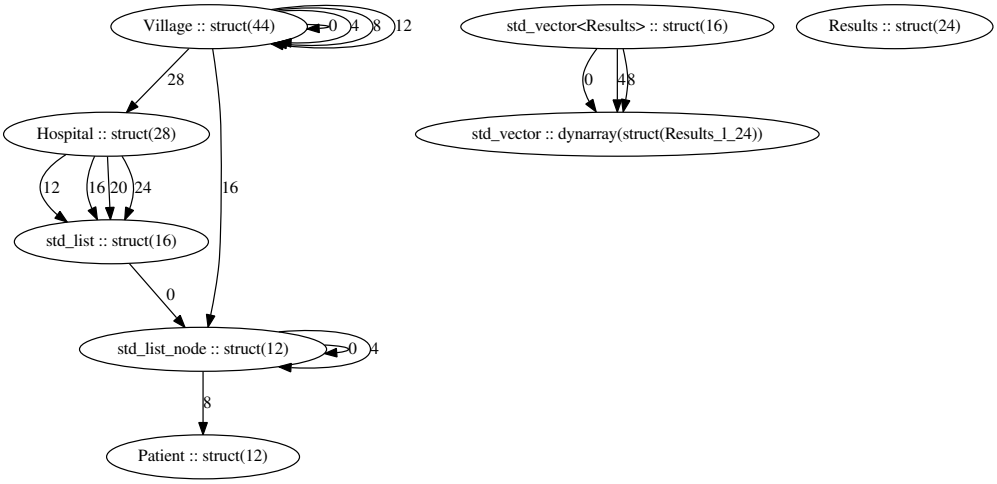
Otra interesante conclusión se relaciona con el uso de la unificación en TIPO-LU. Su uso no reporta mejoras significativas en los resultados. De hecho, en las trazas de ejecución de Pidgin evaluadas, produce más conflictos.

6.1.2. Refinamiento de Tipos Detectados

Otra relevante evaluación que hicimos con estos algoritmos fue comparar como ellos detectan tipos primitivos particulares. Claramente, un algoritmo resulta mejor si puede detectar tipos primitivos más refinados. Los resultados sobre cuales tipos primitivos fueron detectados son resumidos



(a) BHBmk



(b) HealthBmk

Figura 8: Grafos de tipos de referencia de los programas de ejemplo generados a partir de sus símbolos de depuración.

				TIPO-LU			TIPO-A		
	Bytes	**Buffers**	**Callsites**	**Tipado**	$\top$	$\bot$	**Tipado**	$\top$	$\bot$
HealthBmk	806,424	63,310	16	571,812	234,612	0	579,456	226,968	0
	100 %			71 %	29 %	0 %	71.9 %	28.1 %	0 %
BHBmk	330,884	12,993	23	321,096	9,761	0	319,012	11,872	0
	100 %			97.0 %	3.0 %	0 %	96.4 %	3.6 %	0 %
Pidgin	331,011	49,123	170	59,768	263,073	10,170	87,506	244,334	4171
	100 %			17,9 %	79 %	3 %	26,2 %	72,4 %	1,2 %

(a) Comparativa entre TIPO-LU y TIPO-A

				TIPO-C			REWARDS*		
	Bytes	**Buffers**	**Callsites**	**Tipado**	$\top$	$\bot$	**Tipado**	$\top$	$\bot$
HealthBmk	806,424	63,310	16	567,136	239,288	0	84,536	231,136	490,752
	100 %			70.3 %	29.7 %	0 %	10.5 %	28.7 %	60.8 %
BHBmk	330,884	12,993	23	307,424	23,460	0	291,560	10,353	28,971
	100 %			92.9 %	7.1 %	0 %	88.1 %	3.1 %	8.8 %
Pidgin	331,011	49,123	170	45,048	287,963	0	21,400	236,896	74,715
	100 %			13,5 %	86,4 %	0 %	6,4 %	71,1 %	22,4 %

(b) Comparativa entre TIPO-C y REWARDS*

Tabla 4: Resultados de la inferencia de tipos primitivos. Se muestra, del total de bytes reservados durante las ejecuciones, cuantos fueron tipados, no tipados ($\top$) y con conflictos ($\bot$) utilizando diferentes algoritmos.

para Pidgin en la Tabla 5. Para cada algoritmo TIPO y REWARDS*, mostramos cuantos tipos primitivos se localizaron en los buffers durante toda la ejecución.

Como podemos observar, para tipos más refinados, los resultados son similares a la evaluación de número de bytes tipados. Por otro lado, en los tipos más refinados, TIPO-LU detectó 14 % más punteros. Este hecho parece indicar que la unificación tiende a descubrir tipos más refinados con el costo de más conflictos.

6.1.3. Tiempos de Ejecución

En la Figura 9 comparamos los tiempos de ejecución de las variantes de TIPO con REWARDS*. Como podemos observar, los tiempos de ejecución de los algoritmos que requieren unificación son significativamente más largos: hasta 25 veces más lentos que las variantes de TIPO que no la

Pidgin	REWARDS*	TIPO-LU	TIPO-A	TIPO-C
num8	231	231	994	637
num16	0	0	2	0
num32	687	1079	2501	577
ptr32	325	1852	1625	1516
$\top$	48684	52615	48267	57593
$\bot$	14943	2034	834	0

Tabla 5: Número redondeado promedio de tipos primitivos detectados en las trazas de Pidgin.

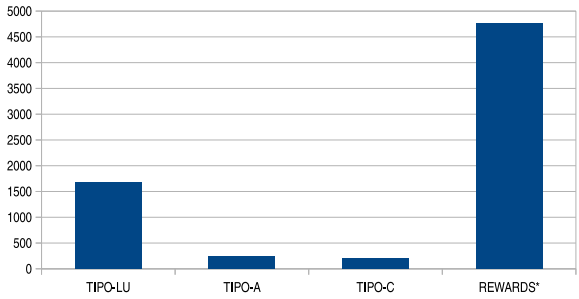


Figura 9: Comparación del tiempo promedio en segundos de los distintos algoritmos.

		TIPO-A				**TIPO-C**			
	Árboles de Callsite	Árboles de Tipos	Tipados	$\top$	$\bot$	Árboles de Tipos	Tipados	$\top$	$\bot$
HealthBmk	16	9	81.0 %	19.0 %	0 %	9	78.6 %	21.4 %	0 %
BHBmk	23	19	83.0 %	17.0 %	0 %	19	81.5 %	18.5 %	0 %
Pidgin	170	123	17.1 %	82.8 %	0.1 %	125	11.6 %	88.4 %	0 %

Tabla 6: Resultados del análisis global. Se incluyen los números de callsites, como también los números de árboles de tipos resultantes luego de combinaros usando la información de punteros.

utilizan. El uso de la unificación tiene un enorme costo computacional. Es importante notar que no se intentó implementar unificación usando una estructura de datos union-find para reducir el costo. La implementación de la unificación requeriría una versión modificada de dicho algoritmo. Además, observando los resultados del tipado de cada algoritmo, el tiempo adicional que utilizan nuestra implementación de unificación no parece justificarse.

6.2. Inferencia de Tipo de Objetos

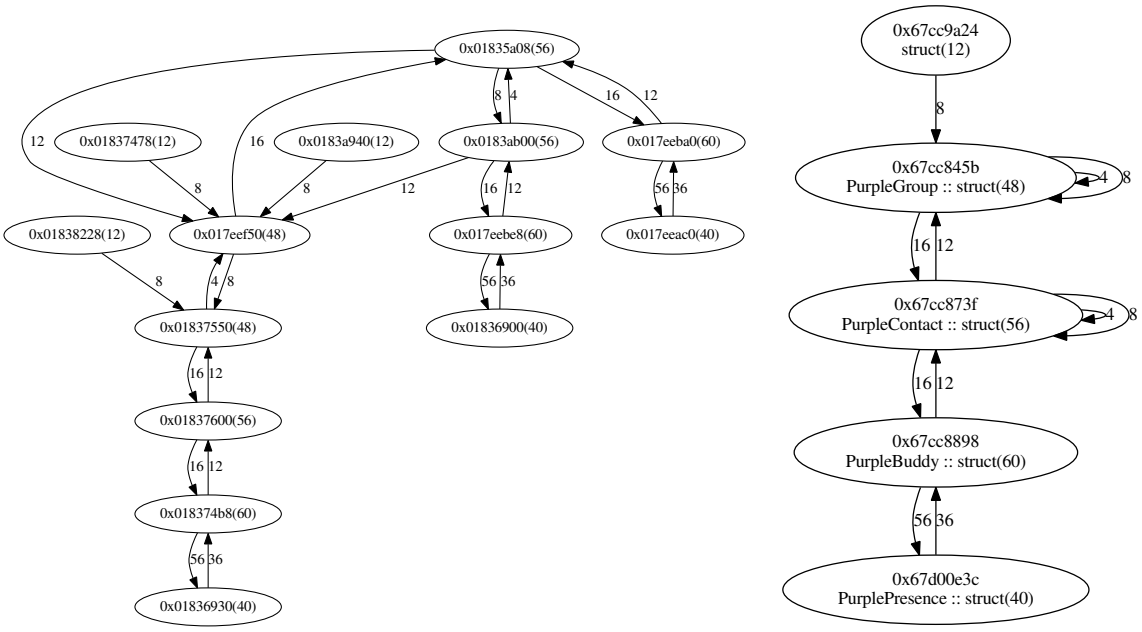
6.2.1. Detección de Tipos Primitivos en Árboles de Tipos

Luego de que el análisis de ejecución se realizó en cada traza, podemos extender la evaluación de cuantos bytes reservados en buffers pueden ser tipados en el análisis global examinando el número de bytes tipados en los árboles de tipos resultantes.

En la Tabla 6 podemos encontrar una comparación entre TIPO-C y TIPO-A entre los bytes tipados luego de aplicar la fase de análisis global. Se muestran estos algoritmos debido a que tiene la máxima diferencia entre los algoritmos TIPO en los tipos detectados en los buffers. Como podemos ver, estas diferencias luego del análisis global son más pequeñas.

6.2.2. HealthBmk y BHBmk

En la Figura 7 podemos observar los grafos de tipos inferidos para HealthBmk y BHBmk. Nuestro formalismo infiere correctamente tipos recursivos como Village o BTree, tanto como no recursivos, por ejemplo HG o Patient. La detección de arreglos dinámicos también se realiza de



(a) Grafo de heap (b) Grafo de tipos

Figura 10: Grafos parciales de heap y tipos del módulo libpurple.dll reconstruidos usando TIPO-LU.

manera exitosa. Estos arreglos dinámicos son parte de la implementación de vectores definidos en la biblioteca estándar de C++ de Microsoft.

Podemos también apreciar sus limitaciones: algunos conjuntos de callsites son inferidos como distintos tipos a pesar de ser el mismo. La combinación de árboles de callsite o árboles de tipos no es completamente efectiva: en el grafo de tipos de HealthBmk, las cuatro listas de Patient no son detectados como el mismo tipo, sino como tipos distinto. Además, el tipo que representa los pacientes es identificado correctamente como los elementos de estas listas, con una sola excepción: la lista referida en el offset 24 en el tipo Hospital. Un rápido análisis del código fuente revela que ningún elemento es insertado nunca en dicha lista. En BHBmk, sólo algunos de los distintos callsites de MathVector son identificados como el mismo tipo. Claramente existen posibilidades de mejora de los métodos hasta tener un sólo tipo correspondiente a MathVector, en vez de 9 distintos tipos de datos.

En BHBmk, es imposible realizar una combinación de ciertos árboles de tipos cuando se intenta combinarlos utilizando la información de punteros. El fallo de esta combinación es esperable debido a que existen varios punteros hacia Node y los subtipos correspondientes, Cell y Body no son compatibles entre sí (sus tamaños son distintos).

6.2.3. Pidgin

Un sencillo experimento fue realizado con Pidgin 2.4.1 para mostrar qué tan práctica puede ser el formalismo aquí definido. Quisiéramos detectar y analizar un reducido subconjunto de todos los buffers creados durante una ejecución. Nos enfocamos en estos buffers porque contienen datos privados del usuario: toda la información de los relacionada con la lista de contactos.

Configuramos Pidgin para comenzar con una cuenta de mensajería instantánea que contiene sólo tres contactos. Estos contactos están separados en dos grupos. Estos grupos y sus contactos pueden verse en el grafo parcial de heap de libpurple.dll en la Figura 10a. Adicionalmente, el grafo parcial de tipos correspondientes a estas estructuras de datos puede apreciarse en la Figura 10b. Fácilmente podemos detectar las estructuras de datos que contienen la información sensible a pesar que el porcentaje de bytes con tipos primitivos refinados inferidos es mucho menor que el de los programas de ejemplo. Esto es debido a que las estructuras de datos que se quieren detectar han sido, sin duda, leídas o escritas en la memoria o registros durante la ejecución. Por el contrario, un gran número de estructuras de datos son creadas durante la inicialización de un programa complejo como Pidgin, pero sólo algunas de ellas se utilizan efectivamente durante su inicialización.

6.2.4. La biblioteca estándar de C++ de Microsoft

La biblioteca estándar de C++ de Microsoft contiene una amplia colección de declaraciones y funciones definidas en un gran número de archivos de cabeceras. En particular, la librería de plantillas estándar (STL por sus siglas en inglés) provee un conjunto de contenedores y algoritmos implementados como plantillas de C++. Dos de los contenedores de la STL son usados en los programas de ejemplos: los vectores y la listas. Tanto los vectores como las listas son contenedores capaces de almacenar un secuencia lineal de elementos. La diferencia es que los vectores permiten una extensión y un acceso aleatorio eficiente, y las listas una inserción y borrado de elementos en cualquier posición. Es fácil pensar que sus estructuras de datos internas son completamente distintas.

A pesar de que nuestra investigación en ingeniería reversa nunca tuvo como objetivo el estudio de ninguna librería dinámica provista por Microsoft (por ejemplo, msvcp100.dll), la estructura interna de los vectores y las listas puede verse en la Figura 7 debido a que estos dos contenedores son implementados como plantillas C++ e incorporados de manera inline en cada módulo que los utiliza.

6.3. Selección de Algoritmo de Detección de Tipos Primitivos

La selección del algoritmo de tipado para tipos primitivos depende de los requerimientos para resolver problemas concretos.

Si se trata de detectar tipos de datos más refinados el uso TIPO-A o TIPO-LU resulta el más adecuado según nuestras evaluaciones. En particular, TIPO-A es una buena primera opción porque más rápido, por otro lado, en algunas circunstancias resulta que TIPO-LU detecta mejor los tipos primitivos más refinados, como los punteros.

Por otro lado, si lo que se busca es recuperar tipos de datos a medida que el programa se ejecuta, el algoritmo que pueda ser implementado y utilizado más eficientemente manteniendo una detección razonable resultará el mejor. En este caso, la selección recomendada es TIPO-C.

Finalmente, según nuestra evaluación, el uso de REWARDS no trae ninguna ventaja frente a los algoritmos TIPO.

7. Trabajos Futuros y Conclusiones

7.1. Trabajos Futuros

En esta sección describimos las mejoras que podemos realizar en un futuro a nuestro formalismo y sus diversas aplicaciones.

7.1.1. Mejorando la Inferencia de Tipos Primitivos

Un simple propuesta para mejorar la precisión de la inferencia de tipos primitivos es agregar más sumideros de tipos. Nuestro trabajo utiliza solamente un limitado conjunto de sumidero de tipos de instrucciones y sólo uno de invocación/retorno de una función conocida. Creemos que hay mucho potencial para mejorar la inferencia de tipo usando más sumideros de tipos. Por ejemplo, existe un gran número de símbolos de depuración de la API de Windows disponibles públicamente para poder mejorar la inferencia de tipos primitivos.

7.1.2. Mejorando el Grafo de Tipos

Tal como mencionamos en el capítulo anterior, el grafo de tipos resultante puede ser mejorado. Algunos nodos corresponde a los mismo tipos. Para superar esta limitación creemos que se pueden aplicar determinadas técnicas de Minería de Datos para obtener información que permita inferir qué nodos pueden corresponden a los mismo tipos. Por ejemplo, definiendo una medida de distancia adecuada entre árboles de tipos usando el látice de tipos primitivos para comparar sus campos, podríamos utilizar alguna técnica de clustering para agrupar tipos similares en el grafo de tipos. La confiabilidad de estas técnicas depende de qué tan disímiles sean los grupos de tipos y funcionará mejor con árboles de tipos con un número considerable de campos distintos, el cual suele ser el caso en los programas grande y complejos.

Otra interesante mejora en el grafo de tipos que estamos considerando para investigaciones futuras es detectar conjuntos de subtipos usando la información de punteros durante el proceso de combinación de árboles de tipos. La combinación por punteros podría fallar porque no hay razón para pensar que los detectados subtipos son compatibles entre sí. Usando esta información, podríamos crear conjuntos de subtipos e intentar encontrar alguna estructura común entre ellos para detectar el tipo padre.

7.1.3. Aplicaciones.

Creemos que este trabajo posee múltiple aplicaciones. Estamos empezando a trabajar en algunas de ellas. Por ejemplo, empezamos a explorar la posibilidad de la detección de punteros colgantes. Esta aplicación requiere solamente una parte de nuestro formalismo: la detección de tipos primitivos usando reglas de inferencia y la detección del grafo de tipos. Un trabajo recientemente publicado por Caballero et al., presenta *Undangle* [9], una herramienta para la detección temprana de punteros colgantes. La detección de esta clase de punteros puede ser realizada fácilmente definiendo un puntero colgante como un tipo primitivo más refinado que un puntero de 32-bit o 64-bit.

Otra aplicación factible en la que estamos trabajando es la generación automática de firmas de estructuras de datos. Estas firmas son usadas para identificar instancias de estructuras de datos y determinar que información almacenan [3,6,14,27,31]. La producción de dichas firmas es un proceso lento y costoso si son producto de un proceso manual, y requieren de acceso a la información de

depuración para ser realizadas. Nuestro formalismo puede incorporar un algoritmo para recuperar invariantes de valor en los campos y crear firmas utilizando la información de tipos inferida.

La implementación de una variante de TIPO ejecutándose paralelamente en una máquina virtual es otra idea con varias aplicaciones en trabajos futuros. Por ejemplo, nos permitiría detectar y neutralizar potenciales problemas de seguridad como punteros colgantes o desbordamiento de buffers antes de puedan ser aprovechados por un atacante.

7.2. Conclusiones

Los resultados de este trabajo indican que es posible recuperar el grafo de un programa disponible sólo como binario. Se plantearon tres algoritmos alternativos dinámicos de recuperación de tipos. Los resultados muestran que es posible recuperar los grafos de tipos de los programas de ejemplo sólo con pequeñas imprecisiones como diferentes nodos representando un mismo tipo. También recuperamos con éxito la información privada manipulada por un programa complejo permitiendo a un analista detectar y recuperar dicha información de un traza de ejecución.

Otra interesante conclusión de este trabajo es que es posible definir técnicas y algoritmos muy generales sin enfocarse en una aplicación particular. Decidimos hacerlo de esta manera porque pensamos que este formalismo puede ser adecuado para una multitud de aplicaciones en ingeniería inversa relacionadas con el análisis de estructuras de datos.

Finalmente, esperamos que nuestro formalismo y herramientas sean utilizadas en futuros trabajos de investigación en un futuro cercano para poder dar otro pequeño paso adelante en el apasionante campo de la ingeniería inversa.

8. Referencias

[1] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *IEEE Symposium on Reliable Distributed Systems*, New Delhi, India, October 2010.

[2] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International Conference on Compiler Construction*, Barcelona, Spain, March 2004.

[3] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Annual Computer Security Applications Conference*, Hsinchu, China, August 2008.

[4] Binary Analysis Plaform: The Next-Generation Binary Analysis Platform. `http://bap.ece.cmu.edu`.

[5] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, pages 446–449, December 1986.

[6] C. Betz. `http://sourceforge.net/projects/memparser/`.

[7] BitBlaze: Binary Analysis for Computer Security. `http://bitblaze.cs.berkeley.edu/`.

[8] Juan Caballero. *Grammar and Model Extraction for Security Applications using Dynamic Program Binary Analysis*. PhD in Electrical & Computer Engineering, Carnegie Mellon University, September 2010.

[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, Minneapolis, MN, July 2012.

[10] Juan Caballero, Pongsin Poosankam, Cristian Kreibich, and Dawn Song. Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, November 2009.

[11] Martim Carbone, Weidong Cui, Long, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conference on Computer and Communications Security*, Chicago, Illinois, November 2009.

[12] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *8th Conference on Operating Systems Design and Implementation (OSDI '08)*, December 2008.

[13] GNU DDD: Data Display Debugger. `http://www.gnu.org/software/ddd/`.

[14] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. Robust signatures for kernel data structures. In *ACM Conference on Computer and Communications Security*, Chicago, Illinois, November 2009.

[15] E. Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

[16] B. Unger G. Lomow, J. Cleary and D. West. A performance study of time warp. pages 50–55, February 1988.

[17] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Symposium on Principles of Programming Languages*, Saint Petersburg Beach, Florida, January 1996.

[18] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, pages 47–57, June 1984.

[19] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Transactions on Parallel and Distributed Systems*, pages 35–47, 1990.

[20] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, pages 243–260, 1992.

[21] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, February 2011.

[22] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, San Diego, CA, February 2011.

[23] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, February 2010.

[24] Pidgin: the universal chat client. `http://www.pidgin.im/`.

[25] Marina Polishchuk, Ben Liblit, and Chloë Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, January 2007.

[26] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999.

[27] Andreas Schuster. Searching for processes and threads in microsoft windows memory dumps. In *Digital Forensics Research Workshop*, Lafayette, Indiana, August 2006.

[28] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS 2011*, San Diego, CA, February 2011.

[29] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, December 2008.

[30] SQLite: self-contained, serverless, zero-configuration, transactional SQL database engine. `http://www.sqlite.org/`.

[31] Aaron Walters. The volatility framework: Volatile memory artifact extraction utility framework. http://www.volatilesystems.com/default/volatility.