Mimicry Monitors: Verification of programs with common code fragments

Felicitas Garcia[1]

Universidad de Buenos Aires
fgarcia@dc.uba.ar

Abstract In modern software development, systems evolve through small incremental changes. This raises a fundamental question in regression testing: when a previously verified program is modified, is it necessary to re-evaluate the entire system, or can the verification effort already invested be reused? This work explores and implements Mimicry Monitors (MM), a technique that capitalizes on the common fragments between two versions of a program. Its goal is to verify at runtime whether the behavior of a program under analysis (PUA) can be mimicked by a reference or oracle program (OP), without the need to execute the latter.

Keywords: Verification, Common fragments, Regression testing, Runtime monitoring, Fuzzing

Mimicry Monitors: Verificación de programas con fragmentos comunes

Resumen En el desarrollo de software moderno, los sistemas evolucionan mediante pequeños cambios incrementales. Esto plantea una pregunta fundamental en el testing de regresión: cuando se modifica un programa previamente verificado, ¿es necesario re-evaluar todo el sistema, o se puede aprovechar el trabajo de verificación ya realizado?
Este trabajo explora e implementa los Mimicry Monitors (MM), una técnica que capitaliza los fragmentos comunes entre dos versiones de un programa. Su objetivo es verificar en tiempo de ejecución si el comportamiento de un programa bajo análisis (PUA) puede ser imitado por un programa de referencia u oráculo (OP), sin necesidad de ejecutar este último.

Keywords: Verificación, Fragmentos comunes, Testing de regresión, Monitoreo en tiempo de ejecución, Fuzzing.

1. Introducción

1.1. Motivación

La verificación completa de sistemas complejos puede ser extremadamente costosa en tiempo y recursos computacionales. En proyectos de software industrial, estas verificaciones pueden tomar horas o incluso días, representando un cuello de botella en el ciclo de desarrollo. Sin embargo, cuando los programas comparten fragmentos significativos de código, gran parte de esta verificación puede ser redundante. El desafío central radica en determinar, sin ejecutar completamente ambos programas, si el comportamiento de un programa modificado puede ser imitado o replicado por el programa original ya verificado, y en ese caso, evitar la re-ejecución completa del proceso de verificación.
Realizar este análisis en tiempo real, permite tomar decisiones inmediatas sobre si continuar con la ejecución completa o no. Para abordar esta incógnita, este trabajo implementa y analiza los Mimicry Monitors, presentados en el trabajo de Postolski et al., 2024. Esta es una técnica innovadora de análisis dinámico que permite verificar en tiempo real si la ejecución de un programa bajo análisis (PUA) tiene una contraparte válida en un programa de referencia u oráculo (OP), evitando la ejecución del programa oráculo mediante la construcción de un autómata de monitoreo. Este simula sus posibles comportamientos, y al instrumentar el PUA con él, genera veredictos inmediatos sobre la validez de la ejecución basados en el análisis de fragmentos de código comunes.

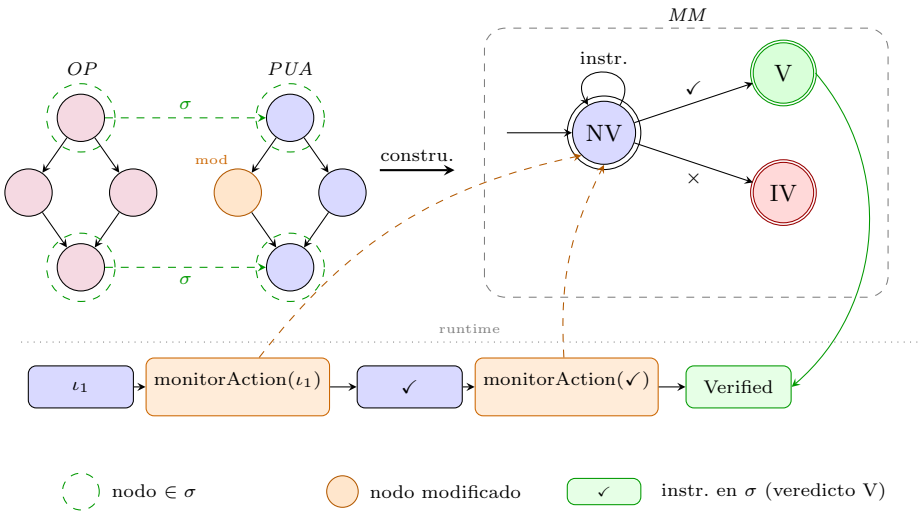

Figura 1: Visión general del Mimicry Monitor: construcción del monitor y uso en tiempo de ejecución.

La Figura 1 ilustra de forma conceptual este proceso. En el lado izquierdo se muestran los grafos de flujo de control (CFG) del programa oráculo (OP) y del programa bajo análisis (PUA). Ambos CFGs comparten ciertos nodos pertenecientes a una relación de emparejamiento que llamamos σ, representados con un borde punteado verde, que corresponden a fragmentos de código equivalentes entre ambos programas. El nodo resaltado en naranja indica una modificación introducida en el PUA. A partir de esta información, durante la fase de construcción, se genera un Mimicry Monitor (MM) representado como un autómata. En tiempo de ejecución, el PUA instrumentado invoca llamadas a `monitorAction` cada vez que alcanza un nodo relevante. El monitor procesa estas acciones y determina si la ejecución observada puede ser reproducida por el OP. Dependiendo de la transición tomada, el monitor puede producir un veredicto final: V (Verified) si la ejecución es compatible con el comportamiento del oráculo, o IV (Invalid) si se detecta una desviación. También es posible que sea temprano para emitir un veredicto final, y que permanezca en un estado NV (Non Verified).

1.2. Hipótesis

Este trabajo parte de la hipótesis de que, en situaciones donde los cambios entre versiones de un programa son pequeños, como ocurre comúnmente en escenarios de mantenimiento, es altamente probable que un porcentaje significativo de las ejecuciones del nuevo programa conserven un comportamiento compatible con el programa original.
En esos casos, el Mimicry Monitor permitiría determinar, en tiempo real, si una ejecución del PUA podría haber sido producida por el OP, evitando ejecutar el OP en paralelo. Esta capacidad de anticipar veredictos podría aprovecharse para reducir ejecuciones redundantes, detectar rápidamente cambios de funcionalidad, y, en definitiva, ahorrar tiempo y recursos en diferentes contextos, como lo puede ser el *testing* de regresión.

1.3. Contribuciones

El trabajo parte de las ideas del position paper de Postolski et al., 2024, que propone explotar los fragmentos comunes entre versiones de un programa para facilitar la verificación en tiempo de ejecución. A partir de una función de alineación semántica σ entre los CFGs de dos programas, se define el Mimicry Monitor (MM): un autómata que opera junto al PUA y determina si su ejecución podría haber sido producida por el OP, sin ejecutarlo. Sin embargo, el trabajo original presenta la construcción del MM a un alto nivel de abstracción y no provee una herramienta automatizada. Este trabajo cierra esa brecha presentando una implementación concreta y evaluando empíricamente su comportamiento sobre benchmarks reales.
Las contribuciones de este trabajo no se limitan a implementar los MM, sino que también incorporan modificaciones teóricas y prácticas orientadas a validar y extender el uso de Mimicry Monitors:

1. **Diseño conceptual.** Se introdujo un mecanismo de *poda temprana* y un paso de determinización para producir monitores más compactos, y se eliminó la noción de *Veredictos sobre Prefijos* por carecer de utilidad práctica.
2. **Implementación integral.** Se desarrolló el pipeline completo de construcción e instrumentación usando LLVM como infraestructura base, cubriendo análisis de accesos a memoria, construcción del CFG e inserción de llamadas al monitor en tiempo de ejecución.
3. **Evaluación empírica.** Se seleccionaron casos de estudio de GNU Core Utilities que simulan escenarios reales de regresión, se ejecutó la herramienta sobre 681 casos de prueba y se analizaron los resultados, mostrando que los MM emiten veredictos pertinentes en una porción significativa de las ejecuciones cuando los cambios entre versiones son acotados.

1.4. Trabajos Relacionados

La comparación dinámica de programas ha sido abordada desde distintos ángulos. Palikareva et al., 2016 proponen un enfoque en *Shadow of a Doubt*, que detecta divergencias entre versiones mediante ejecución simbólica, manteniendo una traza por cada versión. Postolski et al., 2019 extienden el análisis diferencial a propiedades de rendimiento; una sugerencia de ese trabajo motiva directamente el desarrollo de los MM. Jakobs y Pollandt, 2023 construyen un autómata de diferencia para *model checking* de regresión, pero su enfoque es insensible al flujo de datos. Las técnicas clásicas de selección de tests propuestas por Rothermel y Harrold, 1993, 1996 son conservadoras y menos precisas. Los enfoques de verificación relacional discutidos por Barthe et al., 2011; Godlin y Strichman, 2013 permiten detectar alineamientos complejos pero implican costos computacionales prohibitivos para uso en tiempo real.

Los MM se distinguen por ser sensibles tanto al flujo de control como al flujo de datos, operar sin ejecutar el OP, y emitir veredictos en tiempo real.

2. Conceptos Preliminares

En el desarrollo de los MM se consideran tres actores fundamentales: dos programas, denominados Programa Oráculo (OP, por Oracle Program) y Programa Bajo Análisis (PUA, por Program Under Analysis), y una relación de emparejamiento parcial σ entre los nodos de sus respectivos Control Flow Graphs (CFG). Esta correspondencia σ refleja fragmentos de código comunes o equivalentes entre ambos programas y constituye la base sobre la cual se definen las ejecuciones alineadas. En el alcance de este trabajo, no se aborda la construcción de σ, se considera que es dada como input por el usuario. Se dice que dos ejecuciones están σ-alineadas si son equivalentes en su comportamiento cuando se restringen a los nodos emparejados por σ. Intuitivamente, esto significa que, en los fragmentos de código compartido, ambas ejecuciones recorren las mismas instrucciones en el mismo orden, produciendo estados de memoria observacionalmente equivalentes. El MM tiene por finalidad verificar, en run-time, si la ejecución parcial del PUA

podría ser replicada por el OP en los fragmentos compartidos.

Para visualizar esta noción se creó una herramienta [1] con la que pueden apreciarse dos escenarios presentados de forma dinámica. En ella se encuentran dos programas ejemplo (OP y PUA), junto a su monitor final. Se pueden reproducir ejecuciones distintas del PUA que se corresponden con dos escenarios distintos: uno donde se obtiene un veredicto positivo, y otro donde no, y observar cómo se va recorriendo el monitor para obtener un veredicto.

2.1. Ejecuciones σ-alineadas

Formalmente, una ejecución se representa como una secuencia de triplas $\langle$before: State, node: CFG.Node, after: State$\rangle$, donde cada tripla describe el estado de memoria antes y después de ejecutar un nodo del CFG. Sean t_P y t_Q dos ejecuciones de programas P y Q respectivamente, y una correspondencia $\sigma \subseteq \text{Nodes}(CFG(P)) \times \text{Nodes}(CFG(Q))$. Sean $i_P = [0...\|t_P\| - 1]$ e $i_Q = [0...\|t_Q\| - 1]$ los conjuntos de índices válidos para ambas ejecuciones, de ser estas finitas. De lo contrario $i_P = \mathbb{N}$ e $i_Q = \mathbb{N}$. Se define la noción de σ-alineación como sigue:

Definition 1. *Sean t_P y t_Q ejecuciones de los programas P y Q, y σ un emparejamiento parcial entre nodos de sus CFG. Se dice que t_P y t_Q están* ***σ-alineadas*** *si existe una función parcial inyectiva estrictamente creciente $m \subseteq i_P \to i_Q$ tal que:*

- $\forall i \in dom(t_P) :\ t_P(i).node \in Dom(\sigma) \Rightarrow i \in Dom(m)$,
- $\forall j \in dom(t_Q) :\ t_Q(j).node \in Img(\sigma) \Rightarrow j \in Img(m)$,
- $\forall (i,j) \in m :\ (t_P(i).node,\ t_Q(j).node) \in \sigma$,
- $\forall i \in Dom(m) :\ Read(stmt_P,\ t_P(i).before) \equiv Read(stmt_Q,\ t_Q(m(i)).before)$.

Aquí, $stmt_P$ y $stmt_Q$ son los fragmentos de código asociados a los nodos correspondientes en cada ejecución:

$$stmt\_P = Stmt(t\_P(i).node) \qquad y \qquad stmt\_Q = Stmt(t\_Q(m(i)).node)$$

Esta definición asegura que, en los nodos compartidos, ambas ejecuciones acceden a las mismas áreas del estado y realizan transformaciones equivalentes, según el σ-emparejamiento definido. En consecuencia, si se verifica la σ-alineación entre la ejecución observada del PUA y alguna ejecución posible del OP, entonces cualquier propiedad que dependa únicamente del estado observable en dichos nodos será preservada.

[1] Se encuentra disponible en https://felicitasgarcia.github.io/SigmaVisualizer/

2.2. Mimicry Monitors

Los Mimicry Monitors son autómatas construidos a partir de los CFG del OP y del PUA, junto con la correspondencia σ de forma estática. La herramienta de monitoreo propiamente dicha funciona a través de la instrumentación del PUA con el autómata monitor. La versión instrumentada del PUA es capaz de emitir veredictos sobre cualquier ejecución en función de si existe (o no) una ejecución σ-alineada correspondiente en el OP. Crucialmente, este análisis se realiza sin ejecutar el OP, lo que representa una mejora sustancial en términos de eficiencia y escalabilidad.

El MM opera como un autómata que, en cada paso de ejecución del PUA que influya sobre el flujo de control del programa, actualiza su estado interno y clasifica la ejecución de acuerdo con uno de los siguientes veredictos:

- **NV (Non Verified):** existe un prefijo de ejecución del OP que está σ-alineado con la ejecución observada del PUA hasta ese punto. Aún no está determinado que se haya conseguido un veredicto concluyente.
- **V (Verified):** todas las posibles continuaciones del PUA son compatibles con una ejecución σ-alineada del OP.
- **IV (Impossible to Verify):** Ninguna continuación del PUA será determinada como Verificada.

Los veredictos V e IV son terminales, es decir que indican la finalización del monitoreo. Estos veredictos se emiten en tiempo real y pueden utilizarse para tomar decisiones en sistemas de testing, ejecución múltiple o fuzzing, por ejemplo abortando ejecuciones innecesarias o priorizando la exploración de caminos no cubiertos por versiones anteriores del software. Para asegurar que la técnica sea *sound*, el enfoque parte de ciertos supuestos. En primer lugar, se supone que las instrucciones OP no compartidas no producen errores y que los ciclos en código no compartido finalizan, lo cual garantiza *soundness* para veredictos NV. Esta suposición es razonable si se ha verificado que una ejecución OP termina exitosamente para una entrada dada. De manera similar, la correctitud de los veredictos V anticipados requiere que las instrucciones PUA no compartidas no generen fallas y que sus ciclos terminen adecuadamente.

3. Diseño e Implementación

En esta sección se utilizarán las nociones ya presentadas para ofrecer una explicación completa e ilustrativa del proceso de creación de un Mimicry Monitor. El proceso de *setup* para el monitoreo está compuesto por tres etapas principales, como se puede apreciar en la Figura 2:

1. **Procesamiento de los programas *input***: Esta etapa implica el análisis estático de los dos programas de referencia. A partir de estos programas se extrae información relevante que servirá como base para la construcción del monitor. En particular, se extraen los CFGs del OP y el PUA en su versión

dual (aristas etiquetadas), y se realiza un análisis de lecturas y escrituras sobre variables. La función σ, provista como entrada externa, vincula los nodos equivalentes entre ambos CFGs. Las escrituras fuera de σ son de particular interés, pues pueden desincronizar variables compartidas e invalidar lecturas posteriores en fragmentos comunes.

2. **Construcción del autómata monitor**: Utilizando los datos obtenidos en la etapa anterior, se genera un autómata que modela el comportamiento esperado del sistema. Este autómata puede ser no determinista en un principio, y requerirá transformaciones como la determinización y minimización para ser lo suficientemente compacto, para instrumentar el programa bajo análisis de forma *lightweight*.
3. **Instrumentación del programa bajo análisis con el monitor**: Finalmente, se modifica el programa objetivo, el que se desea monitorear, para insertar llamadas al monitor en puntos asociados a condiciones de control. De esta forma, durante la ejecución, el monitor podrá verificar si las transiciones observadas son consistentes con el modelo y emitir un veredicto si se detecta una desviación.

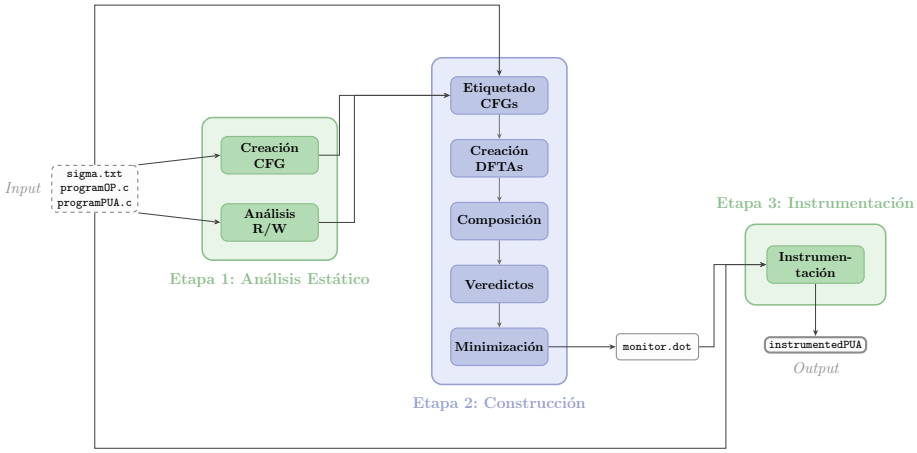

Figura 2: Pipeline de construcción de un Mimicry Monitor. Las tres etapas principales comprenden el análisis estático de los programas de entrada (Etapa 1), la construcción del autómata monitor (Etapa 2) y la instrumentación del programa bajo análisis (Etapa 3). El proceso de implementación se detalla con precisión en el trabajo de tesis de licenciatura de García, 2025, y se encuentra disponible en el repositorio asociado.[2]

3.1. Profundización sobre la construcción del autómata monitor

La segunda etapa es, sin duda, la más compleja a nivel conceptual. Como se observa en la Figura 2, consta de una serie de pasos más pequeños que operan sobre los CFGs hasta obtener el autómata monitor final. A continuación se presentan estos pasos.

Etiquetado. Las transiciones de los CFGs se etiquetan según su pertenencia a σ: las instrucciones comunes reciben la propia instrucción como etiqueta; las condicionales se distinguen con sufijos `:then`/`:else`; las escrituras fuera de σ que afectan variables compartidas reciben sufijos `OP` o `PUA`; el resto queda sin etiquetar.

DFTAs (Data Flow Tracking Automata). Por cada variable v escrita fuera de σ y leída dentro de ella se construye un par de autómatas de dos estados (sincronizado/desincronizado). Solo en estado sincronizado se permiten lecturas de v en fragmentos comunes; una escritura no alineada transiciona al estado desincronizado, y una escritura alineada lo restaura.

Composición paralela. Se construye el autómata producto $OP \parallel PUA \parallel DFTA_1 \parallel \cdots$, cuyos estados son tuplas $(q_{OP}, q_{PUA}, \vec{s})$. Las transiciones $\alpha \in \sigma$ requieren avance simultáneo de ambos CFGs con DFTAs en estado válido; las exclusivas de cada programa avanzan solo el componente correspondiente.

Veredictos. Los nodos del autómata producto se etiquetan estructuralmente (NV/IV/V) y luego se propagan por punto fijo: IV mediante mínimo, V mediante máximo. Se introduce una *poda temprana* que elimina sucesores de nodos IV alcanzables solo por transiciones internas del OP, no observables desde el PUA, preservando la coherencia semántica del monitor. Finalmente, los nodos V e IV se compactan en estados únicos.

Minimización. Sobre el autómata resultante se aplica una serie de transformaciones que buscan compactar el monitor lo más posible. Las mismas consisten del ocultamiento de etiquetas no condicionales y la clausura transitiva (Floyd-Warshall). Luego se aplica una minimización por bisimulación. Hasta este punto no se tiene asegurado el determinismo del monitor, por lo que se decidió incorporar un paso de determinización y una *segunda pasada de optimización* post-determinización (re-propagación IV, poda adicional y compactación final) para restaurar la coherencia semántica que la determinización puede desestabilizar.

4. Evaluación

La validación de la metodología propuesta para la construcción y uso de Mimicry Monitors requiere una evaluación que demuestre tanto la viabilidad

técnica como la efectividad práctica del enfoque desarrollado en escenarios reales de verificación.

La evaluación se centra en el análisis del comportamiento de Mimicry Monitors aplicados a diferentes herramientas de las GNU Core Utilities, aprovechando los conjuntos de pruebas existentes que estas utilidades incorporan. Esta aproximación experimental permite evaluar cómo el sistema responde ante casos de prueba reales que han sido diseñados y refinados a lo largo de años de desarrollo, proporcionando un entorno de evaluación robusto y representativo de escenarios de uso práctico.

La metodología experimental utiliza pares de versiones de Core Utilities seleccionadas estratégicamente para representar diferentes tipos de evolución de software: correcciones de bugs, adición de funcionalidades, refactorizaciones, entre otros. Para cada par de programas, se construye el Mimicry Monitor correspondiente y se ejecutan los tests asociados, registrando los veredictos obtenidos (V, IV, NV) junto con las trayectorias de ejecución que los producen. Este análisis permite caracterizar qué tipos de cambios entre versiones generan diferentes clases de veredictos y bajo qué condiciones.

Los experimentos buscan responder preguntas fundamentales sobre el comportamiento práctico del sistema:

- ¿qué proporción de tests resulta en ejecuciones σ-alineadas versus no alineadas?
- ¿cómo influyen las características específicas de cada programa en la distribución de veredictos?
- ¿qué patrones de ejecución conducen a terminaciones tempranas con veredictos definitivos versus aquellas que requieren ejecución completa?

4.1. Benchmark

Como fue anticipado en la sección anterior, el benchmark utilizado para evaluar esta herramienta fue extraído del proyecto de GNU Core Utilities, una colección de utilidades fundamentales del sistema operativo Unix/Linux que proporcionan funcionalidades básicas de manipulación de archivos, texto y procesos. La selección de este conjunto de programas se fundamenta en su amplia adopción, madurez del código, y disponibilidad de historial de versiones bien documentado.

4.2. Criterios de selección

Para el benchmark se seleccionaron cinco programas que presentan variaciones significativas en tamaño, complejidad algorítmica y patrones de uso: `cat`, `timeout`, `expand`, `mv`, y `ls`. Esta selección fue diseñada para cubrir un espectro representativo de las características típicas encontradas en software de sistemas. La Tabla 1 presenta los cinco programas seleccionados (subjects), junto con su tamaño en líneas de código para las versiones OP y PUA, la cantidad de tests disponibles, y los identificadores de commit correspondientes a cada versión analizada. En algunos casos fue necesario realizar adaptaciones menores al OP. Utilizar

el archivo original de un commit anterior podía generar incompatibilidades con el entorno de compilación actual o introducir complejidad innecesaria.

Programa	Líneas OP	Líneas PUA	Tests	Commit OP	Commit PUA
`cat`	813	813	23	dff821d	7386c29
`expand`	312	296	31	97807bf	28b1760
`ls`	5612	5612	459	0c195b6	0d04b98
`mv`	558	528	144	a6ab944	1040610
`timeout`	631	631	24	cb7c210	–

Tabla 1: Resumen de las características de los programas analizados, para sus versiones OP y PUA

Para cuatro de los cinco programas se identificó un commit específico que alterara su comportamiento de manera significativa, priorizando cambios que impactaran la lógica de control principal o introdujeran nuevas funcionalidades. En tres de estos casos, las modificaciones se concentraron en la función principal (`main`), facilitando el análisis de σ-alineamiento. El caso de `expand` requirió un enfoque especial, ya que los cambios se ubicaron en una función auxiliar, lo que requirió realizar inlining manual para incorporar la funcionalidad modificada dentro del flujo principal de ejecución.

Para un único programa, `timeout`, el PUA fue producido artificialmente para propósitos demostrativos de este análisis, sirviendo como ejemplo introductorio.

4.3. Resultados

Los resultados revelan patrones significativos en el comportamiento de las herramientas de línea de comandos evaluadas y proporcionan perspectivas valiosas sobre la efectividad del enfoque de verificación implementado.

Programa	Archivos de Prueba	Casos Totales	V (Verificado)	IV (Imposible Ver.)	Sin Veredicto	% Éxito (V/Total)
catPUA	4	23	14	8	1	60.87 %
expandPUA	10	31	12	19	0	38.71 %
lsPUA	59	459	0	459	0	0.00 %
mvPUA	49	144	51	90	3	35.42 %
timeoutPUA	4	24	1	18	5	4.17 %
TOTAL	**126**	**681**	**78**	**594**	**9**	**11.45 %**

Tabla 2: Resumen Cuantitativo de Resultados de Pruebas PUA

4.4. Evaluación General

Los resultados presentados en la Tabla 2 muestran la efectividad de los Mimicry Monitors aplicados a diferentes herramientas de GNU Core Utils. La evaluación comprende un total de 126 archivos de prueba que incluyen 681 casos de prueba individuales, revelando patrones distintos de comportamiento según el tipo y la naturaleza de las modificaciones entre versiones de los programas analizados.

Se puede observar que para algunos casos se obtienen ejecuciones que no alcanzan ningún veredicto. Más específicamente, el 1.32 % de los casos de test obtuvieron este resultado. Esto puede suceder si el programa termina por fuera de la estructura de control representada del monitor. Si bien en algunos casos es posible hacer *inlining* de las funciones llamadas para hacer un análisis más completo, y evaluar todos los puntos posibles de retorno de una función (incluso aquellos fuera del main), en este caso no se creyó primordial, ya que se buscaba dar una idea general del funcionamiento de los monitores. Se puede observar que `timeout` obtuvo la mayor cantidad de casos de este estilo, alcanzando un 20,8 %. Este resultado es coherente con la naturaleza del cambio implementado, dado que al incorporar una modificación que frecuentemente resulta en error, es probable que se requiera invocar métodos externos para abortar la ejecución del programa.

4.5. Análisis por categorías de programas

En base a los resultados obtenidos, es posible agrupar los subjects en tres categorías según la naturaleza e impacto de las modificaciones realizadas entre versiones.

Programas con modificaciones de bajo impacto: cat, mv y expand Las herramientas `cat`, `mv` y `expand` conforman una categoría de programas que comparten una característica fundamental: las modificaciones entre versiones afectan el programa de manera localizada, permitiendo la existencia de flujos equivalentes en ambas versiones. Esta categoría muestra porcentajes de éxito que varían entre el 35.42 % y el 60.87 %, estableciendo un rango considerable pero consistentemente positivo de casos verificados.

El programa `cat`, con su 60.87 % de éxito, representa el extremo superior de esta categoría debido a que las modificaciones entre versiones se concentran principalmente en la validación de parámetros y el manejo de casos *edge*, sin alterar significativamente el núcleo algorítmico.

Por otro lado, `expand` alcanza un 38.71 % de casos verificados, reflejando modificaciones más profundas en la lógica de transformación de caracteres. Las diferencias entre versiones afectan el manejo de tabulaciones y la expansión de caracteres especiales, introduciendo variaciones en el flujo de control que reducen la cantidad de fragmentos comunes ejecutables de manera σ-alineada.

La herramienta `mv` presenta un comportamiento intermedio con 35.42 % de éxito, donde las modificaciones se concentran en la validación de operaciones del sistema de archivos y el manejo de metadatos. Aunque estas modificaciones

introducen cierta complejidad en el flujo de control, el núcleo de la operación de movimiento y renombrado mantiene suficientes fragmentos comunes como para generar un porcentaje significativo de veredictos V.

Esta categoría demuestra que los Mimicry Monitors obtienen una cantidad considerable de veredictos positivos cuando las modificaciones entre versiones son acotadas, permitiendo que una proporción considerable del código original permanezca semánticamente equivalente y ejecutable de manera alineada.

Modificación Artificial: `timeout` El caso de `timeout` merece un análisis particular debido a la naturaleza de las modificaciones introducidas. En este caso, se agregó una modificación de forma artificial, es decir que no estaba vinculada con ningún commit del proyecto. Esto se debe a que fue uno de los ejemplos introductorios, y resultaba de interés observar la proporción de veredictos IV cuando el cambio introducido alteraba muy claramente el funcionamiento del programa.

Con apenas un 4.17 % de casos verificados, este resultado nos demuestra que el Mimicry Monitor efectivamente pudo reconocer que la mayor parte de las ejecuciones atravesaban este código agregado que cambiaba el resultado de retorno. La modificación introducida consistió en alterar el valor de salida del programa de manera uniforme, estableciendo un código de retorno específico independientemente del resultado real de la operación.

Esta modificación, aunque aparentemente menor, tiene un impacto devastador en la capacidad de σ-alineación del programa. El cambio sistemático del valor de salida crea una divergencia fundamental en el estado final del programa que el Mimicry Monitor detecta tempranamente, resultando en veredictos IV para la gran mayoría de los casos de prueba. Los 18 casos con veredicto IV de los 24 totales reflejan esta situación.

Divergencia Estructural Fundamental: `ls` El programa `ls` representa el caso más extremo en la evaluación, con un 0.00 % de casos verificados a pesar de contar con 459 casos de prueba distribuidos en 59 archivos. Este resultado no refleja una falla en la implementación de los Mimicry Monitors, sino una situación donde las modificaciones entre versiones han creado una divergencia estructural fundamental que impide cualquier forma de σ-alineación.

El análisis del código revela que las modificaciones en `ls` afectan variables críticas que controlan el flujo principal del programa desde etapas muy tempranas de la ejecución. Estas variables actúan como puntos de decisión que determinan qué ramas del código serán ejecutadas, y su modificación crea un efecto cascada que propaga diferencias a través de toda la estructura de ejecución del programa.

Cuando el Mimicry Monitor detecta que una variable crítica ha sido modificada fuera del matching σ y posteriormente es utilizada para tomar decisiones de control de flujo dentro de fragmentos que sí pertenecen al matching, el sistema de tracking de data-flow inmediatamente invalida la posibilidad de alineación.

Esta situación se traduce en un veredicto IV temprano y definitivo, haciendo imposible que cualquier ejecución posterior genere un veredicto V.

4.6. Implicaciones para las Aplicaciones Prácticas

Estos resultados revelan que la efectividad de los Mimicry Monitors está intrínsecamente ligada a la naturaleza de las modificaciones entre versiones de software. En contextos de testing de regresión, las herramientas de la primera categoría pueden beneficiarse significativamente de la terminación temprana cuando se alcanza un veredicto V, particularmente en `cat` donde aproximadamente el 60 % de los casos pueden ser resueltos sin ejecución completa.

Para aplicaciones de fuzzing diferencial, los casos como `timeout` y `ls` proporcionan información valiosa al identificar rápidamente que toda nueva entrada dirigirá la ejecución hacia comportamientos no equivalentes, permitiendo enfocar los recursos de generación de inputs en la exploración de estas diferencias.

5. Conclusión

Este trabajo presentó una implementación completa y una evaluación empírica de los Mimicry Monitors sobre cinco herramientas de GNU Core Utilities. Los resultados confirman parcialmente la hipótesis central: es posible evitar ejecuciones redundantes mediante el análisis de fragmentos comunes, con hasta un 60,87 % de terminación anticipada en `cat`. La efectividad depende de la naturaleza de las modificaciones: cambios localizados (`cat`, `mv`, `expand`) favorecen veredictos V; cambios en variables centrales (`timeout`) generan mayoría de IV; y modificaciones estructurales profundas (`ls`) imposibilitan cualquier alineación. Aunque los MM no constituyen una solución universal, son especialmente valiosos en escenarios de mantenimiento incremental y pipelines de integración continua.

5.1. Trabajo futuro.

Las principales líneas de trabajo propuestas se organizan en torno a cuatro ejes. En primer lugar, la construcción automática de σ mediante análisis semántico, que eliminaría la necesidad de proveer manualmente la función de emparejamiento. En segundo lugar, la medición del impacto temporal de la instrumentación, lo que permitiría evaluar el costo real de monitoreo en escenarios de uso continuo. En tercer lugar, la extensión a otros lenguajes soportados por LLVM ampliaría el alcance del enfoque más allá de C, abriendo la posibilidad de aplicar Mimicry Monitors en ecosistemas como C++, Rust o Swift. Finalmente, una línea de particular interés es la integración de los MM en pipelines de *fuzzing*, donde los veredictos en tiempo real permitirían guiar la generación de inputs hacia caminos de ejecución no cubiertos por versiones anteriores del software, priorizando la exploración de divergencias y reduciendo ejecuciones redundantes.

Referencias

Barthe, G., Crespo, J. M., & Kunz, C. (2011). Relational Verification Using Product Programs. *FM 2011: Formal Methods*, 200-214. https://doi.org/10.1007/978-3-642-21437-0_17

García, F. (2025). MimicryMonitors: Verificación de programas con fragmentos comunes. Tesis de Licenciatura en Ciencias de la Computación. https://gestion.dc.uba.ar/media/academic/grade/thesis/tesis_Felicitas_Garc%C3%ADa.pdf

Godlin, B., & Strichman, O. (2013). Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, *23*(3), 241-258. https://doi.org/10.1002/STVR.1472

Jakobs, M.-C., & Pollandt, T. (2023). diffDP: Using Data Dependencies and Properties in Difference Verification with Conditions. En P. Herber & A. Wijs (Eds.), *iFM 2023 - 18th International Conference on Integrated Formal Methods* (pp. 40-61, Vol. 14300). Springer. https://doi.org/10.1007/978-3-031-47705-8_3

Palikareva, H., Kuchta, T., & Cadar, C. (2016). Shadow of a doubt: testing for divergences between software versions. *Proceedings of the 38th International Conference on Software Engineering*, 1181-1192. https://doi.org/10.1145/2884781.2884845

Postolski, I., Braberman, V., Garbervetsky, D., & Uchitel, S. (2024). Verification of Programs with Common Fragments. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 487-491. https://doi.org/10.1145/3663529.3663783

Postolski, I., Braberman, V. A., Garbervetsky, D., & Uchitel, S. (2019). Simulator-based diff-time performance testing. *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, 81-84. https://doi.org/10.1109/ICSE-NIER.2019.00029

Rothermel, G., & Harrold, M. J. (1993). A Safe, Efficient Algorithm for Regression Test Selection. En D. N. Card (Ed.), *Proceedings of the Conference on Software Maintenance (ICSM 1993)* (pp. 358-367). IEEE Computer Society. https://doi.org/10.1109/ICSM.1993.366926

Rothermel, G., & Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, *22*(8), 529-551. https://doi.org/10.1109/32.536955