Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

# XEGMAP: Adaptive Symbolic Execution for Race Condition Detection in Binary Programs

**Keith Makan**

k3170313370gmail.com

University of the Western Cape

**Mehrdad Ghaziasgar**

University of the Western Cape

**Research Article**

**Keywords:** symbolic execution, race conditions, binary analysis, symbolic explosion problem, vulnerability analysis

**Posted Date:** June 23rd, 2026

**DOI:** https://doi.org/10.21203/rs.3.rs-9969464/v1

**License:** This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

**Additional Declarations:** No competing interests reported.

# XEGMAP: Adaptive Symbolic Execution for Race Condition Detection in Binary Programs

**Abstract**

Identifying race conditions in Commercial Off-The-Shelf (COTS) binary-only programs is challenging due to the lack of source code and the scalability limits of symbolic execution, such as state explosion and high memory consumption. We present XEGMAP, an adaptive symbolic analysis technique that strategically guides state exploration. XEGMAP utilizes a two-phase process: NEGMAP, a naive execution phase, followed by DEGMAP, a directed phase triggered by the detection of semantic guidance. In the absence of symbols, the tool prioritizes the detection of Global Memory Access Points to pinpoint potentially race-prone thread interactions using a hybrid lock-set and happens-before analysis.

Our evaluation demonstrates that XEGMAP significantly enhances code coverage over existing methods in the Angr framework. We show that XEGMAP can achieve a code block count of up to 36,755, metrics comparable to state-of-the-art fuzzing engines. Furthermore, XEGMAP consistently discovers races in ground-truth binaries where stochastic and loop-focused approaches in angr fail, successfully identifying up to 360 GMAPs in complex COTS environments without requiring full source-level semantics.

*Keywords:* symbolic execution, race conditions, binary analysis, symbolic explosion problem, vulnerability analysis
*PACS:* 0000, 1111
*2000 MSC:* 0000, 1111

## 1. Introduction

Programs are often required to handle sensitive information, due to this they are often scrutinized for behaviour that violates security and usability properties. One means of scrutinizing programs is to subject them to automated analysis which uncovers security vulnerabilities without the need for

constant human intervention. Automated detection of race condition vulnerabilities in binary programs is proving to be a very challenging problem [1]. Due to this race conditions can confound even the most advanced and contemporary analysis methods since it brings with it many sources of complexity. Furthermore, automated analysis is often burdened with the need of complex compilation dependencies and execution/emulation environments. As far as current reading, one of the most effective means to forgo input or environment is to subject binary programs to symbolic analysis, since it brings with it the power to unearth or determine input needed to reach a given execution state.

Incorporating symbolic and binary analysis for the purpose of race condition detection brings with it a number of potential of complexities; in that it blends together 3 disciplines; each contributing their own hard sub-problem namely: (i) Binary format interpretation: which introduces what we term the *Loss of Semantic Information Problem* (LSIP), a phenomenon caused by modern compiler pipelines stripping away semantic information (ii) Concurrency analysis: which suffers from scaling problems due to the need to account for combinations of thread interleaving, we term the *Thread Interleaving Enumeration Problem* (TIEP) (iii) Symbolic Execution: which is greatly deterred by the growth in the number of states, which could deter or disable meaningful analysis termed the *Symbolic Explosion Problem* (SEP). Contemporary and effective approaches need to address these 3 sources of complexity in order to provide a meaningful solution that does not require source code, complex environments or input samples in order to achieve detection. Due to the growing need for binary analysis tools, especially in security contexts [1, 2, 3], frameworks like Angr [4, 5] provide a means to automate analysis of binaries while providing additional powers of analysis like (i) symbolic execution, (ii) virtual emulation (iii) binary interpretation all with pythonic interfaces which allow for modification and improvement.

We introduce an approach called XEGMAP a binary symbolic race detection approach implemented in Angr, which is capable of adapting to the presence of semantic information by fielding two state exploration algorithms that it switches between: (i) a naive mode that monitors for available semantic information or semantic-"like" information to guide symbolic execution usefully and (ii) a mode which incorporates discovered semantics and drives execution toward more shared memory interaction. XEGMAP achieves high block coverage for some samples often scaling to 36000 blocks, exceeding some concolic based fuzzing approaches[6]. Additionally XEGMAP shows

consistent GMAP discovery, out performing current exploration techniques in angr, across a large sample set of COTS binaries and detects races reliably in prepared test cases.

This paper is structured as follows: Section 2 provides a theoretical background on key concepts / techniques used in race conditions, symbolic execution and binary analysis; Section 3 provides a literature on known approaches to binary symbolic race condition detection, Section 4 provides a brief overview of the XEGMAP design and Section 5 walks through the NEGMAP and DEGMAP algorithms which comprise XEGMAP, Section 6 details the evaluation done on XEGMAP to prove its race detection capabilities.

## 2. Theoretical Background

This section provides a theoretical background on key concepts / techniques used in race condition vulnerability detection, including race conditions described in Section 2.1, lock-set analysis described in Section 2.2, happens-before analysis described in Section 2.3, symbolic execution discussed in Section 2.4 and binary analysis discussed in Section 2.5.

### *2.1. Race Conditions*

The common theory is that, in order for a race condition to exist, two or more threads should be able to interact with a single memory address/store, with at least one accessing said memory store through a write operation [7]. Race conditions may seem purely data driven at first, but they can often have a direct impact on execution flow and integrity. For instance threads/execution contexts with conditional constructs—if statements, while loops etc.—depend on specific values stored in memory—may encounter non-deterministic computation in configurations where the outcome of said memory address is reliant on the scheduling or execution order of threads. This non-determinism arises due to the varying sequences in which threads access and modify shared memory, potentially leading to different results each time the program is executed. This is since the kernel/operating system is responsible for scheduling threads, which may vary depending on factors that are (i) out of control of the thread programmer and (ii) varying depending on factors of the environment i.e. memory usage, disk usage, peripheral interaction etc. Additionally race conditions that cause *Use After Free* (UAF) flaws

demonstrate direct impact on execution flow and security properties due to their propensity to cause arbitrary code execution vulnerabilities [8].

Because of the significant consequences that race conditions can entail, it becomes crucial to accurately analyze, predict and detect circumstances during execution that lead to potentially race-able memory access. Often this means addressing the TIEP. We find that unless precluded by some technicality, most approaches to race detection must issue some address or strategy to enumerate threads. Typically approaches resort to a *Partial Order Reduction* (POR) technique which usually works to exclude inter-leavings and/or execution paths that do not contribute "racey" access.

The problem of detecting race conditions in programs has been shown to be NP-complete [9], but research studies have proposed a number of approaches to detect a sub-class of all possible race conditions. This is especially achievable due to constructs in modern programming languages and the ability to emulate execution, as well as introspect program behavior via memory and other resource usage.

The following section 2.2 introduces a concept which modern race detectors analyze the usage of the above mentioned constructs, i.e. locks.

### *2.2. Lock-Set Analysis*

Programs have constructs like *Locks*, *Mutexes* and *Barriers* which help explicitly establish the context of a piece of code with relation to concurrent execution. Unfortunately one lock or single mutex for an entire code base seems impractical, and thus modern lock usage tends to scale into having multiple locks for multiple variables which goes wrong very quickly, especially for code as critical as that of the Linux Kernel. The overzealous use of locks regularly culminates in a phenomenon called *Dead Locking* in which a program blocks completely due to an incorrect interleaving of locking operations. Due to this, multi-threaded applications must walk a tight rope of too much lock interaction causing blocking and too little causing serious race-able access which may totally ruin program security properties. *Lock-Set Analysis* (LSA) determines whether locks are used in such a way that races are not possible.

In order to clearly express situations that cause dead locking and raceable access; we formalize this in order to provide concise and clearly defined statements on what a lock race condition is. First a few preliminary definitions.

Let $\exists t \in \mathbb{N}$ which represents the thread id of every thread.

For every thread $t$, analysis with LSA maintains the following sets:

- $L(t, e) : \mathbb{N} \times \mathbb{E} \longrightarrow \{l \| \forall l \in \mathbb{L}\}$ which is the set of locks currently held by thread $t$, once event $e$ has transpired. $\mathbb{L}$ which is a subset of all the locks held through the life time of the program.

- $C(t, v) : \mathbb{N} \times \mathbb{V} \longrightarrow \{l \| \forall l \in \mathbb{L}\}$ which is the full set of locks required for safe interaction during the access to a variable $u$. Where $C(t, v) \subseteq \mathbb{L}$.

Given the above defined symbols, LSA accounts for, and enforces, lock behaviour during interactions with global variables in the following manner and using the following defined operations and events:

- **Acquiring a lock**: An event $e \in \mathbb{E}$ is termed an acquisition of a lock $l$ by a thread $t$ is denoted: $a_t(l)$. The operation serves to add $l$ to the set of currently held locks, given by: $a_t(l) \Rightarrow L(t, e) \cup \{l\}$.

- **Releasing a lock**: An event $e \in \mathbb{E}$ is termed an releasing of a lock $l$ by a thread $t$ is denoted: $x_t(l)$. The operation serves to remove $l$ to the set of currently held locks, given by: $x_t(l) \Rightarrow L(t, e) \setminus \{l\}$.

- **Reading a global variable**: An event $e \in \mathbb{E}$ is termed a read operation on a variable $u$ by thread $t$, denoted $r_t(u)$ is given by: $r_t(u) \Rightarrow C(u, e) \cup L(t, e)$.

- **Writing to a global variable**: An event $e \in \mathbb{E}$ is termed a write operation on a variable $u$ by thread $t$ denoted $w_t(u)$ given by: $w_t(u) \Rightarrow C(u, e) \cup L(t, e)$. Please note the locks that thread acquire for reading are free to be distinctly defined from those taken to write to a variable, namely: $L(t, w_t(u)) \neq L(t, r_t(u))$.

Based on the above, it is possible to obtain the two formal definitions of race conditions which are principal to LSA. The definition of a simple race condition which is a race condition arising from the absence of locks as in Definition 2.1.

**Definition 2.1** (Simple Lock-Set Race Condition)**.** Let there exists events $r_t(*)$ a read event and $w_{t'}(*)$ a write. Where $*$ indicates application to any variable. The event set $\{w_{t'}(*), r_t(*)\}$ constitutes a simple race condition iff:
$L(t, r_t(*)) = \emptyset \wedge L(t', w_{t'}(*)) = \emptyset$ when $t \neq t'$.

In the presence of locks, race conditions can occur as defined in Definition 2.2.

**Definition 2.2** (Complex Lock-Set Race Condition)**.** Let $u$ be a variable involved in an event $e$ with a read and write in a complete set of threads $t$ and $t'$ as follows $e \in \{r_t(u), w_{t'}(u)\}$. Event $e$ constitutes a lock-set race condition if $L(t, r_t(u)) \cap L(t', w_{t'}(u)) = \emptyset$ when $t \neq t'$. Namely, both threads may hold locks but if none of them are in common, the thread will not block and unsynchronized or "racey" access is possible.

Lock-Set race conditions as formally defined above will include any thread interaction that doesn't have consistent locking disciplines. The next section expands some formal definitions on happens-before analysis. The section will also hybridize the lock-set race conditions defined above to cover both HBA contingent races as well as those attributed to lock-set disciplines.

*2.3. Happens Before Analysis*

In order to establish awareness of concurrent events many approaches in race detection resort to *Happens-Before Analysis* (HBA). HBA relies on deriving a Lamports *Happens Before* (HB) relation in order to establish a partial ordering over events in concurrent threads/processes which allows for inference of race-prone memory and resource accesses, i.e., accesses that do not harbor any indication of strict or any other imposed ordering. This is possible by tracking certain events that have an impact on partial ordering and exploring the different ways they can interact under happens-before.

In order to represent the partial ordering of events we need to be able to map an event into an ordered set, we will term this globally ordered set of events $\mathbb{T}$ defined as follows:

**Definition 2.3** (Happens Before Relation)**.** We define a Global Event Clock as $\mathbb{T} : E \longrightarrow \mathbb{Z}^+$ which maps evens to their real ordering in execution. The Global Event Set is defined as follows: $E = \{e \| e_i \xrightarrow{hb} e_j\}$ iff $\mathbb{T}(e_i) < \mathbb{T}(e_j)$; where $e_i, e_j \in E$ and $\mathbb{T}(e_i), \mathbb{T}(e_j) \in \mathbb{Z}^+$. We term the integers $i, j$ to be the Event IDs of their respective events. We assume that in the absence of other concurrency impacting events, instructions will follow the contiguous ordering indicated by their event IDs.

Awareness of thread interactions is key to modeling concurrency of events; for this reason, we must represent thread creation and destruction. We omit definitions of post-wait join ordering since it's outside of our scope. Fork-join operations are defined as follows:

**Definition 2.4** (Fork/Join Event Happens Before Ordering)**.** A fork event, $e_f$ for the thread $t$ is of the form $e_f \implies f(t)$, a join event $e_j$ from the same thread is written as $e_j \implies j(t)$, are related via happens before in the following way: $e_f \xrightarrow{hb} e_j \forall e_f, e_j \in \mathbb{E}$. This means to say, all fork events in a given thread, always "happen before" all join events.

We can now define the way events effect ordering, essentially showing how happens-before relations can be draw between certain impacting events. The following definition lays out how locks and fork/join operations effect happens-before relationships.

**Definition 2.5** (Lock Event Happens Before Ordering)**.** Lock-Release events may affect happens-before relationships in the following way, for a given thread $t$, variable $u$ and associated lock $l$: $a_t(l,u) \xrightarrow{hb} x_t(l,u)$ iff $l \notin C(t,u)$, since we cannot take the same lock twice. The opposite is implied as well, $x_t(l,u) \xrightarrow{hb} a_t(l,u)$ iff $l \in C(t,u)$. That is, acquiring a lock $l$ for variable $u$ in thread $t$ must "happen before" all events that constitute a release of the lock.

We can also define transitivity for happens-before events extending definitions simply: for events in the following strict order in an isolated thread context $e_i$ $e_j$, $e_k$: $e_i \xrightarrow{hb} e_k$ iff $e_i \xrightarrow{hb} e_j \bigwedge e_j \xrightarrow{hb} e_k$ since this implies that $\mathbb{T}(i) \leq \mathbb{T}(j) \leq \mathbb{T}(k)$ because $\mathbb{T}(i), \mathbb{T}(j), \mathbb{T}(k) \in \mathbb{Z}^+$.

Given the previous definitions we can derive an expression for a race condition which takes into account happens-before impacting events and lock-set behaviour, as follows:

**Definition 2.6** (HB/LSA Race Condition)**.** We say that events $e_i, e_j$ as part of threads $t, t^{'}$ respectively, constitute a race condition iff:

$$e_j \xrightarrow{hb} e_i \bigwedge e_i \xrightarrow{hb} e_j \bigwedge \left( L(t, e_i) \bigcap L(t^{'}, e_j) = \emptyset \right)$$

when $t \neq t^{'}$. We write that $e_i$ "races" with $e_j$ as $e_i \leftrightarrow e_j$.

We left open the definition for how events are reduced to positive strictly ordered integers. This gap is filled by the implementation of a *Fork-Join Graph* (FJG), which constitutes non-cyclic graph data-structure that tracks concurrency-impacting events.

FJGs are data structures that capture the number of threads that are active at a given time during execution, as well as the relationship of these

threads to join events during their lifetime [10]. FJGs have been used for HBA in a number of studies, some of which include various improvements and extensions to the data structure [11]. The following definition proposes simplified notation in order to describe some properties of a FJG as applied in XEGMAP.

**Definition 2.7** (Fork-Join Graph)**.** A FJG is a DAG $F = \{V, D, V_{src}, V_{sink}\}$ where $V$ is a set of vertices $v$, $D$ is a set of edges, $V_{src}$ is the source vertex or starting point of execution, and $V_{sink}$ is the sink vertex or endpoint of the execution, such that there is a path from $V_{src}$ to all other vertices $v$, denoted $V_{src} \xrightarrow{F} v \; \forall \; v \in V$ and a path from all vertices $v$ to $V_{sink}$, denoted $v \xrightarrow{F} V_{sink}$ $\forall \; v \in V$[1]. The following properties are also implied:

4.7.1 **Vertices:** Vertices $V$ in $F$ represent a set of Thread IDs of known and executing threads. These vertices are used to scrutinize events with parallelism taken into context. Formally, a mapping function is defined to express this property as follows: $\exists \tau : V \longrightarrow \mathcal{P}(T)$ where $\exists t = \tau(v)$ where $\tau(v) \subseteq T \; \forall v \in V$ and $t \in \mathbb{Z}^+$. Using the mapping $\tau$ the following constraints can be placed on Thread IDs with respect to vertex concurrency.

4.7.1.0 *Vertex Degrees*: We denote the degree by the mapping of a function defined as follows $\delta : V \longrightarrow \mathbb{Z}^+$. In-degree of a vertex is denoted as $\delta^< : V \longrightarrow \mathbb{Z}^+$, which can be 0 but is always a positive integer, and out is noted $\delta^> : V \longrightarrow \mathbb{Z}^+$. This property may be useful in predicting when thread concurrent events end or can be predicted to end, aiding graph partitioning.

4.7.1.1 *Source-Sink*: $V_{src}$ and $V_{sink}$ represent points in execution where at least a single execution context is active and no other threads exist in parallel: $|\tau(v)| = 1$. For vertices $v^{'} \in V_{sink}$ and $v \in V_{src}$ we can write: $\left(\delta^<(v^{'}) = 1 \bigvee \delta^>(v) = 1\right)$.

4.7.1.2 *Thread Creation:* If $\epsilon$ is a creation event, the active thread set grows strictly: $\tau(w) = \tau(u) \cup \{t_{new}\}$ such that $|\tau(w)| > |\tau(u)|$.

[1]Note that this relation seems analogous to the reach-ability for SE, the approach presented here exploits this fact by annotating SE states with fork-join meta-data.

4.7.1.3 *Thread Join/Exit:* If $\epsilon$ is a join or destruction event, the active thread set shrinks strictly: $\tau(w) = \tau(u) \setminus \{t_{end}\}$ such that $|\tau(w)| < |\tau(u)|$.

4.7.1.4 *Internal Steps:* For standard execution steps not affecting concurrency, $|\tau(w)| = |\tau(u)|$.

4.7.2 **Edges:** Edges in $F$ represent events that impact concurrency. In the proposed implementation, edges will either be (i) *creation* events which introduce new threads to an execution context or (ii) *join* events which impose a wait on events in another thread until termination of a given thread. Edges are associated to events between vertices, which may impact thread live-ness. The set of edges is defined as $D = \{(v, e, w) \| v < e < w \bigwedge \forall v, w \in V\}$ where $e \in \mathbb{E}$ iff $v \xrightarrow{hb} w \bigvee w \xrightarrow{hb} v$. Extending from the above, we term $v$ the source vertex of an edge, and $w$ the sink of an edge.

Figure 1 illustrates how the $V_{src}$ and $V_{sink}$ behave with regard to their incoming and outgoing vertices. It shows how property 4.7.1.1 applies.

Event sets, as defined previously, are associated with each vertex in order to analyze events with HBA. Analysis involves comparing events per vertex, taking into account its reachability, considering its context in the larger execution flow.

Given Definition 2.7, the vertex-edge event mapping in a FJG can be formalized as in Definition **??** below:

Based on the above, Definition 2.8 formalizes the HB relation in the context of a FJG:

**Definition 2.8** (Happens-Before Relation on FJG)**.** Let $d_1, d_2 \in D$ be two edges in the Fork-Join Graph $F$, defined as tuples $d_1 = (u_1, e_1, v_1)$ and $d_2 = (u_2, e_2, v_2)$.The event $e_1$ is said to **happen-before** $\epsilon_2$, denoted $e_1 \xrightarrow{hb} e_2$, iff there exists a directed path from the destination vertex of the first edge to the source vertex of the second edge. Formally:

$$v_1 \xrightarrow{F} u_2$$

Where $\xrightarrow{F}$ denotes reachability within the graph $F$. If $v_1 = u_2$, the relation holds immediately.

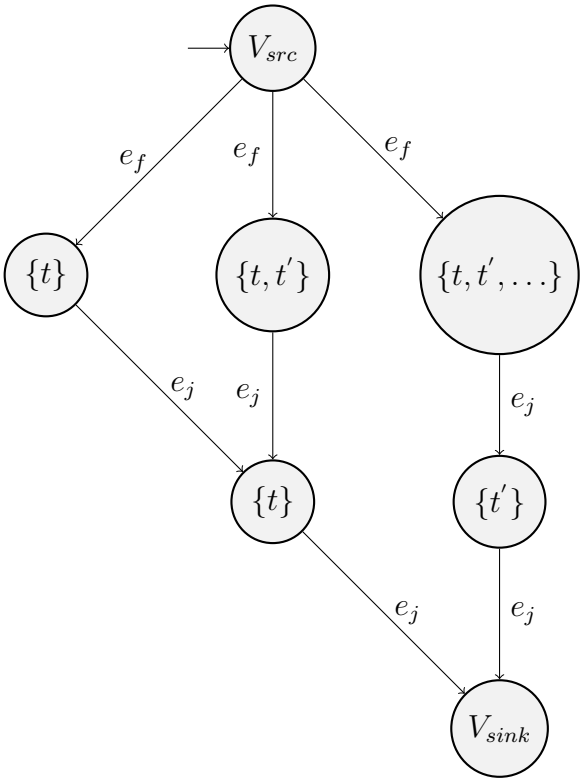



Figure 1: Example of a FJG showing $V_{src}$ and $V_{sink}$ with thread creation ($e_f$) and destruction ($e_j$) events labeled.

This relation can be derived due to the strong ordering on fork operations i.e. no event's thread comes into existence before another event can have free ordering with that event. The same relation can be derived, therefore, by relying on the strong ordering implied by join operations i.e. no event that happens after a given event's thread join operations, can have free ordering with that event. This is because an event cannot happen after it's host thread joins/exists.

Having obtained a means of establishing the partial ordering of events in Definition 2.8, it becomes possible to derive an expression for raceable access by negation, as in Definition 2.9 below:

**Definition 2.9** (Fork-Join Race Condition)**.** Events $e, e^{'} \in E$, as associated to edges $d$ and $d^{'}$ can be said to race if the following is true in a FJG:

$$\neg\Big(e^{'} \xrightarrow{\gamma} e\Big) \bigwedge \neg\Big(e \xrightarrow{\gamma} e^{'}\Big) \bigwedge \neg\Big(\tau(e) \cap \tau(e^{'}) = \emptyset\Big)$$

$$\longrightarrow \neg\Big(e \xrightarrow{hb} e^{'} \vee e^{'} \xrightarrow{hb} e\Big)$$

Thus far, this section has defined three different race condition criteria, specifically those based on LSA, HBA and FJGs. These criteria define races at varying levels of granularity, with LSA being the least granular in the sense that it is only aware of Lock-Set disciplines. These criteria can be combined into a single hybrid criterion for race detection in order to achieve functionally complete and sound race detection, as in Definition 2.10 below:

**Definition 2.10** (Hybrid Race Condition)**.** Two events $e, e^{'} \in E$ can be said to be co-engaged in a hybrid race, written $e||^{hb}e^{'}$, iff it can be shown that:

$$\neg\Big(e^{'} \xrightarrow{\gamma} e\Big) \bigwedge \neg\Big(e \xrightarrow{\gamma} e^{'}\Big) \bigwedge \neg\Big(\tau(e) \cap \tau(e^{'}) = \emptyset\Big) \bigwedge$$
$$\Big(L(\tau(e), e) \cap L(\tau(e^{'}), e^{'}) = \emptyset\Big)$$

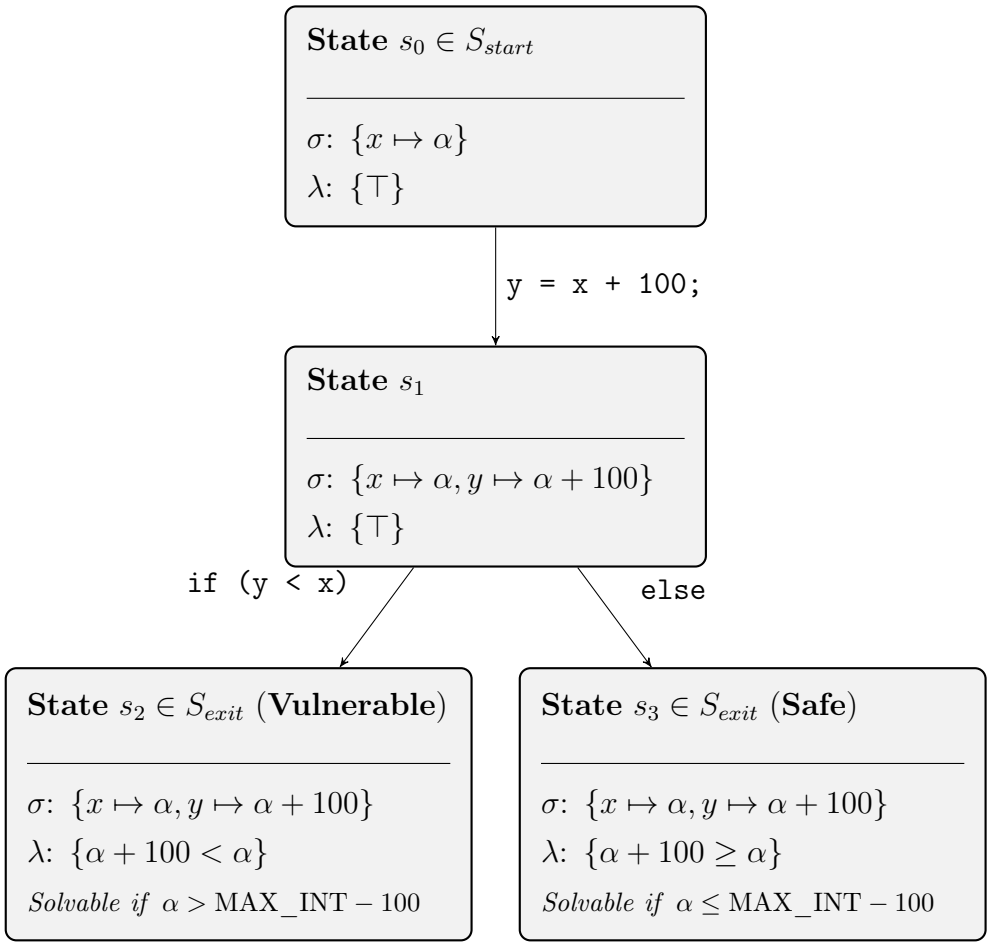
The inclusion of lock-sets makes it possible to carry out a number of key optimizations to race detection, especially given the nature of FJGs in specific, and DAGs in general. Approaches in the literature have resulted in partitioning CFGs and other execution representative data structures into raceable zones, i.e. zones where no races are possible according to vector clock value and FJG properties. The optimization allows approaches to add an additional criterion to steer execution away from redundant paths that do not impact concurrency during fuzzing or symbolic analysis.

HBA requires a significant overhead to implement, as well as complex reasoning and instrumentation, which means implementations of the technique tend to either be slow but accurate, or fast but imprecise. Accordingly, research studies [12, 13] have proposed various means of improving the various underlying sub-components of HBA.

LSA operates efficiently with minimal overhead by primarily monitoring significant lock events. However, its scope is limited to identifying races resulting from inconsistent or ineffective utilization of locks. Due to the potentially lower overhead of LSA compared to more resource-intensive methods like HBA, research efforts have explored combining the two approaches, resulting in the development of hybrid HBA-LSA approaches [13, 14, 15], which reports races by leveraging both techniques. Essentially, the hybrid method relies on HBA only when LSA has been applied and indicates potentially raceable access.

The coming section will discuss the background theory required for Symbolic Execution employed by the approach documented in this paper.

### *2.4. Symbolic Execution*




SE is a static analysis method for analyzing programs under symbolic, instead of concrete, inputs. The technique allows for automated generation of concrete test cases and enables a wider range of theoretical analysis when compared to methods driven by concrete execution or simpler static analysis alone [16].

Symbolic engines convert program interaction under given symbolic inputs into representations as boolean constraints, which are consumable by SAT/SMT solvers. SAT/SMT solvers employ constraint-solving techniques

in facilitating proof of reachability of a given state under the analyzed input. This not only means producing proof of reachability, but also results in a concrete test case being available due to the nature of constraint solvers, namely approximating the literal input required to show Satisfiability.

SE although allowing for powerful and comprehensive analysis, however, has complexities that can make its practical application challenging. A number of sources of complexity and overhead exist some of which are summarized in the following list:

- *Constraint Solving*: SE engines rely on the work of constraint solvers to determine the outcome of a given execution path. Just as in any analysis, constraint solvers and SMT solvers are vulnerable to being overwhelmed by an explosion of constraint solver terms to solve for [17], or may encounter non-linear arithmetic which greatly slows down computation. Angr addresses this by using Claripy, its internal SMT-level manager, which provides a unified interface for various solvers, with a primary focus on Z3. Angr also offers plugins and built-in optimizations to speed up the constraint solving process by applying concretization strategies. Other approaches have attempted load-balancing between different solver engines, caching [18] or re-using results from previously solved constraints.

- *State Explosion*: The number of execution paths, and therefore also the number of symbolic states, typically undergoes significant growth as a program is analyzed; the number of paths at least doubles for every conditional branch encountered. Consequently, symbolic analysis may result in a very large number of states that then require an overwhelming amount of computation to derive decisions from [**?** 19, 20]. Angr attempts to combat state explosion through several built-in mechanisms. *SimProcedures* which replace funciton calls and other literal execution, as well as Exploration Techniques and are also key, as they model common library functions to avoid exploring their internal logic, thus pruning the state space. Although none of these are tested on COTS binaries in any extensive academic reporting, which means their effectiveness in real world scenarios is largely under studied.

- *Memory Modeling*: Due to the amount of computation and impact on the number of program states that depend on memory values and com-

plex pointer arithmetic[2], many contemporary frameworks [22, 19] provide very complex memory models which sometimes carry with them an overhead of storage, disk space and computational load. Angr's memory model, known as *SimMem*, is a core component that handles symbolic and concrete memory operations. It abstracts memory and provides efficient representations for symbolic pointers and arrays, which is crucial for analyzing vulnerabilities. Despite this, its complexity can contribute to overhead, impacting overall performance [23, 24] and resilience to state explosion.

Due to the state explosion and other problems mentioned, research focused on the application of SE often employs a number of strategies to prune down data structures and code representations. Pruning down is often done in order to focus the algorithm on only those areas of the program covering points of interest to a specific kind of analysis, e.g. vulnerability assessment, run-time memory usage, or program correctness, while achieving as complete, sound, and reproducible an analysis as possible.

In order to provide clear terms of reference for the aspects of symbolic states and wider execution we introduce a few simple formal definitions.

**Definition 2.11** (Symbolic State)**.** We define $s$ a symbolic state such that $s \in \mathbb{S}$ the set of all symbolic states. Each symbolic state is defined as am ordered set $s = \{\sigma, \lambda, s_{pred.}, s_{succ.}\}$, where (i) $\sigma$ is the symbolic store, which holds which variables have been symbolized or involved/targeted in constraint solving steps, (ii) $\lambda$ is the set of constraints required to evaluate true in order to reach $s$ in the Symbolic Map[3](iii) $s_{pred.}$ is the set of symbolic states that precede $s$ in the symbolic map. And finally, $s_{succ.}$ a set of states that succeed $s$ in the symbolic map.

In order to show reachability between symbolic states we are required to relate them in a directed graph.

**Definition 2.12** (State Map)**.** We define $S_{map}$ a symbolic state map to be a Directed Graph (DG) which is an ordered set follows: $S_{map} = \{V, D, S_{start}, S_{exit}\}$

[2]This is especially important for security analysis due to the noted dependence between complex pointer arithmetic, buffer overflows and other memory corruption vulnerabilities [21]

[3]defined later in 2.12

where (i) $V$ is the set of vertices which is a set of symbolic states such that $V \subseteq \mathbb{S}$, (ii) $D \subseteq V \times V$ the set of edges which is of the following form $D = \{(S_i, S_j) \| S_i, S_j \in V \text{ and } S_i \longrightarrow S_j\}$ and (iii) $S_{start}, S_{exit}$ are defined as $S_{start} \in V$ and $S_{exit} \subset V$ is the start state, and the set of states that exist or terminate in execution, respectively. (iv) We say there is an edge between two states $S_i, S_j$, and that they are "connected" which is written as: $i \rightarrow j$ iff the symbolic state $j$ is connected to $i$. Namely we can say $j \in i_{succ.} \wedge i \in j_{pred.}$

Symbolic states may be connected i.e. find themselves members of each one another's successor and predecessor sets, $S_{succ.}$ and $S_{prod.}$ accordingly, but may not be "reachable". That is, in symbolic analysis they may be connected via return or call statements, but not reachable in real execution. We define reachability as a factor of whether the constraints associated to a *Symbolic Path* can be solved. Since it indicates that an existing, enumerable number can be used to fulfill the constraints.

XEGMAP makes use of Angr's SimState plugins in order to implement an inheritable bookkeeping which is transferred to successor states in the state map on forking and branching so that everything recorded about concurrent events is carried forward to each state derived from a symbolic execution step. Memory accesses which involve thread accessible memory, namely GMAPS, are recorded with bookkeeping information so as to allow for HBA.

In order to represent paths over a given Symbolic Map, we define Symbolic Paths as follows:

**Definition 2.13** (Symbolic Path and Reachability)**.** We define a symbolic path $\pi$ of length $k$ as an ordered sequence of states $\pi = \langle s_0, s_1, \ldots, s_k \rangle$.A state $s_k$ is said to be **reachable** from $s_0$ along $\pi$ iff the following conditions hold:

1. **Connectivity:** For all $0 \leq i < k$, there exists an edge connecting the states such that $(s_i, s_{i+1}) \in D$.

2. **Constraint Solvability:** Let $K(e)$ be the constraints associated with edge $e$. The logical meet (conjunction) of all constraints along the path is satisfiable: $\bigwedge_{i=0}^{k-1} K(s_i, s_{i+1}) \not\equiv \bot$.

3. **Resource Bound ($\Pi$):** The accumulation of variable stores along $\pi$ does not exceed $\Pi$, the upper limit of available memory.

4. **Time Bound ($\Theta$):** The total time required to solve the constraints along $\pi$ does not exceed $\Theta$, the upper limit of execution time.

Although seemingly a standard directed graph with simple connectivity rules, "reachability" implies that a machine can execute all of the aspects specified in the symbolic states along the path, this implies then that a solution to the constraints for all the states along the path can be found. Namely we say the collection of constraints as a total set of all the boolean formulas can all be determined to have solutions in a reasonable amount of time requiring a reasonable amount of resources. The definition although only for the purposes of our terminology in this paper, does offer the ability to host symbolic maps that have partially executable and partially connected paths. The difference between the two is crucial and models how symbolic engines over-approximate the reachability of states, before interrogating the constraint solver. The last few definitions expound a formal summarization of a definition for Symbolic State Exploration Step Functions, which partially models how we interact with the Angr symbolic execution session.

**Definition 2.14** (Symbolic State Exploration Step Function)**.** We define the Symbolic State Exploration Step Function $\Phi : V \to 2^V$ as a mapping that accepts a symbolic state $s \in V$ and returns the set of all immediate successor states defined by the edges in $S_{map}$. Formally:

$$\Phi(s) = \{s' \in V \mid (s, s') \in D\}$$

This function generates the next set of reachable states by traversing the edges connected to $s$.

Additionally, using this definition, we can derive proofs and algorithms or selection schemes that posit reachability across a path that is connected, and we can prune states that are connected in order to prioritize constraint solving.

We make use of the Angr symbolic execution engine which models state maps analogous to the—$S_{map}$ defined above—as objects in Python, allowing us to interrogate the connectedness of states. Symbolic states are represented by *SimState* objects that can take plugins that form bookkeeping information and extensions of state behaviour. We make use of SimStates in order to carry bookkeeping information about concurrent information forming a symbolic fork-join graph. Angr specifically allows API users to create a *Simulation Manager*—analogously represented by the $\phi$ function defined above—which steps through symbolic execution until no states are reachable or all have exited. Each simulation manager can be extended in its symbolic exploration

by *Exploration Techniques* such as *Depth First Search* (DFS) / *Breadth First Search* (BFS), *Director* which is the directed execution mode that can target symbolic states further down in the execution path or CFG.

XEGMAP employs a path selection scheme that prioritizes program points that interact with shared or thread-global memory. In later sections we will show this selection strategies greatly enhances relevant executions paths and mitigates state explosion that deters compatible analysis.

### *2.5. Binary Analysis*

Binary formats are documents which contain the last textual description of a program before it is instantiated as a running process. Binary formats are thus also commonly termed "executable" formats due to this. Analysis which targets executables therefore benefits from having a very literal description of process behaviour and may allow analysis to focus on real behaviour instead of over-approximation. Due to the complex transformation program definitions undergo during compilation into executable formats; a lot of the semantic information embedded in code is lost. Semantic information such as type declarations, function names, and parameter specifications among others. Hence the presence of the SILP, which—counter to some of the benefits of binary analysis—tends to force binary analysis methods to approximate metadata.

Frameworks that handle interpretation and monitoring of binary programs are required to achieve sophisticated *Dynamic Binary Analysis* (DBA) of programs in their executable/binary formats even the midst of problems like the SILP. For this reason, many efforts in research present *Dynamic Binary Instrumentation* (DBI) frameworks and *Binary Translation Frameworks* (BTF), sometimes referred to as binary "re-writing" frameworks.

The analysis of code in binary form requires the code to undergo several conversion processes to incorporate the side effects of instructions on memory and program state [25]. The goal of the binary translation/analysis processes is to regenerate the behavioral and some semantic details embedded into the executable by the compiler. The process typically involves [26]: (i) *Disassembly*, converting code in executable form to the appropriate assembler format (i.e., `x86_64`, `MIPS32`, etc.); (ii) *Code Lifting/IR Generation*, converting assembler into an *Intermediate Representation* (IR) [27] such as CFGs and Call Chains, or into an *Intermediate Language* (IL) that captures all the side effects that assembler instructions have (e.g., FLAGS registers, cache

state, indirect jumps and others); and (iii) *IR analysis*, which involves analysis of IR properties, structure and emulation of execution via SE or even via recompilation into a binary termed *Binary Re-Writing*. Carrying out these processes is vital to achieving comprehensive, sound and complete interpretation of the binary. (iv) *Emulation and Execution* Here the framework will optionally, execute or emulate execution via a staged environment.

Although important, the previously mentioned processes have a strong dependence on the consistent and reproducible interpretation of code, which means that BA suffers from many pitfalls and challenges which commonly stem from the SILP, [26, 25] these include: (i) *Variable Aliasing*, where it can be immensely challenging to discern between overlapping instances of variables, determine variable live-ness and access scope; (ii) *Call Argument Identification*, as arguments to function calls and system calls provide tremendous detail in understanding program data flow; (iii) *Function Boundary Identification*, as determining when procedures begin/end in contiguous code, this allows analysis to reconstruct execution contents that may be allowed to happen concurrently; (iv)*External Dependencies and Late Link Resolution*, where binaries may depend on external libraries; (v) *Control Flow Graph Recovery*, as its accuracy is critical for analysis techniques but complexity introduced by indirect function calls and jump/branching semantics can render recovery challenging or even infeasible; and *Instruction Set Architecture* (ISA) Complexity, as the complexity of the ISA contributes quite significantly to the integrity of BA implementations, with ISAs coming in many forms, some supporting multiple encoding and execution modes, and all having important side effects as regards computation and execution flow.

Managing these pitfalls means relying on novel techniques to recreate or approximate semantic information lost during compilation. A novel example of managing this source of approximation is seen in a technique called *Value Set Analysis* which approximates values to be assigned to a given memory cell based on the range of references it assumes during execution [28]. The method allows binary analysis approaches to issues a range of guesses for things like function pointers which may greatly impact control flow.

## 3. Literature Review

This section discusses and details prominent Binary Symbolic Race Detectors (BSRDs) specifically those that make use of the Angr framework.

*COMRace* [5] is a binary analysis-driven race detector for Windows, focused on detecting race conditions in binary implementations of the Microsoft Common Object Model (COM) Interfaces. COMRace performs static analysis using Python IDA and Angr [19] to decompile binaries for analysis. It only performs LSA which limits its ability to detect complex races that require thread awareness i.e. awareness of which threads are alive/dead during race events. It was evaluated by running it on a Free and Open Source version of Windows called ReactOS, and was found to be able to successfully analyze 92.1% of all registered COM Objects, as well as uncover 26 previously undiscovered vulnerabilities. COMRace operates on binaries, making it BFF, but it does not explicitly utilize SE improvements or contribute any novel SSEA in Angr. Additionally COMRace and does not do any contemporary forms of HBA or thread IE in its race detection approach, limiting its effectiveness. Due to the way it places reliance on Angr; COMRace is however BFF and relies on SE making it the only known study in our sample that has all three attributes, namely being BFF, COTS and symbolic. As per our reading, the currently cited study on COMRace does not report any comprehensive ground truth verification metrics.

*HardRace* [4] is a post-mortem style Hardware Feature based HB/LockSet race detector built on the Angr [19] binary SE framework. HardRace leverages the Intel proprietary PTWRITE instruction in order to track memory internactions. Additionally it introduces a novel form of Value Set Analysis (VSA) which incorporates some awareness of concurrent thread activity. The VSA produces a set of locations $T_{mayRace}$ which is essentially the naive set of a-locs over shared memory named, $T_{shared}$ but pruned of any locations that have been determined not to be raceable. i.e. doesn't involve any thread-accessible memory and is race-able under LockSet Analysis. $T_{mayRace}$ instructions are instrumented with the Intel PWRITE instruction which targets a given register. Instrumented instructions produce a trace of thread accesses which is then analyzed using an offline HB/Lockset algorithm sourced from FastTrack. Evaluation worked by comparing HardRace to other Hardware-Driven race detectors namely Kard [29] and ProRace [3], over a dataset that consisted of samples of the PARSEC benchmark and 7 COTS binaries with known race or "pre-baked"[4] conditions, including: *mysql*,*apache*, *cherokee*, *pbzip* and *aget*. Results showed that HardRace could reliably detect races in

[4]artificially added for the sake of testing

all the sample cases with a probability of 100% compared to ProRace which only achieved 100% detection in 2 of the test cases and Kard which only achieved 100% in 6 cases. HardRace works on the binary as an input which makes it BFF, leverages SE although reportedly only during CFG construction (not for full binary emulation/execution) and is evaluated on a sample of COTS binaries.

*BinTSAN* [1], is an effort to provide the same race detection capability as seen in ThreadSanitzer [30] to binary COTS programs. The study also unpacks the difficulties in porting other popular bug oracle sanitizers like Undefined Behaviour Sanitizer to binary only analysis. BinTSAN makes use of binary instrumentation framework called ZIPR [31] to build a bug oracle which reports race conditions. BinTSAN was evaluated in comparison to TSAN and Helgrind, on 41 binaries, namely: (i) 10 from the popular PARSEC version 3[5], (ii) 7 from SPEC CPU 2017 version 1.1.0 datasets and (iii) 24 COTS programs. Results showed that additionally BinTSAN reported: (i) the highest false-positive rate (ii) the highest false-negative rate and (iii) runtime overhead was around 20.7x (which was 1.15x the overhead of TSAN). Authors attribute the high false-negative and false-positive rates to compiler optimizations which omit certain function calls from the call stack. The approach relies purely on the executable binary as input without any prior knowledge of its structure or dependencies and thus can be classified a BFF approach with evaluation on COTS binaries. BinTSAN does not reportedly make use of any comprehensive form of SE. Additionally we consider BinTSAN ground-truth verified.

In the previous section we review a sample of 4 of the most prominent contemporary automated binary race detection approaches which make use of Angr as a symbolic execution framework. We believe that these constitute the summary of approaches relevant for comparison to XEGMAP as they use Binary friendly, ground truth verified approaches that are evaluated for performance against COTS binaries. Although the full spectrum of performance metrics are not available in the cited studies, we find enough to compare with XEGMAP in terms of block coverage and ground truth based performance.

The next section provides an overview XEGMAP and the decisions behind its design.

[5]specifically programs pertaining to multi-threaded behaviour

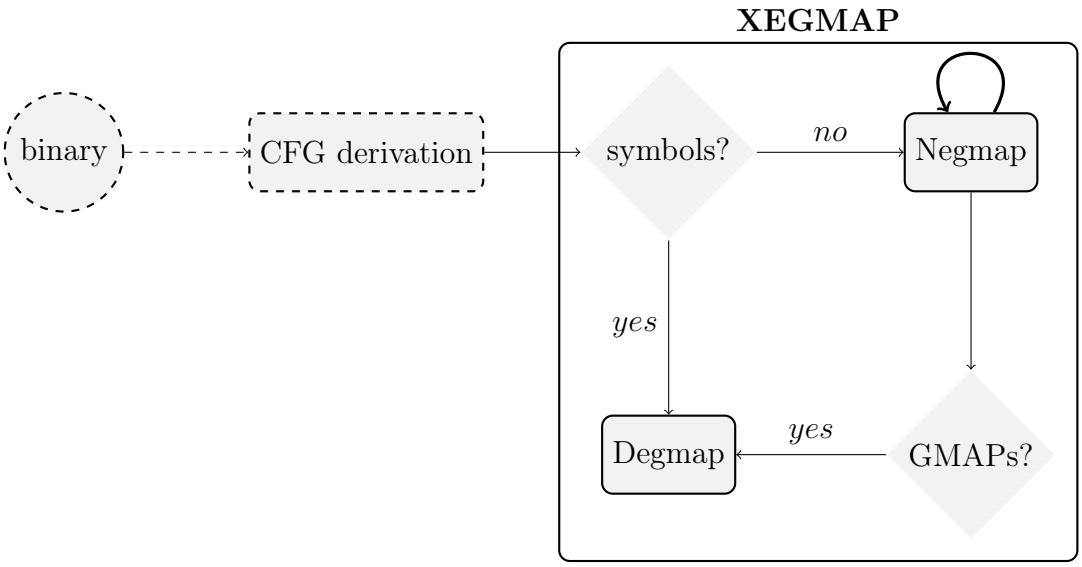



Figure 2: Overview of the proposed race detection approach "XEGMAP". A binary input is processed to obtain a CFG; while not shown here for brevity, this is done in Angr as per the process in Figure **??**. The resulting CFG is used by XEGMAP. XEGMAP first determines whether symbol and other information are available; if they are, Degmap is invoked; if not, Negmap is invoked. If Negmap is invoked, and if at any time global memory access points are encountered, processing immediately switches to the more efficient Degmap using the new information.

## 4. XEGMAP Overview

XEGMAP or "X"-ecution for Global Memory Address Points (GMAPs), so named because the "X" represents both the "exe" syllable in the word execution and that it can be operating in either Naive Execution mode called "NEGMAP" or Directed execution mode called "DEGMAP". Naive mode being the mode that it starts in when there is a lack of symbolic information for guiding exploration of symbolic state space. Once it encounters a GMAP, it switches to directed mode using this information to structure the rest of the state space exploration.

Referring to Figure 2, XEGMAP invokes Degmap when the input binary is found to contain relevant symbol information at the start, this is either the presence of a GMAP or symbols for the relevant set of pthread calls. Degmap accordingly prioritizes paths in execution which lead to concurrent activity in order to provide effective optimization in managing state explosion, while discovering potentially raceable accesses. This includes memory

access instructions for program areas with global access permissions i.e. areas in which all threads can interact with memory locations at any time. Given the presence of guidance information, this is a significantly more capable process than Negmap.

If no symbol information is located at the start, XEGMAP reverts to Negmap. Negmap is a modified Depth-First exploration algorithm that prioritizes states that contribute to higher block count and call stack depths. Negmap immediately switches to Degmap once a GMAP is discovered, and once Degmap is invoked this way it treats all previously discovered GMAPs with a negative polarity allowing it to avoid reanalyzing symbolic states that do not contribute new GMAP information. This optimization also naturally contributes to avoidance of loops.

Negmap and Degmap are implemented as Exploration Techniques in the Angr framework. In order to track memory operations, the Angr framework offers an API with "hooks" that trigger whenever any read or write to memory is performed, XEGMAP uses these hooks to inspect the target of the memory read/write and determine if they are in a globally addressable memory space. XEGMAP makes use of Angr's Simprocedure functionality in order to track pthread call interaction. This intercepts and overrides calls encountered in the binary to a targeted function, but only if one is aware of the symbol name or function address. Making use of all of this information requires Xemgap to construct a happens-before aware data structure and track events relevant to concurrent interaction as well as book keep interaction with GMAPs. The following section provides more detail as to these aspects of the approach: Sections 5.1 and 5.2 detail the Negmap and Degmap algorithms, respectively.

## 5. Implementation

We've implemented XEGMAP using Angr version 9.2.8 and ran our experiments on a Ubuntu version 20.04.6 LTS running Linux Kernel version 5.4.

The following section details Negmap the naive exploration component of XEGMAP which runs first before any semantic or symbol information is discovered.

*5.1. Naive Symbolic Exploration with Negmap*

While DSE provides significant improvement to the state growth and efficiency of many SE implementations, it requires knowledge of the structure of the binary under analysis, as well as target points to steer execution towards. In the absence of this information, symbolic exploration proves to be entirely naive, particularly with regard to awareness of thread activity i.e. detecting and modelling concurrent behaviour.

The proposed approach to naive symbolic exploration in Negmap uses a modified version of Depth First Search Exploration (DFS) in order to overcome the aforementioned limitation. DFS is modified in the following ways:

- **Variable Keep *Alive* States**: The approach allows for a configurable number of states to be simultaneously maintained during execution. Once the number of possible tracked states—called *alive* states—exceeds the configured limit, the proposed algorithm randomly selects a subset of the currently *alive* states to maintain and follow during execution. The modification allows XEGMAP users and researches to adapt the algorithm to resource requirements limiting XEGMAP to lower actively executing states when memory pressure and demands are higher.

- **Deferred State Recycling**: The proposed approach recycles deferred states—states that are de-prioritized in execution and stored for later analysis. Furthermore, this modification allows XEGMAP to conitnue execution in states that were previously abandoned in analysis.

- **Code Block and Call Stack Depth prioritization** Negmap prioritizes states in path selection when they are shown to either increase block count or call stack depth, this allows Negmap to uncover new and more complex interaction, potentially increasing the chance of encountering a GMAP or pthread operations.

Algorithm 1 details the implementation of Negmap (Naive Execution for Global Memory Access Point Discovery). The algorithm operates over the Symbolic State Map $S_{map} = (V, D, s_{start}, S_{exit})$. Execution begins by initializing a Worklist $\mathbb{W}$ with the start state $s_{start}$.In each iteration, Negmap retrieves a state $s$ from $\mathbb{W}$ and inspects its associated instructions using the mapping Events($s$), which returns the set of events $\mathbb{E}$ occurring within that symbolic state. The algorithm checks each event $e \in \mathbb{E}$ to determine

if it involves a memory read or write operation—NEGMAP checks only for GMAPs here since there may not be enough information to check for thread interactive calls like locks, start routines etc. If an operation is detected, it further verifies if the target memory address $m$ resides in Global Memory (indicating a potential shared resource).If a Global Memory Access Point (GMAP) is identified, Negmap records the access point and associated event as an ordered tuple $(e, m, p)$ into the set $\Gamma$ and invokes the Degmap procedure (Directed Execution), passing the current state and discovered memory context to analyze thread interactions and potential race conditions.To optimize state exploration, Negmap employs a heuristic pruning strategy before generating successor states. This is defined by two evaluation functions: (i) $\Delta(s) \rightarrow \{True, False\}$, which determines if the call stack depth of state $s$ exceeds the current known maximum; and (ii) $\beta(s) \rightarrow \{True, False\}$, which checks if the block count exceeds the known maximum. If both return False (indicating low coverage contribution), the state is deprioritized with a 40% probability to prevent resource exhaustion on unpromising paths. Finally, if the state is deemed viable, Negmap utilizes the Symbolic State Exploration Step Function $\Phi(s)$ to generate the set of successor states $S_{next}$. These successors are added to $\mathbb{W}$ for future processing.

Negmap was developed through an iterative process of experimentation on COTS binaries from a contemporary Linux x86_64 OS, Ubuntu 20.04. During experimentation, it was found that the proposed Negmap implementation, including the proposed optimizations, provided resilience to many complex constructs in modern binaries and showed a marked performance improvement over naive approaches in the Angr Framework during evaluation.

### *5.2. Directed Symbolic Exploration with Degmap*

Degmap is shares the same pthread and memory read/write checks as Negmap, it is unique in its polarization of states during exploration. Each time a state is to be advanced in execution, the address of the symbolic state is examined against the CFG to ascertain whether it traverses a block that includes an already detected memory/read or write operation, should it be in the list of detected memory states. Degmap avoids a state associated to a code block, which includes in its path, a return to a previously discovered state which has GMAP interaction with a known memory address. These states are de-prioritized in execution according to a heuristic based on probability. The polarization ensures that some states are more likely

to be avoided in execution, this is done to avoid loops and re-current structures in code that may trap the exploration and negatively affect the rate at which Degmap becomes aware of concurrent thread activity and additionally avoids the overhead in constraint solving and memory modelling imposed by the analysis of states, especially those that do not contribute new GMAP discovery.

The proposed Degmap implementation specifically aims to enhance the rate of race condition detection. The following aspects to code using the following adaptations and optimizations:

- *GMAP Discovery*: Interacting with memory artifacts that are shared across multiple threads is pivotal to the success of a race condition detection algorithm. Therefore, the proposed approach aims to increase the number of discovered GMAPs compared to other methods.

- *Code Coverage*: As with any static analysis, code coverage is an important factor that specifies the completeness of the inspection of code. The proposed approach shows its marked improvement in the depth of code it reaches, and the sparsity of code it is made aware of, over standard approaches built on Angr.

- *Symbolic Efficiency*: Due to the nature of this approach, effective management of factors such as the state storage growth rate, the possibility of symbolic state explosion and the complexity of constraints, and other factors, have a significant impact on the practicality and resilience to the complexity of COTS binaries. These are all areas in which the proposed implementation shows a marked improvement.

Furthermore, the proposed approach to DSE implemented in Degmap shows improvement in the speed of SE, is capable of detecting races with very little information or involving large state spaces, and is capable of continuing execution through conditions which typically overwhelm other comparable SE approaches.

Crucially, the proposed approach promises significantly increased resilience to the lack of symbol and other information in COTS binaries.

Algorithm 3 formalizes the Degmap (Directed Execution for GMAP Discovery) approach. The procedure is instantiated with the necessary context from the previous Negmap phase: the Symbolic State Map $S_{map}$, the history of discovered memory accesses $\Gamma$, and the current worklist of states $\mathbb{W}$. Once

execution is underway, Degmap applies a heuristic pruning strategy to the current state $s$. This corresponds to the check HeuristicCheck, which analyzes the Control Flow Graph (CFG) to determine if the current path is likely to reach a relevant Global Memory Access Point (GMAP) within a specific horizon (e.g., 100 edges). If the state is determined to be non-contributory (or "negatively polarized"), it is pruned from execution to conserve resources.If the state is retained, Degmap inspects the events $\mathbb{E}$ derived from the state. It first handles thread and synchronization primitives to maintain the execution context:(i) Thread Creation: If an event satisfies IsThreadCreate, the active thread set $\mathbb{T}(s)$ is updated to include the new thread ID.(ii) Thread Join: If an event satisfies IsThreadJoin, the target thread is removed from $\mathbb{T}(s)$.(iii) Locking: If an event satisfies IsMutexLock or IsMutexUnlock, the lock set $\mathbb{L}(s)$ is updated to reflect the currently held mutexes.Note that in this formalization, the state $s$ implicitly inherits the $\mathbb{T}$ and $\mathbb{L}$ context from its parent state (a feature typically handled by the underlying symbolic execution engine, such as Angr).Finally, Degmap analyzes memory operations. If an event involves a read or write to a global memory address $m$ with size $p$, the algorithm compares this access against the history $\Gamma$ candidates labeled $m_c, p_c$ to detect potential race conditions. A race is reported if a prior access $(e_c, m_c, p_c) \in \Gamma$ satisfies three conditions:

- Concurrency: The thread responsible for the prior access is distinct from the current thread: $\mathbb{T}(e_c) \neq \mathbb{T}(e)$.
- Lack of Synchronization: The intersection of the lock sets held during the two accesses is empty: $\mathbb{L}(e_c) \cap \mathbb{L}(e) = \emptyset$.
- Memory Overlap: The memory intervals overlap, defined formally as:

$$[m_c, m_c + p_c) \cap [m, m + p) \neq \emptyset$$

  If these conditions are met, ReportRace is triggered.

Following analysis, the current access is added to $\Gamma$, and the successor states are generated via the step function $\Phi(s)$ and added to the worklist.

## 6. Evaluation

This section details the results obtained during experimentation, along with relevant analyses. Sections 6.1–6.3 below, respectively, discuss the results and corresponding analyses on the UBN11-noraces-cots, CUST5-races-nosem and OSL21-noraces datasets.

Table 1: Cross-study comparison of XEGMAP against other contemporary BSRDs. "Detection Style (RCDS)" compares how race detection interacts with memory, namely if memory inspection happens before execution we term it Compile-Time; during execution as On-The-Fly or after execution as Post-Mortem. "Concurrent Analysis" lists the strategy used to account for concurrent events being either Happens-Before (HBA), Lock-Set Analysis (LSA) or hybird HBA/LSA. "Binary-Friendly" shows whether the analysis method takes in binaries as an input material requiring no input samples or specialized environment. "False Positive Rate (FPR)" shows the currently attested/reported rate of false positive detection. "Runtime Overhead" shows the tested and reported overhead imposed by instrumentation and inspection/analysis steps incurred by the listed approach.

| **Metric** | **XEGMAP** (Ours) | **HardRace** [2024] | **COMRace** [2022] | **BinTSAN** [2024] |
|---|---|---|---|---|
| **Detection Style (RCDS)** | Compile-Time | Post-Mortem | Compile-Time | On-The-Fly |
| **Concurrent Analysis** | LSA/HBA | LSA/HBA | LSA | LSA/HBA |
| **Binary-Friendly (BFF)** | Yes | Yes | Yes | Yes |
| **False Positive Rate (FPR)** | **0%** (CUST5) | Negligible | 31.4% (ReactOS) | High (Reported) |
| **Race Detection (TPR)** | 100% (CUST5) | 100% (PARSEC) | 194/256 PoCs | 117 (Identical) |
| **Runtime Overhead** | High (Symbolic) | **1.6%** | Moderate* | 20.7x |

### *6.1. Results on the UBN11-noraces-cots dataset*

The complete table of results for this experiment are provided in Table 2. In each row, and for each metric category, the best performing package between XEGMAP, Stochastic and Loopseer has been highlighted in the table.

What is immediately apparent from the table is that no race conditions were detected in any of the binaries, as expected and previously explained. Binaries in the open source community are routinely maintained and updated to remove all known vulnerabilities by a large and vibrant community of open source. While latent vulnerabilities in these binaries are believed to exist, they are believed to be associated with very rare execution paths and conditions, which are not easily encountered.

Crucially, it is observed in the GMAP results that XEGMAP out-performs Stochastic and Loopseer in terms of the number of GMAPs discovered in the vast majority of the binaries—9 of the 11 binaries. The following key findings are made:

1. For 5 of the 11 binaries, Stochastic and Loopseer are not able to detect any GMAPs, whereas XEGMAP successfully locates several of these crucial structures in the binaries.

2. In general, XEGMAP significantly out-performs Stochastic and Loopseer in terms of the number of GMAPs discovered in almost every case.

| Binary | Race Detected | | | GMAPs | | | Call Stack Depth | | | Code Blocks | | | Avg. Step Time (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS |
| rfcomm | N | N | N | 1 | 13 | 8 | 2 | 13 | 9 | 9 | 3068 | 93 | 0.00062 | 0.0058 | 0.026 |
| xedit | N | N | N | 360 | 0 | 0 | 4 | 16 | 16 | 396 | 4096 | 5000 | 0.23 | 0.0039 | 0.0061 |
| hostname | N | N | N | 3 | 0 | 0 | 2 | 13 | 4 | 24 | 843 | 5 | 0.0067 | 0.0297 | 0.000048 |
| macptopbm | N | N | N | 4 | 0 | 0 | 3 | 8 | 5 | 9721 | 307 | 21 | 0.1893 | 0.0956 | 0.000071 |
| bmptopnm | N | N | N | 12 | 0 | 0 | 6 | 11 | 5 | 9227 | 624 | 27 | 0.44 | 0.0457 | 0.000063 |
| bluetoothctl | N | N | N | 82 | 12 | 0 | 4 | 16 | 5 | 36755 | 2101 | 9 | 0.27 | 0.0284 | 0.00014 |
| fstopgm | N | N | N | 1 | 0 | 0 | 2 | 9 | 5 | 241 | 561 | 23 | 0.0155 | 0.2079 | 0.000060 |
| dpkg-divert | N | N | N | – | 30 | 29 | – | 8 | 8 | – | 633 | 738 | – | 0.1295 | 2.6 |
| psresize | N | N | N | 49 | 48 | 2 | 3 | 12 | 6 | 49 | 1004 | 110 | 0.0042 | 0.0255 | 40.6 |
| sha1sum | N | N | N | 52 | 7 | 0 | 3 | 8 | 4 | 41 | 1093 | 6 | 0.041 | 0.0770 | 0.000051 |
| systemd-run | N | N | N | 85 | – | 48 | 3 | – | 9 | 83 | – | 240 | 0.029 | – | 2.44 |
| **Total** | | | | 649 | 110 | 87 | 32 | 114 | 76 | 56546 | 14330 | 6272 | | | |

Table 2: Summary of execution results of XEGMAP (XM), Stochastic (STC) and LoopSeer (LS) on the UBN11-noraces-cots dataset. The COTS binaries were compared in terms of the number of memory interactions with globally addressable variables ("GMAPs"), the number of logically contiguous blocks of code executed ("Code Blocks"), the depth of nested function calls in the current call stack ("Call Stack Depth") and the time in seconds that Angr took to complete a single SE step ("Avg. Step Time"). Note that the best performing package between XM, STC and LS for each binary (row) in each respective metric category column is indicated in yellow highlighting. Test cases marked with a "–" were not completed due to exceeding the execution time constraints in the experiment or due to a failure of Angr to successfully start up.

Overall, across all 11 binaries, XEGMAP detects close to 6 times more GMAPs than Stochastic, and close to 7.5 times more GMAPs than Loopseer.

3. While Stochastic and Loopseer detect GMAPs in some of the binaries, they don't detect any GMAPs in five of the binaries, whereas XEGMAP detects GMAPs consistently in all 10 binaries that it successfully executed on. This speaks to the consistency of XEGMAP.

4. XEGMAP was only outperformed by Stochastic and LoopSeer in GMAP discovery for the rfcomm binary. Stochastic's higher GMAP count can be attributed to its randomized path selection, which may occasionally appear to enhance GMAP discovery but does not reflect consistent performance across other binaries. LoopSeer likely performed better in this instance due to the presence of large loop structures within rfcomm, which Loopseer is specifically designed to analyse. Given rfcomm's role in maintaining network connections, it likely contains loops for tasks such as collecting network packets, monitoring disconnect events, and parsing commands and packet formats.

This is a very exciting result that clearly indicates the superior exploration effectiveness of XEGMAP as compared to Stochastic and Loopseer in COTS binaries.

XEGMAP was unable to execute and produce results on dpkg-divert. XEGMAP begins recording metrics from the first controllable symbolic execution step. However, during the CFG analysis, there is interaction with several components, including the SE engine and CLE, the binary loader that loads the binary into memory before execution. These interactions may fail or significantly slow down the analysis, as observed with dpkg-divert. CFG derivation is not entirely deterministic due to challenges such as complex function pointer arithmetic, intricate instruction formats, and variations in calling conventions, among other complexities.

While XEGMAP's clear superiority in exploration effectiveness over Stochastic and Loopseer has clearly been demonstrated, a further analysis of other metrics is carried out below.

- **Code Blocks Coverage**: XEGMAP uncovered a maximum of 36,755 code blocks during its execution, a figure typically associated with

fuzzing or concrete execution-driven approaches. This level of coverage underscores XEGMAP's thoroughness in exploring a binary's code paths. In comparison, Stochastic's maximum code block count was only 9,862, and LoopSeer uncovered 5,000 blocks at its best. This significant difference shows that XEGMAP not only performs better in terms of GMAP discovery but is also capable of exploring more extensive portions of code during execution, indicating its ability to handle complex binaries more effectively.

- **Consistency Across Binaries**: XEGMAP demonstrated impressive consistency in GMAP detection, finding GMAPs in every single run, regardless of the binary being analysed. Neither Stochastic nor LoopSeer can claim this level of performance. XEGMAP's ability to maintain performance across all binaries—without prior knowledge of their internal structures—highlights its robustness, even when facing COTS binaries.

- **Call Stack Depth**: While XEGMAP generally operates with lower call stack depths compared to Stochastic, it remains competitive with Loopseer. For example, in the bluetoothctl binary, XEGMAP reached a depth of 4 which competes with Loopseer. In context, considering that XEGMAP is more effective at discovering GMAPs, these lower depths can be considered to contribute to reducing the computational complexity during execution, especially in contrast to the high depths seen in some STC runs. Despite managing lower stack depths, XEGMAP still manages to uncover more GMAPs and code blocks, indicating a more efficient traversal of execution paths.

- **Average Step Time**: Stochastic generally outperformed XEGMAP in terms of average step time, with respective average step times of 0.065 and 0.123, although XEGMAP's step time is not high either. While Loopseer had the overall lowest average step times for specific binaries, such as hostname, macptobpm and bmptopnm, it also had the overall highest average step times for other binaries, such as dpkg-divert, systemmd-run and psresize, which demonstrates a lack of consistency in average step time. In general, XEGMAP's balance of speed and thorough exploration makes it more efficient for this specific task, as it uncovers more relevant states and code paths while maintaining very reasonable step times.

- **Performance Issues**: While XEGMAP generally excels, it faced some challenges with certain binaries. For example, it could not complete the 4096 execution steps for binaries like bmptopnm, bluetoothctl, and fstopgm. This was due to the complexity of the binaries and the aforementioned resource constraints. It is important to note that such issues are common across all evaluated approaches, as Stochastic and LoopSeer also struggled with similar resource consumption challenges on various binaries.

Based on the above analysis, it is clear that XEGMAP's ability to consistently find GMAPs in every run, paired with its extensive code block coverage, efficient call stack depth and low overall average step time positions it well above Stochastic and Loopseer.

### *6.2. Results on the CUST5-races-nosem dataset*

The complete table of results for this experiment are provided in Table 3. In each row, and for each metric category, the best performing package between XEGMAP, Stochastic, and Loopseer has been highlighted in the table.

Most notably, it is very encouraging to observe that XEGMAP detects races in all test cases. Further, it did so in less than 4096 SE steps, without prior knowledge of the binary or requiring source code. The result here is unprecedented for the Angr platform, given that no publicly available contribution to the Angr platform has been shown to enable race condition detection for programs in binary-only format. XEGMAP is therefore to date the only tool making use of Angr that has been shown to successfully detect races at different complexity levels. Additionally XEGMAP secures a place in the very rare category of race condition detectors which leverage symbolic execution and are binary format friendly.

A comparison of XEGMAP to Stochastic and Loopseer in terms of the number of GMAPs once again definitively shows the superiority of XEGMAP, where it obtained the most GMAPs on every binary, in fact at least double or more than its competitors, but also consistently detected GMAPs on every binary. In this case, Stochastic completely failed to detect a single GMAP on any of the binaries.

Furthermore, XEGMAP's superiority was also clear in terms of the code block coverage, where XEGMAP uncovered several times more blocks than Stochastic and Loopseer, showing that not only does it explore more of the

| Test | Race Detected | | | GMAPs | | | Call Stack Depth | | | Code Blocks | | | Avg. Step Time (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Case | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS |
| 1 | Y | N | N | 5 | 0 | 2 | 8 | 3 | 8 | 468 | 123 | 223 | 0.047 | 0.017 | 0.0002 |
| 2 | Y | N | N | 5 | 0 | 2 | 7 | 3 | 7 | 1023 | 122 | 222 | 0.050 | 0.016 | 0.0002 |
| 3 | Y | N | N | 5 | 0 | 2 | 8 | 3 | 8 | 1910 | 248 | 372 | 0.053 | 0.203 | 0.0004 |
| 4 | Y | N | N | 7 | 0 | 2 | 8 | 3 | 8 | 1001 | 258 | 382 | 0.021 | 0.011 | 0.0004 |
| 5 | Y | N | N | 13 | 0 | — | 8 | 3 | 0 | 2492 | 1158 | 0 | 0.054 | 0.011 | — |

Table 3: Summary of execution results of XEGMAP Stochastic and LoopSeer on the CUST5-races-nosem dataset. The binaries were compared in terms of the GMAPs, Code Blocks, Call Stack Depth Average Step Time. As before, the the best performing package is indicated in yellow highlighting and test cases marked with a "–" were not completed.

code than Stochastic and LoopSeer but it does so fruitfully while avoiding the constructs that delay or deter the compared approaches.

In terms of the call stack depth, XEGMAP and Loopseer came out on top and on par with each other. Finally, Loopseer was generally the most efficient approach but, with fewer GMAPS detected than XEGMAP, and with no races detected in any of the test cases, the efficiency is not matched by any effectiveness in the approach.

### *6.3. Results on the OSL21-noraces dataset*

This analysis of results is divided into two parts. First, a comparison of XEGMAP, Stochastic, and Loopseer on the OSL21-noraces-sem dataset is carried out, similar to the analyses in previous subsections. The aim is to further and finalize the evaluation of the exploration effectiveness of XEGMAP on this dataset, as compared to Stochastic and Loopseer. This is discussed in Sub-section 6.3.1 below.

Second, a comparison of XEGMAP's performance is carried out on the OSL21-noraces-sem dataset as compared to the OSL21-noraces-nosem dataset. The aim is to determine the extent to which the lack of semantic information affects the performance of XEGMAP. This is discussed in Sub-section 6.3.2 below.

#### *6.3.1. Comparison of XEGMAP, Stochastic and Loopseer on the OSL21-noraces-sem dataset*

Table 4 provides the execution results of XEGMAP, Stochastic and Loopseer on the OSL21-noraces-sem dataset. In each row, and for each metric category, the best performing package between XEGMAP, Stochastic and Loopseer has been highlighted in the table.

As with the UBN11-noraces-cots dataset, no races were detected by any of the packages, for the reasons previously explained.

A comparison of XEGMAP to Stochastic and Loopseer in terms of the number of GMAPs once again definitively shows the superiority of XEGMAP, where it consistently obtained the most GMAPs on all binaries that had GMAPs i.e. all but the binary "False", which is attributed to the lack of GMAP interaction in the false source code. Furthermore, it once again obtained one, and in some cases two, orders of magnitude more GMAPs than Stochastic. Loopseer performed particularly poorly on this dataset, where it failed to detect any GMAPs on all but one of the binaries. It is also im-

| Test Case | Race Detected | | | GMAPs | | | Call Stack Depth | | | Code Blocks | | | Avg. Step Time (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS | XM | STC | LS |
| dd | 0 | 0 | 0 | 368 | 20 | 19 | 6 | 7 | 5 | 1266 | 1024 | 52 | 3.4e-1 | 4.9e-3 | 3.2e-4 |
| df | 0 | 0 | 0 | 103 | 16 | 0 | 3 | 7 | 5 | 43 | 850 | 7 | 2.5e-3 | 2.6e-3 | 1.2e-4 |
| dircolors | 0 | 0 | 0 | 36 | 17 | 0 | 3 | 8 | 5 | 41 | 1024 | 7 | 6.1e-4 | 3.3e-3 | 1.2e-4 |
| dirname | 0 | 0 | 0 | 18 | 3 | 0 | 3 | 8 | 5 | 39 | 1024 | 7 | 3.2e-4 | 6.8e-2 | 1.2e-4 |
| du | 0 | 0 | 0 | 94 | 4 | 0 | 3 | 12 | 5 | 4551 | 1024 | 7 | 2.0e-2 | 3.2e-3 | 1.2e-4 |
| echo | 0 | 0 | 0 | 11 | 4 | 0 | 3 | 7 | 5 | 225 | 665 | 7 | 1.0e-1 | 4.1e-3 | 1.3e-4 |
| bracket | 0 | 0 | 0 | 100 | 14 | 0 | 9 | 7 | 5 | 485 | 1024 | 7 | 7.2e-3 | 8.1e-3 | 1.2e-4 |
| env | 0 | 0 | 0 | 105 | 3 | 0 | 3 | 7 | 5 | 81 | 1024 | 7 | 2.2e-3 | 7.4e-3 | 1.2e-4 |
| expand | 0 | 0 | 0 | 31 | 8 | 0 | 3 | 8 | 5 | 40 | 1024 | 7 | 6.0e-4 | 3.9e-3 | 1.2e-4 |
| expr | 0 | 0 | 0 | 148 | 1 | 0 | 11 | 7 | 5 | 315 | 1024 | 7 | 2.1e-1 | 2.0e-3 | 1.2e-4 |
| factor | 0 | 0 | 0 | 21 | 17 | 0 | 3 | 8 | 5 | 52 | 1024 | 7 | 5.4e-5 | 2.9e-3 | 1.2e-4 |
| false | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 3 | 3 | 4 | 5 | 7 | 3.8e-5 | 1.7e-3 | 1.1e-4 |
| fmt | 0 | 0 | 0 | 49 | 7 | 0 | 5 | 8 | 5 | 95 | 993 | 7 | 2.7e-2 | 5.6e-3 | 1.3e-4 |
| fold | 0 | 0 | 0 | 26 | 2 | 0 | 3 | 8 | 5 | 41 | 1024 | 7 | 4.0e-4 | 1.7e-3 | 1.3e-4 |
| getlimits | 0 | 0 | 0 | 13 | 1 | 0 | 3 | 7 | 5 | 46 | 1024 | 7 | 7.8e-4 | 3.3e-3 | 1.2e-4 |
| ginstall | 0 | 0 | 0 | 66 | 15 | 0 | 3 | 10 | 5 | 60 | 1024 | 7 | 1.5e-3 | 7.1e-3 | 1.2e-4 |
| groups | 0 | 0 | 0 | 19 | 3 | 0 | 3 | 11 | 5 | 38 | 1024 | 7 | 4.2e-4 | 5.9e-1 | 1.2e-4 |
| head | 0 | 0 | 0 | 93 | 8 | 0 | 7 | 8 | 5 | 398 | 967 | 7 | 1.7e-1 | 1.4e-1 | 1.2e-4 |
| hostid | 0 | 0 | 0 | 14 | 1 | 0 | 3 | 7 | 5 | 46 | 1024 | 7 | 3.1e-4 | 2.8e-3 | 1.1e-4 |
| id | 0 | 0 | 0 | 45 | 2 | 0 | 3 | 12 | 5 | 40 | 1024 | 7 | 7.4e-4 | 5.2e-3 | 1.3e-4 |
| join | 0 | 0 | 0 | 108 | 12 | 0 | 3 | 7 | 5 | 58 | 1024 | 7 | 1.7e-3 | 9.2e-2 | 1.3e-4 |

Table 4: Summary of execution results of XEGMAP Stochastic and LoopSeer on the OSL21-noraces-sem dataset. The binaries were compared in terms of the GMAPs, Code Blocks, Call Stack Depth Average Step Time. As before, the best performing package is indicated in yellow highlighting and test cases marked with a "–" were not completed.

portant to note that XEGMAP once again consistently detected GMAPs on every binary, except for "False".

Overall, in terms of call stack depth and code block coverage, Stochastic had higher metric values than both XEGMAP and Loopseer on this dataset, while Loopseer was, overall, once again the most efficient technique in terms of pure coverage metrics. However, given the significantly smaller, or even lack of, GMAPs detected by these techniques as compared to XEGMAP, these metric values are of little to no significance, and rather indicate ineffective use of thorough or efficient computation.

Of interest is XEGMAP's code block coverage on the binary "du" which is extremely impressive, since it is on the level of concrete execution. This level of coverage once again underscores XEGMAP's thoroughness in exploring binary code paths.

#### *6.3.2. Comparison of XEGMAP on the OSL21-noraces-sem versus OSL21-noraces-nosem datasets*

Table 5 presents the execution results of XEGMAP on the OSL21-noraces dataset, both with and without symbol information, i.e., on the OSL21-noraces-sem and OSL21-noraces-nosem datasets. Each binary was "stripped" of its symbol information by using the strip binary program. For ease of reference in this section only, the OSL21-noraces-sem and OSL21-noraces-nosem datasets will be referred to with labels "Sem" and "NoSem". To facilitate comparison, each metric category includes a "$\Delta$" column showing the difference in results between the Sem and NoSem datasets, allowing for a clear assessment of the impact of semantic information on XEGMAP's performance. Note that positive $\Delta$ values indicate a higher value for Sem while negative $\Delta$ values indicate a higher value obtained with NoSem.

As expected, no races were detected on either the Sem or NoSem datasets.

In general, there was no difference in XEGMAP's performance in terms of GMAPs, code block coverage or call stack depth, with or without extra semantic information. In fact, across the 21 binaries and 3 metric categories—a total of 69 comparisons—there were only 11 results that were different, clearly highlighted in the table. This is a statistically relevant result and indicates that XEGMAP is, on average, unperturbed in exploring binaries when semantic information is removed, this showing resilience towards general lack of semantics, rendering it more applicable to COTS and binaries sourced with arbitrary complication configurations. The behaviour shown here means that producing XEGMAP, fulfills the initial aim of developing an approach that

| Test Case | Race Detected | | | GMAPs | | | Call Stack Depth | | | Code Blocks | | | Avg. Step Time (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Sem | NoSem | Δ | Sem | NoSem | Δ | Sem | NoSem | Δ | Sem | NoSem | Δ | Sem | NoSem | Δ |
| dd | 0 | 0 | 0 | 368 | 369 | -1 | 6 | 6 | 0 | 1266 | 1139 | 127 | 3.4e-1 | 3.3 | -3.0 |
| df | 0 | 0 | 0 | 103 | 134 | -31 | 3 | 3 | 0 | 43 | 43 | 0 | 2.5e-3 | 1.5e-1 | -1.5e-1 |
| dircolors | 0 | 0 | 0 | 36 | 36 | 0 | 3 | 3 | 0 | 41 | 41 | 0 | 6.1e-4 | 1.3e-2 | -1.2e-2 |
| dirname | 0 | 0 | 0 | 18 | 18 | 0 | 3 | 3 | 0 | 39 | 39 | 0 | 3.2e-4 | 5.1e-3 | -4.8e-3 |
| du | 0 | 0 | 0 | 94 | 94 | 0 | 3 | 3 | 0 | 4551 | 4551 | 0 | 2.0e-2 | 2.9e-1 | -2.7e-1 |
| echo | 0 | 0 | 0 | 11 | 11 | 0 | 3 | 3 | 0 | 225 | 237 | -12 | 1.0e-1 | 3.8e-1 | -2.8e-1 |
| bracket | 0 | 0 | 0 | 100 | 14 | 86 | 9 | 7 | 2 | 485 | 485 | 0 | 7.2e-3 | 7.6e-3 | -4.0e-4 |
| env | 0 | 0 | 0 | 105 | 105 | 0 | 3 | 3 | 0 | 81 | 81 | 0 | 2.2e-3 | 1.3e-1 | -1.3e-1 |
| expand | 0 | 0 | 0 | 31 | 31 | 0 | 3 | 3 | 0 | 40 | 40 | 0 | 6.0e-4 | 2.6e-2 | -2.5e-2 |
| expr | 0 | 0 | 0 | 148 | 148 | 0 | 11 | 11 | 0 | 315 | 320 | -5 | 2.1e-1 | 3.0 | -2.8 |
| factor | 0 | 0 | 0 | 21 | 21 | 0 | 3 | 3 | 0 | 52 | 52 | 0 | 5.4e-5 | 1.0e-2 | -9.9e-3 |
| false | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 4 | 4 | 0 | 3.8e-5 | 2.0e-4 | -1.6e-4 |
| fmt | 0 | 0 | 0 | 49 | 49 | 0 | 5 | 5 | 0 | 95 | 120 | -25 | 2.7e-2 | 1.8e-1 | -1.5e-1 |
| fold | 0 | 0 | 0 | 26 | 26 | 0 | 3 | 3 | 0 | 41 | 41 | 0 | 4.0e-4 | 9.2e-3 | -8.8e-3 |
| getlimits | 0 | 0 | 0 | 13 | 13 | 0 | 3 | 3 | 0 | 46 | 46 | 0 | 7.8e-4 | 1.5e-2 | -1.4e-2 |
| ginstall | 0 | 0 | 0 | 66 | 66 | 0 | 3 | 3 | 0 | 60 | 60 | 0 | 1.5e-3 | 4.6e-2 | -4.4e-2 |
| groups | 0 | 0 | 0 | 19 | 19 | 0 | 3 | 3 | 0 | 38 | 38 | 0 | 4.2e-4 | 7.6e-3 | -7.2e-3 |
| head | 0 | 0 | 0 | 93 | 39 | 54 | 7 | 4 | 3 | 398 | 141 | 257 | 1.7e-1 | 2.6 | -2.4 |
| hostid | 0 | 0 | 0 | 14 | 14 | 0 | 3 | 3 | 0 | 46 | 46 | 0 | 3.1e-4 | 5.1e-3 | -4.8e-3 |
| id | 0 | 0 | 0 | 45 | 45 | 0 | 3 | 3 | 0 | 40 | 40 | 0 | 7.4e-4 | 1.7e-2 | -1.6e-2 |
| join | 0 | 0 | 0 | 108 | 108 | 0 | 3 | 3 | 0 | 58 | 58 | 0 | 1.7e-3 | 7.9e-2 | -7.7e-2 |

Table 5: Comparison of XEGMAP execution results on OSL21-noraces binaries with ("Sem") and without ("NoSem") semantic information i.e. execution on the OSL21-noraces-sem versus OSL21-noraces-nosem datasets. The Δ column indicates the difference between Sem and NoSem for corresponding metric values. Non-zero *Delta* values are indicated in pink highlighting.

can successfully process stripped binaries has been achieved.

Of the 11 results with differences across the GMAP, code block coverage, and call stack depth metrics, 7 cases resulted in lower values under NoSem than Sem, and this was expected, given the lack of semantic information.

In 4 cases, NoSem had higher values than Sem i.e. negative $\Delta$ values, as follows. Binaries "dd" and "df" had higher GMAPs detected under NoSem. This is attributed to the high amount of GMAPs that do not have any computational load, or complex structures obscuring their access. A cursory review of the dd source code shows this to be the majority of GMAP interaction, particularly when issuing a shallow exploration or a low amount of pre-constrained symbolic execution steps.

Binaries "echo", "expr" and "fmt" had higher code block coverage under NoSem. This can be attributed to the lack of impact that stripping semantic and symbolic information had on CFG derivation. This was likely due to the relatively simple nature of these binaries;which meant that even if blocks are sampled randomly, the small population of logical blocks causes symbolic exploration algorithms that are non-dependent on semantic information–i.e. those that use statistical heuristics– sometimes win in block coverage count.

Finally, observing the average step time, all $\Delta$ values are negative, which means that processing under NoSem took longer than under Sem, which is as expected. However, for 18 of the 21 binaries, the difference in average step time was hardly significant, with differences ranging between magnitude -1 and -3, implying very small differences. This is along with expectation; processing without any semantic information causes XEGMAP to misidentify certain function calls—-it may also affect CFG accuracy as some return branches in the CFG may depend on semantic information—that would greatly affect state selection heuristics and this expectably increases the time to process a single SE step.

For only 3 of the 21 binaries i.e. "dd", "expr" and "head", XEGMAP's average step time increased relatively significantly by between 2.4 and 3.0 seconds in the absence of semantic information. The overall lack of difference in timing is attributed partially to the lack or low amount of pthread specific calls in the sample set of binaries, removing the semantic information would therefore not affect the timing in a lot of test cases.

Overall, it is clear that XEGMAP is completely effective in the absence of semantic information and seems to be mostly unaffected by it. A result showing the resilience of the algorithm and its applicability to a wide variety of binaries. Essentially allow users to analyze binaries for race conditions,

without having to control for the availability of semantic information, to a large extent.

## 7. Summary and Conclusion

This paper presented a comprehensive evaluation of XEGMAP against two state-of-the-art symbolic execution guidance strategies, Stochastic and LoopSeer. The evaluation spanned three distinct datasets: UBN11 (COTS binaries), a custom dataset with injected race conditions, and OSL21 (open-source binaries tested with and without semantic information). The results highlight several critical findings regarding the efficacy of XEGMAP in the domain of binary-only race detection.

First, regarding Exploration Effectiveness, XEGMAP demonstrated a decisive advantage in locating Global Memory Access Points (GMAPs). Across the UBN11 dataset, XEGMAP identified approximately 6 to 7.5 times more GMAPs than the comparative approaches and achieved significantly higher code block coverage (peaking at 36,755 blocks). Crucially, XEGMAP exhibited superior consistency, successfully locating GMAPs in binaries where Stochastic and LoopSeer failed completely. This suggests that the directed execution strategy employed by XEGMAP is better suited for navigating the complex state spaces of real-world binaries than randomized or loop-centric approaches.

Second, in terms of Race Detection, XEGMAP proved to be the only viable tool for the target problem. In the custom dataset, XEGMAP successfully detected 100% of the prepared race conditions within the execution limit. In contrast, neither Stochastic nor LoopSeer detected a single race. This result validates XEGMAP's core hypothesis: that effective race detection in binaries requires a guidance system explicitly tuned to track thread interactions and memory access patterns, rather than general coverage metrics alone.

Third, the evaluation of Resilience to Stripped Binaries (OSL21) confirmed XEGMAP's robustness. The comparison between semantic-rich and stripped (NoSem) execution showed negligible performance degradation. In 69 metric comparisons, only 11 showed divergence, and in several cases, XEGMAP performed marginally better without semantic data. This finding is significant for the analysis of Commercial Off-The-Shelf (COTS) software, confirming that XEGMAP does not rely on debug symbols or source code to function effectively.

However, the evaluation also identified Limitations. XEGMAP incurs a higher computational cost per step compared to Stochastic, a trade-off necessitated by its heavier context tracking. Additionally, the failure to process `dpkg-divert` highlights remaining challenges in handling complex binary loaders and non-deterministic Control Flow Graph derivation.

In conclusion, XEGMAP fills a significant gap in the current state of the art. It stands as a unique solution capable of performing race detection on binary-only formats without source code. By balancing deep code coverage with heuristic prioritization of memory-interactive states, XEGMAP offers a robust, consistent, and effective approach to vulnerability discovery in modern software binaries.

## 8. Declarations

### *8.1. Funding*

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

## References

[1] J. Schilling, A. Wendler, P. Görz, N. Bars, M. Schloegel, T. Holz, A binary-level thread sanitizer or why sanitizing on the binary level is hard, in: 33rd USENIX Security Symposium (USENIX Security 24), 2024, pp. 1903–1920.

[2] R. Pereira, G. Stelle, P. Carribault, Taskgrind: Heavyweight dynamic binary instrumentation for parallel programs analysis, in: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2024, pp. 214–221.

[3] T. Zhang, C. Jung, D. Lee, Prorace: Practical data race detection for production use, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 149–162. doi:10.1145/3037697.3037708.
URL `https://doi.org/10.1145/3037697.3037708`

[4] X. Sun, Z. Chen, J. Shi, Y. Zhang, P. Di, X. Li, Z. Zuo, Hardrace: A dynamic data race monitor for production use, arXiv preprint arXiv:2410.18412 (2024).

[5] F. Gu, Q. Guo, L. Li, Z. Peng, W. Lin, X. Yang, X. Gong, COMRace: Detecting data race vulnerabilities in COM objects, in: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA, 2022, pp. 3019–3036.
URL https://www.usenix.org/conference/usenixsecurity22/presentation/gu-fangming

[6] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, 2016. doi:10.14722/ndss.2016.23368.

[7] Q.-L. Chen, J.-J. Bai, Z.-M. Jiang, J. Lawall, S.-M. Hu, Detecting data races caused by inconsistent lock protection in device drivers, 2019, pp. 366–376. doi:10.1109/SANER.2019.8668017.

[8] J. Huang, Ufo: predictive concurrency use-after-free detection, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE, 2018, pp. 609–619.

[9] Netzer, Detecting race conditions in parallel programs that use semaphores, Algorithmica 35 (2003) 321–345.

[10] J. Mellor-Crummey, On-the-fly detection of data races for programs with nested fork-join parallelism, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, Association for Computing Machinery, New York, NY, USA, 1991, p. 24–33. doi:10.1145/125826.125861.
URL https://doi.org/10.1145/125826.125861

[11] V. Kahlon, C. Wang, Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs, in: CAV, 2010.

[12] C. Flanagan, S. N. Freund, Fasttrack: Efficient and precise dynamic race detection, in: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 121–133.

doi:10.1145/1542476.1542490.
URL https://doi.org/10.1145/1542476.1542490

[13] W. C. Yuan Yu, Tom Rodeheffer, Racetrack: efficient detection of data race conditions via adaptive tracking, in: Proceedings of the twentieth ACM symposium on Operating systems principles, 2005, pp. 221–234.

[14] M. Xu, S. Kashyap, H. Zhao, T. Kim, Krace: Data race fuzzing for kernel file systems, in: IEEE S&P 2020, 2020.

[15] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, I. Shin, Razzer: Finding kernel race bugs through fuzzing, 2019 IEEE Symposium on Security and Privacy (SP) (2019) 754–768.

[16] E. J. Schwartz, T. Avgerinos, D. Brumley, All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask), in: 2010 IEEE Symposium on Security and Privacy, IEEE, 2010. doi:10.1109/sp.2010.26.

[17] I. Erete, A. Orso, Optimizing constraint solving to better support symbolic execution, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2011, pp. 310–315.

[18] W. Visser, J. Geldenhuys, M. B. Dwyer, Green: reducing, reusing and recycling constraints in program analysis, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–11.

[19] F. Wang, Y. Shoshitaishvili, Angr-the next generation of binary analysis, in: 2017 IEEE Cybersecurity Development (SecDev), IEEE, 2017, pp. 8–9.

[20] C. Cadar, D. Dunbar, D. Engler, Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, USENIX Association, USA, 2008, p. 209–224.

[21] I. Haller, A. Slowinska, M. Neugschwandtner, H. Bos, Dowsing for overflows: A guided fuzzer to find buffer boundary violations, in: Proceedings of the 22nd USENIX Conference on Security, SEC'13, USENIX Association, USA, 2013, p. 49–64.

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, Bitblaze: A new approach to computer security via binary analysis, in: International Conference on Information Systems Security, Springer, 2008, pp. 1–25.

[23] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley, Unleashing mayhem on binary code, in: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, IEEE Computer Society, USA, 2012, p. 380–394. doi:10.1109/SP.2012.31.
URL https://doi.org/10.1109/SP.2012.31

[24] T. Avgerinos, A. Rebert, S. K. Cha, D. Brumley, Enhancing symbolic execution with veritesting, Commun. ACM 59 (6) (2016) 93–100. doi:10.1145/2927924.
URL https://doi.org/10.1145/2927924

[25] D. Brumley, I. Jager, T. Avgerinos, E. Schwartz, Bap: A binary analysis platform, Vol. 6806, 2011, pp. 463–469. doi:10.1007978-3-642-22110-1_37.

[26] A. Di Federico, M. Payer, G. Agosta, rev. ng: a unified binary analysis framework to recover cfgs and function boundaries, in: Proceedings of the 26th International Conference on Compiler Construction, 2017, pp. 131–141.

[27] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, R. Barua, A compiler-level intermediate representation based binary analysis and rewriting system, in: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 295–308.

[28] G. Balakrishnan, T. Reps, Analyzing memory accesses in x86 executables, in: International conference on compiler construction, Springer, 2004, pp. 5–23.

[29] A. Ahmad, S. Lee, P. Fonseca, B. Lee, Kard: Lightweight data race detection with per-thread memory protection, in: Proceedings of the 26th

ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 647–660.

[30] K. Serebryany, T. Iskhodzhanov, Threadsanitizer: data race detection in practice, in: Proceedings of the workshop on binary instrumentation and applications, 2009, pp. 62–71.

[31] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, J. W. Davidson, Zipr: Efficient static binary rewriting for security, in: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2017, pp. 559–566.

**Algorithm 1** Naive Execution for Global Memory Access Point (Negmap) Discovery

**while** $\mathbb{W} \neq \emptyset$ **do**
    $s \longleftarrow \text{Pop}(\mathbb{W})$
    $\rho \longleftarrow \text{Random}(0, 1)$
    **if** $\neg\text{HeuristicCheck}(s, \Gamma, S_{map}, n) \wedge (\rho < 0.40)$ **then**
        $\mathbb{D} \longleftarrow \mathbb{D} \cup \{s\}$ ▷ Check failed: Defer state and skip processing
    **else**
        $\mathbb{E} \longleftarrow \text{Events}(s)$
        **for** $e \in \mathbb{E}$ **do**
            **if** $\text{IsMemRead}(e) \vee \text{IsMemWrite}(e)$ **then**
                $(m, p) \longleftarrow \text{GetMemOp}(e)$
                **if** $\text{IsGlobalMem}(m)$ **then**
                    $\Gamma \longleftarrow \Gamma \cup \{(e, m, p)\}$
                    $\textsc{Degmap}(S_{map}, \Gamma, s)$
                **end if**
            **end if**
        **end for**
    **end if**
    **if** $|\Phi(s)| > 0$ **then**
        $S_{next} \longleftarrow \Phi(s)$
        $\mathbb{W} \longleftarrow \mathbb{W} \cup S_{next}$
    **end if**
    **if** $\mathbb{W} = \emptyset \wedge |\mathbb{D}| > 0$ **then**
        $s_{retry} \longleftarrow \text{Pop}(\mathbb{D})$
        $\mathbb{W} \longleftarrow \{s_{retry}\}$
    **end if**
**end while**

**Algorithm 2** HeuristicCheck

1: **procedure** HEURISTICCHECK($s, \Gamma, S_{map}, n$) ▷ Discard state if it leads to a known GMAP, keep if it improves Block/Stack coverage
2: **if** $\Big(\exists \gamma \in \Gamma : s \rightsquigarrow^{n} \gamma\Big)$ **then**
3: **return** FALSE
4: **end if**
5: **if** $\neg\Big(\exists s_{succ} \in S_{map} : s \rightsquigarrow^{n} s_{succ}\Big) \bigvee \Big(\beta(s_{succ}) > \beta(s) \wedge \Delta(s_{succ}) > \Delta(s)\Big)$ **then**
6: **return** FALSE ▷ $\beta(s)$ returns block count, $\delta(s)$ returns callstack length
7: **end if**
8: **return** TRUE
9: **end procedure**

**Algorithm 3** Directed Execution For Global Memory Access Point (Degmap) Discovery

1: **procedure** DEGMAP($S_{map}, \Gamma, \mathbb{W}$)
2: **while** $\mathbb{W} \neq \emptyset$ **do**
3: $s \longleftarrow \text{Pop}(\mathbb{W})$
4: $\rho \longleftarrow \text{Random}(0, 1)$
5: **if** $\neg\text{HeuristicCheck}(s, \Gamma, S_{map}, n) \wedge (\rho < 0.40)$ **then**
6: $\mathbb{D} \longleftarrow \mathbb{D} \cup \{s\}$ ▷ Check failed: Defer state and skip processing
7: **else**
8: $\mathbb{E} \longleftarrow \text{Events}(s)$
9: **for** $e \in \mathbb{E}$ **do**
10: CallCheck$(s, e)$ ▷ check thread calls, update state context
11: **if** IsMemRead$(e) \vee$ IsMemWrite$(e)$ **then**
12: $(m, p) \longleftarrow \text{GetMemOp}(e)$ ▷ Address $m$, Size $p$
13: **if** IsGlobalMem$(m) \wedge m \notin \mathbb{M}_{history}$ **then**
14: **for** $(e_c, m_c, p_c) \in \Gamma$ **do**
15: **if** $\mathbb{T}(e_c) \neq \mathbb{T}(e) \wedge \big(\mathbb{L}(e_c) \cap \mathbb{L}(e) = \emptyset\big)$ **then**
16: **if** $[m_c, m_c + p_c) \cap [m, m + p) \neq \emptyset$ **then**
17: REPORTRACE$(s, m, p)$
18: **end if**
19: **end if**
20: **end for**
21: $\Gamma \longleftarrow \Gamma \cup \{(e, m, p)\}$
22: **end if**
23: **end if**
24: **end for**
25: **end if**
26: **if** $|\Phi(s)| > 0$ **then**
27: $S_{next} \longleftarrow \Phi(s)$
28: $\mathbb{W} \longleftarrow \mathbb{W} \cup S_{next}$
29: **else if** $\mathbb{W} = \emptyset \wedge |\mathbb{D}| > 0$ **then** $\mathbb{W} \longleftarrow \text{Pop}(\mathbb{D})$
30: **end if**
31: **end while**
32: **end procedure**

**Algorithm 4** CallCheck

1: **procedure** CallCheck($s, e$) ▷ check if an event contains any thread specific function calls
2: **if** IsThreadCreate($e$) **then**
3: $\mathbb{T}(s) \longleftarrow \mathbb{T}(s) \cup \{\text{NewThreadID}(e)\}$
4: **else if** IsThreadJoin($e$) **then**
5: $\mathbb{T}(s) \longleftarrow \mathbb{T}(s) \setminus \{\text{TargetThreadID}(e)\}$
6: **else if** IsMutexLock($e$) **then**
7: $\mathbb{L}(s) \longleftarrow \mathbb{L}(s) \cup \{\text{LockID}(e)\}$
8: **else if** IsMutexUnlock($e$) **then**
9: $\mathbb{L}(s) \longleftarrow \mathbb{L}(s) \setminus \{\text{LockID}(e)\}$
10: **end if**
11: $s \longleftarrow \text{UpdateContext}(s, \mathbb{T}, \mathbb{L})$
12: **end procedure**