Automatic Analysis of Critical Sections for Efficient Secure Multi-Execution

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

Thomas Göthel
Technische Universität Berlin
Berlin, Germany
thomas.goethel@tu-berlin.de

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

***Abstract*—Enforcement of hypersafety security policies such as noninterference can be achieved through Secure Multi-Execution (SME). While this is typically very resource-intensive, more efficient solutions such as Demand-Driven Secure Multi-Execution (DDSME) exist. Here, the resource requirements are reduced by restricting multi-execution enforcement to critical sections in the code. However, the current solution requires manual binary analysis. In this paper, we propose a fully automatic critical section analysis. Our analysis extracts a context-sensitive boundary of all nodes that handle information from the reachability relation implied by the control-flow graph. We also provide evaluation results, demonstrating the correctness and acceleration of DDSME with our analysis.**

***Keywords*-Noninterference Enforcement, Binary Analysis, Multi Execution**

I. Introduction

Automatic data processing has become an important backbone of modern society. The ever-increasing amount of processed data, at finer levels of granularity, together with new economic models that focus on data acquisition, pushes the need to ensure its confidentiality. This can be achieved by enforcing security properties such as noninterference [1]. Unfortunately, traditional Information Flow Control (IFC) systems that monitor a single execution at a time are unable to do so precisely [2]. Noninterference is a 2-safety hyperproperty in nature [3], [4] and, thus, can only be validated precisely by relating sets of execution traces. This has given rise to multi-execution enforcement systems such as Secure Multi-Execution (SME) [5]. In Secure Multi-Execution, a program is executed once per level in the underlying security lattice. Each execution is responsible for output on channels with matching security classification and only has access to information from channels with equal or lower classification. It is secure by design; since output at one level is produced by an execution that does not obtain information with higher classification, there can be no interference. Given the right scheduling, SME is also transparent [6] and timing-sensitive [7]. Hence, it fulfills the main requirements for binary rewriting enforcement mechanisms as defined in Hamlen et al. [8]. Still, we do not see a lot of SME systems used in the wild.

The problem with SME enforcement is twofold. First, due to the intrinsic mechanism of multi-execution, SME enforcement comes with multiplied resource requirements. Second, as noted by Schmitz et al. [9], building an SME system takes a considerable effort. Consequently, most existing SME solutions are specially tailored to one (high-level) language and impose intolerable overhead.

Our goal is to design and build a more efficient SME-based enforcement method, that applies to low-level code. To achieve higher efficiency than traditional SME, we proposed Demand-Driven Secure Multi-Execution (DDSME) [10]. Using DDSME, we reduce the resource requirements by restricting SME enforcement to *critical sections* of the target binary program. Naturally, the challenge is to do so without negative impact on the strong security and transparency guarantees of traditional SME. The limiting factors are the scalability of the analysis of the critical section and the precision of the results.

In this paper, we propose an efficient, fully automatic static analysis to determine critical sections. We demonstrate the correctness and increased efficiency of our updated prototype with new evaluation results. We begin by providing some preliminaries in the next section, including our Demand-Driven Secure Multi-Execution approach. In Section III, we introduce our critical section analysis method, based on context-sensitive reachability analysis. Finally, in Section IV we provide new measurements and then give an overview of related work. Finally, we conclude this paper in Section VI.

II. Background

In this section, we introduce noninterference as the desired security policy and Secure Multi-Execution as a way to enforce it. We then outline our Demand-Driven Secure Multi-Execution approach [10] and give a definition for control-flow graphs.

A. Noninterference

Noninterference defines that a program is confidential if every execution trace produced under environments that agree on public inputs also agrees on public outputs. If that is the case, private inputs are *not interfering* with the production of public outputs. Consequently, public outputs are (strongly) independent of private inputs. Noninterference is considered a 2-safety or hypersafety property [3], [4], as at least two traces are needed to falsify it. As such, enforcement is similar to proving behavioral equivalence, which is known to be Π_2-hard [8]. Such properties can be enforced using binary rewriting systems, of which Secure Multi-Execution (SME) [5] offers the strongest security guarantees [2].

B. Secure Multi-Execution

As the name suggests, SME works by executing the same code multiple times - once for each level in the security lattice. Each execution gets input only from channels with equal or lower classification and is responsible for output on channels at its own level. As such, SME is secure by design. Because every output is produced in ignorance of information with higher classification, it cannot be affected by it. It is also transparent. Transparency means that the behavior of any program that already satisfies noninterference is not altered too much. In the context of this paper, it means for any program that satisfies progress-sensitive noninterference, the same events occur on each channel as under unprotected execution and they are in the same order. SME achieves this, since each execution has access to all information with equal or lower classification. Hence, if private information is unnecessary, it can produce the same output. Finally, SME can enforce timing-sensitive noninterference if all executions are scheduled independently. For more details, we again refer the reader to [6].

C. Demand-Driven Secure Multi-Execution

To make SME more efficient, we have introduced the concepts of Secretless Execution, Critical Section Analysis and Demand-Driven Secure Multi-Execution (DDSME) [10]. A secretless execution describes an execution that has no knowledge about confidential information (i.e., through inputs from channels with high classification in the typical two-level lattice). The reasoning behind this is that if a deterministic program is secretless, it is naturally non-interferent and does not require expensive Secure Multi-Execution enforcement. Since secretless execution is a 1-safety property, it can be efficiently enforced using a single-execution monitor. Once the execution is no longer secretless, i.e. upon obtaining classified inputs, we switch to more expensive Secure Multi-Execution (SME) enforcement. By using SME enforcement, we can guarantee noninterference even in the presence of classified information. However, SME enforcement is less efficient than secretless execution. Thus, we aim to switch back to the single-execution monitor as soon as the classified information is no longer needed. Since this requires knowledge about the future states of the program, we apply static analysis before running executing it. Through the static analysis, we extract a description of all *critical* states that require SME enforcement. Here, a state is considered critical if

1) it contains sensitive information,
2) there exists a (feasible) path to an output operation, and
3) the output information is strongly dependent on sensitive information.

We call the set containing all critical states the *critical section*.

During Secure Multi-Execution enforcement, we monitor if the various variants are still in critical states. When the critical section has been left by all variants, the SME-executions are terminated or replaced. Enforcement continues with less expensive secretless execution monitoring. More details on Demand-Driven Secure Multi-Execution can be found in our original paper [10].

D. Control-Flow Graph

A control-flow graph (CFG) is defined as a directed graph $G := (N, \rightarrow)$, where N is a set of nodes and $\rightarrow: N \times N \times L$ is a labeled vertex representing control flow. To differentiate between kinds of control flow, we use the label set $L = \{call, return, fake, jump\}$. We write $s \xrightarrow{\ell} t \in G$ to say that there exists a transition from node s to node t with label ℓ in graph G. We use $s \xrightarrow{*} t \in G$ to say that there is a (possibly empty) path from node s to node t. We assume that the set of nodes N is implied by the transition relation. Hence, we write $n \in G \Rightarrow \exists m, \ell : (n, m, \ell) \in G \vee (m, n, \ell) \in G$.

III. Critical Section Analysis

The increased efficiency of Demand-Driven Secure Multi-Execution comes from a focus of the expensive multi-execution enforcement on a limited number of critical states. States are critical if there exists a feasible path that leads to an output which is strongly dependent on sensitive information in the state. Unfortunately, both feasibility and strong dependency are semantic properties and therefore usually require very complex program-wide analyses. To avoid these, static information flow analysis must trade precision for efficiency. As a consequence, such analyses are often performed only intra-procedurally or ignore the calling context. The resulting methods are fast and sound, but less permissive due to over-approximations.

As Secure Multi-Execution is secure by design, over-approximated partitioning does not harm its security guarantees. It does, however, reduce the *efficiency* of our demand-driven optimization. In an execution where no state is critical, the H-execution is never run. Therefore, the overall approach is very efficient, but may be intransparent. Conversely, if every state is considered critical, then the H-execution is run continually. This guarantees the usual per-channel transparency, but is as inefficient as traditional Secure Multi-Execution. The goal of our critical section analysis is therefore to find a tight over-approximation that enables efficient and transparent enforcement.

Unfortunately, the heavy use of pointers over flat memory in binary code render most data-flow analyses impractical. In the absence of declared variables and type information, information flow analyses like dependence analysis become significantly harder to solve. In light of these challenges, we propose a very fast estimation based on reachability. While this provides a strong over-approximation, we consider this a secure and transparent solution that enables us to increase the efficiency of Secure Multi-Execution at very little cost. It also offers a starting point for further improvements in the future.

Our key idea to reduce the over-approximation due to context-insensitivity is to use the fact that called functions will eventually return to their call-site[1]. Hence, we add "fake returns" [11] to the CFG, which directly connect a call node to its call-site. This allows us to construct a context-sensitive critical section without the need for expensive tracing the call-stack. Instead, we model the call stack height indirectly through the call and return edges from the CFG. The details of this analysis are given in section III-B.

The resulting critical section covers states that may be critical. However, in practice it is very costly to check for membership at every step. For our Demand-Driven Secure Multi-Execution Enforcement, we are actually only interested in whether the execution has left the critical section. Additionally, as a consequence of our context-sensitivity, our critical section result excludes functions that are "stepped-over". Therefore, the result of our analysis is a mixed-approximation. We solve both problems by boundary analysis, which we compute in the final step of our algorithm. We provide the details of this analysis in section III-C. In the next section, we introduce a running example to outline the differences between a trivial, context-insensitive solution and our our context-sensitive approach.

A. Context-Insensitive Example

Simply slicing the control-flow graph to calculate reachability-based criticality of states creates context-insensitive results that are unnecessarily large. To illustrate this, consider the example in Figure 1. The pseudo-code program consists of three functions, `main`, `a`, and `b`. The `main` function first calls the system function `read`, the two external functions `lib_a` and `lib_b`, and then the system function `write`. The functions `a` and `b` each wrap one of the external functions and a system function.

The corresponding control-flow graph (CFG) is given in Figure 1b. Here, the nodes are labeled according to the function they belong to. Nodes $A_{1...3}$ ($B_{1...3}$) belong to function `a` (`b`), and nodes $M_{1...5}$ to `main`. Additionally, the read function is denoted ? and the write function is denoted !. Finally, the external functions are represented using nodes L_A and L_B. Edges representing control transfer via a call instruction are depicted as solid lines. Edges representing control transfer via return instructions are depicted as dotted lines.

Note that both `a` and `main` call to and return from `read` and subsequently `lib_a`. Similarly, `b` and `main` both call to and return from `lib_b` and subsequently `write`. As the created control-flow graph does not contain information about the calling context, it is an over-approximation. Hence, if we intersected all nodes reachable from the source node ? with all nodes reaching the sink node !, we would get the result shown in Figure 1c. This result is context-insensitive. Therefore, while the critical path $?.M_2.L_A.M_3.L_B.M_4.!$ correctly is part of this set, so are paths starting with $?.A_2.L_A.M_3...$ or ending with $....M_3.L_B.B_2.!$.

The problem is that this analysis does not enforce legal control-flow. If L_A has been called from A_2, the control-flow legally has to return to A_3. This also implies that after calling L_B from M_3 it is illegal to return to B_2. Consequently, the result in Figure 1c is unnecessarily large, and thus may lead to less efficient enforcement.

B. Context-Sensitive Reachability

Our goal is to find tight estimations of the critical section based on reachability. As we have seen above, the control-flow graph (CFG) is context-insensitive and thus any information directly inferred from it is also imprecise. The key idea to make our reachability analysis context-sensitive is to interpret call instructions with fall-through semantics. This expresses the underlying assumption that a called function eventually must return to the instructions following the call site. Interpreting calls with fall-through semantics means that new edges are added to the call graph, such that each call node is connected directly to its return target. These additional "fake" edges directly enforce that control-flow respects the calling context in the control-flow graph. In the following, we outline our method to extract critical sections from binaries based on context-sensitive reachability. To make our approach efficient, it is works fully graph-based and does not require expensive call stack analyses. We achieve this by indirectly modeling the call stack height through the call and return edges from the CFG. The result has mixed precision, as unnecessary calls to functions which neither contain the sink nor source nodes are cut away. We remedy this under-approximation though our boundary analysis described in Section III-C.

Reconsider the CFG in Figure 1b. If we removed all call edges, to replace them with context-sensitive intra-procedural edges, we would also remove the edge connecting M_4 to the sink node !. This is because source and sink node are connected via a function that is lower in the call stack. Hence, we first need to return from the source node ? and later call the sink node !. This implies that both the return and call edges contain important information that cannot be discarded. On the other hand, retaining both edges and adding context-sensitive ones, only makes the graph more connected and thus even more imprecise.

Our solution is to split the context-sensitive control-flow graph into two versions. On the one hand, we use a version where all call edges are removed. Consequently, forward analyses on this graph can only go downward in the call stack. As it matches the notion of "stepping out" of a procedure used in the debugging world, we call it step-out graph (SOG).

$$SOG := \{(s, t, \ell) \mid s \xrightarrow{\ell} t \in CFG \wedge \ell \neq call\}$$

Conversely, backward analyses on the SOG naturally can only go upward in the call stack. On the other hand, we use a version where all return edges are removed. Here, forward analyses can only go upward in the call stack and backward analyses can only go downward in the call stack. As this graph matches the notion of "stepping into", we call it step-into graph (SIG).

$$SIG := \{(s, t, \ell) \mid s \xrightarrow{\ell} t \in CFG \wedge \ell \neq return\}$$

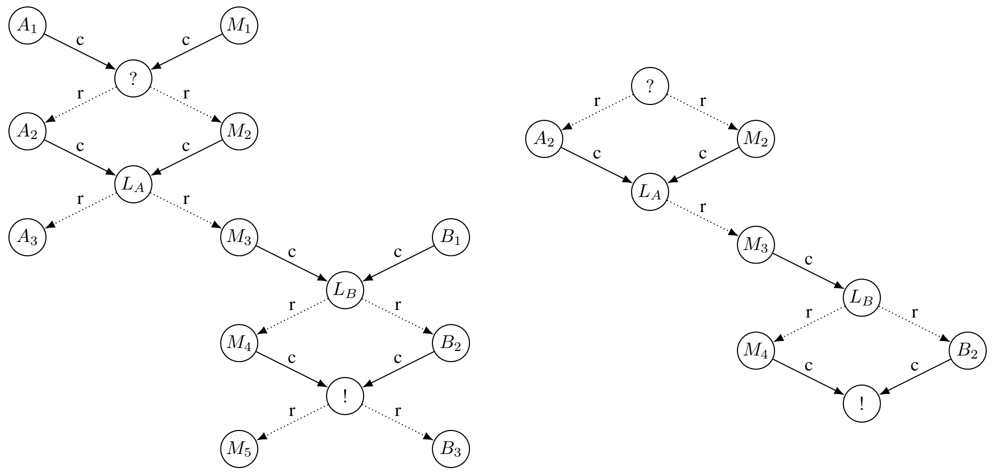
[1]We assume that non-returning or tail-recursive functions have been handled during control-flow recovery.

```
main() {
  read(buf);
  lib_a(buf);
  lib_b(buf);
  write(buf);
}

a() {
  read(buf);
  lib_a(buf);
}

b() {
  lib_b(buf);
  write(buf);
}
```

(a) Example Code



(b) Control-Flow Graph

(c) Context-Insensitive Chop

Fig. 1: Running Example

The SOG and SIG of our example from Figure 1 are given in Figure 2a and Figure 2b, respectively. We depict the added context-sensitive "fake" edges as dashed lines.

Using the SOG and SIG, we can address the problem explained above. First, we calculate the subgraph that is forward-reachable from the source node using the SOG: $FG := \{(s,t,\ell) \mid src \xrightarrow{*} s \xrightarrow{\ell} t \in SOG\}$. This gives us the "downward-reachable" graph from the source (marked with thickness in Figure 2a). By construction, it is assured that nodes belonging to functions at a higher call stack are removed. Hence, unless the source node is inside the function with the highest call stack, the result must be smaller than the CFG. However, as stated above, the sink node may very well be one of the removed nodes. Therefore, we also calculate the subgraph that is backward-reachable from the sink node using the SIG: $BG := \{(s,t,\ell) \mid s \xrightarrow{\ell} t \xrightarrow{*} snk \in SIG\}$. This gives us the graph that is "upward-reaching" the sink node (marked with thickness in Figure 2b).

If the source node reaches downward into a function which subsequently reaches upward to the sink node, the intermediate nodes of this function are in the intersection of the subgraphs above: $IG := FG \cap BG$. In our example, these are the nodes $M_{2\ldots4}$ from the `main` function. Naturally, this would be an under-approximation as neither source nor sink node are contained in it. Conversely, the union of both sets may contain nodes that reachable from the sink but not reaching the source and vice versa. In the example, these include $A_{2,3}$ and $B_{1,2}$.

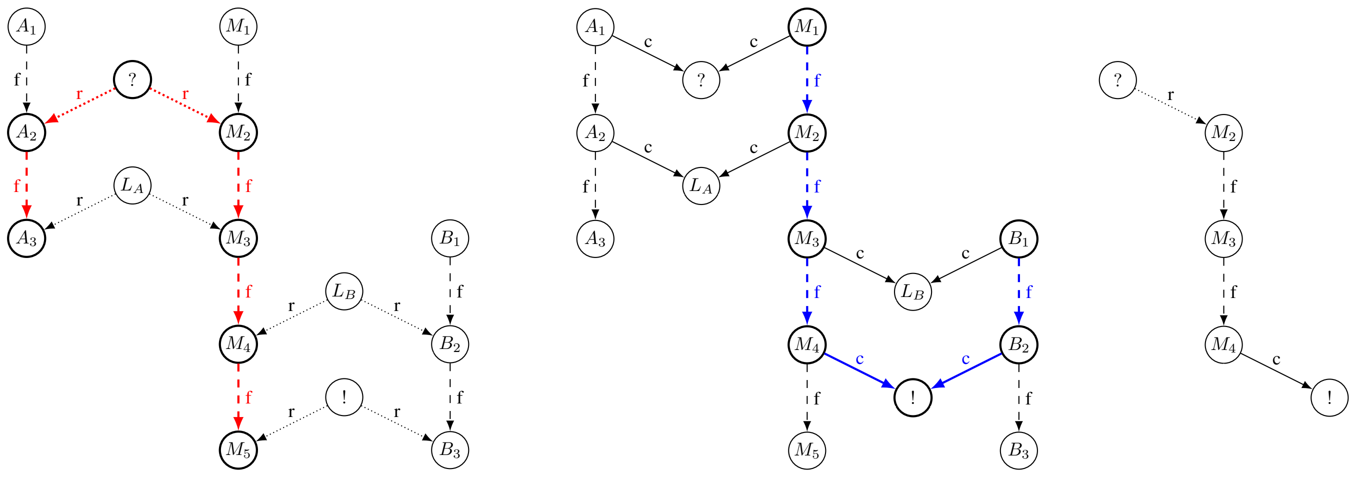
Our solution to both problems is to first intersect the sets and then inflate to include both sink and source. We achieve this by again using our SOG and SIG graphs. First, we calculate the set of backward-reachable nodes from the intersection using the SOG: $IBG := \{(s,t,\ell) \mid s \xrightarrow{\ell} t \xrightarrow{*} r \in SOG \wedge r \in IG\}$. This gives us all nodes that are downward-reaching the intersection. Then, we calculate the set of forward-reachable nodes from the intersection using the SIG: $IFG := \{(t,r,\ell) \mid s \xrightarrow{*} t \xrightarrow{\ell} r \in SIG \wedge s \in IG\}$. This gives us all nodes that are upward-reachable from the intersection. In union, these sets describe all nodes that downward-reach into the intersection or are upward-reachable. However, this union may again contain too many nodes (here, $M_{1,5}$ and $L_{A,B}$).

We solve this with the last step of our analysis. Before constructing the union of the downward-reaching and upward-reachable nodes of the intersection, we intersect each with the initially constructed reachability sets of the source and the sink nodes. First, we intersect the nodes that are downward-reaching the intersection with the downward-reachable nodes from the source to get exactly all nodes that downward-reach the intersection from the source node. Then, we intersect the upward-reachable nodes from the intersection with the nodes that are upward-reaching the sink to get exactly all nodes that upward-reach the sink from the intersection. The union of these two sets finally gives our result.

$$CS := (FG \cap IBG) \cup (BG \cap IFG)$$

It describes exactly all nodes that downward-reach into the intersection from the source node and upward-reach the sink node from the intersection. The result is depicted in Figure 2c.

Note that our result contains fewer nodes than the context-insensitive result from Figure 1c Also, it only contains the correct path. However, the external functions L_A and L_B are not part of the result, as they are stepped over. This is due to the context-sensitive assumption that we must eventually return from these functions to their respective return targets. To solve this, we instead use the boundary of this under-approximated critical section. We explain this in more detail in the next section.



(a) Step-Out Forward-Reachability from the Source (b) Step-Into Backward-Reachability from the Sink (c) Context-Sensitive Chop

Fig. 2: Context-sensitive Reachability Analysis

C. *Boundary Analysis*

The goal of our critical section analysis is to use the result to determine when the expensive multi-execution enforcement can be stopped. Hence, if it is too big, the multi-execution will run longer, diminishing the resource reduction of our approach. On the other hand, if it is too small, we may stop the multi-execution prematurely and thus potentially harm the transparency of our approach. Unfortunately, while our context-insensitive analysis described above creates inefficient over-approximations, our context-sensitive analysis creates intransparent under-approximations. We solve this problem by not using the results to determine when to stay in multi-execution mode but by using the opposite test. This means that we calculate the boundary of the critical section and stop when it is trespassed during execution. The advantage of this approach is that we can precisely control the amount of over-approximation added to make our analysis sound. Using the boundary has another great advantage. Given the nature of most programs, the boundary will usually contain far fewer nodes than the corresponding critical section. Consequently, using the boundary allows us to speed up the implementation by conducting fewer checks.

As described above, the result of our context-sensitive analysis is an under-approximation of the critical section as intermediate functions are not included. Due the construction of the result, they are instead stepped over. Hence, if we used the result directly, we would terminate the multi-execution whenever a function outside of the critical section is called. Therefore, our assumption that these functions must eventually return to the critical section must be added back to the approach. We do this by calculating the boundary of our critical section by collecting all destinations of edges in the original CFG, that

$$B = \{t \mid s \xrightarrow{\ell} t \in CFG \land s \in CS \land t \notin CS \land \ell \neq call\}$$

Calls outside of the critical section are not part of the boundary as we assume that they will eventually return to the critical section. In our example, the boundary would thus be M_5 and B_3. The boundary analysis is the last step in our reachability-based critical section analysis. We focus on reachability, as it can be extracted from realistic binaries through very fast, syntax-based analysis [12]. We demonstrate the analysis speed and correctness of results in Section IV. In the future, we may include more complex analyses to create tighter bounds, some of which we describe in the next section.

IV. Evaluation

With the evaluation, we demonstrate that Demand-Driven Secure Multi-Execution is applicable to real-world machine code. We show that our fully automatic critical section analysis can very quickly extract a reachability-based critical section description from real-world binaries. We also show that the extracted critical section can be used to securely enforce noninterference using DDSME. We compare our DDSME approach to traditional Secure Multi-Execution enforcement and show that our DDSME enforcement is more efficient. Since there exists no other implementation of neither traditional nor demand-driven SME enforcement for machine code, we use our own systems for this comparison. This has the advantage that the compared systems are relatively similar.

To evaluate our approach, we implemented a new version of our Demand-Driven Secure Multi-Execution (DDSME) system with queue-scheduling. Simulation of barrier-based scheduling comes down to joining the active H-execution with the L-execution before switching to M-level semantics. Additionally, we fully automatized the critical section analysis. For our tests, we are using `cat` from the coreutils package. `cat` reads the contents of from a list of files provided arguments and outputs them to `stdout`. The processing is performed in chunks, with the maximum size of each chunk determined by the size of the buffer. If reading from a file returns zero (indicating that the input stream has ended), the next file is processed. If no more files are listed, the program terminates.

cat is among the best-tested open-source software, relatively small but sophisticated, uses read and write, and varies with input size. Hence, it is a suitable candidate to demonstrate our approach. All evaluations have been performed on a Intel® Core™ i7-5600U CPU @ 2.60GHz machine with four physical cores.

A. Critical Section Analysis

The critical section analysis is based on the angr binary analysis framework [11]. We use their emulated control-flow recovery algorithm to obtain the control-flow graph (CFG). Note that while their approach is not necessarily sound, it is relatively fast and precise enough for our needs. Additionally, angr performs function detection, which allows us to easily identify calls to read and write. As an alternative, more efficient syntax-based analyses [12] could be used as we do not require additional information like points-to or alias analysis. However, these analyses may require additional information such as debugging symbols or Control-Flow Integrity (CFI) information to be present in the binary. Hence, for this paper, we stick with the out-of-the-box solution.

For the cat binary, it takes angr roughly three minutes to extract the CFG. The extracted CFG contains 10764 nodes and 17301 edges. This shows that, although cat can be considered a rather small binary, complexity quickly rises at machine code level. Since angr comes with control- and data-dependence graph analyses as well as a backwards slicing algorithm, we tried to apply these to cat to get a more precise critical section. However, after running the analyses for 48 hours, the backward slicing failed with an internal error. Hence, we continued with our reachability-based analyses.

In total, our analyses took under 10 minutes (including the three minutes for angr's CFR algorithm). The critical section covers reachability from read to write as well as from write to write (in case the same information is reused). Our context-sensitive critical section contains 180 nodes, which is less than 1.7% of the complete CFG. In contrast to this, the trivial, context-insensitive approach yields a critical section with 5312 nodes, which is roughly half of the CFG. Thus, our context-sensitive optimization yields a result that is nearly 30 times more precise. As expected, a more precise critical section also leads to less entries in the boundary. The boundary of our context-sensitive analysis contains two entries. The first is reached if the program encounters a critical error after which it will be terminated with no further output. The second is reached if all input provided to cat has successfully been processed. Under normal conditions, only the second bound should be reached. In contrast to this, the context-insensitive approach leads to a boundary with 103 elements. As we demonstrate in the following section, this leads to stark decrease in efficiency.

B. Demand-Driven Secure Multi-Execution

To show the increased efficiency of our approach, we tested cat with various input sizes. A larger input size implies that cat loops longer in the critical section. To be precise, cat requires an additional 107 user instructions to process a chunk of 131072 bytes on our machine. For the first chunk, cat requires 195953 instructions on average. Hence, the cost estimation for cat is described by $195953+\lfloor n/131072 \rfloor * 107$, where n is the input length. Figure 3 show our sampled runs and the cost estimation for natively running cat at various input lengths. When cat is run under ptrace tracing, the monitoring environment requires roughly 90000 additional instructions to set up, but does not affect the cost per chunk.

To show that our SME implementation is fairly efficient, we compare our solution to an idealistic SME without any per-chunk overhead. We assume that no input is provided as dummy values. Consequently, the increment depends only on the real input and not on any dummy values that have to be processed in the background. Hence, we assume a duplication of the base instructions of our monitored run of cat and use the 107 increment. As can be seen in Figure 3, our SME solution comes quite close to the base instructions, but adds about seven times more instructions per chunk.

Next, we measured the efficiency of DDSME using our context-sensitive analysis to determine the boundary. As expected, our DDSME optimization is significantly more efficient than traditional SME. In cases where the input is classified (DDSME High), the increment is the same as for traditional SME. However, as the critical section is relatively small, the majority of the execution happens outside of it. Consequently, the overall resource requirement is reduced by roughly 30%. Our DDSME implementation is also more efficient than ideal SME for classified inputs up to 16 MB.

As we use the same enforcement inside the critical section as we use for traditional SME, it makes sense that the increment is similar in this case. However, if the input turns out to be unclassified, our DDSME enforcement does not enter the critical section. As a consequence, the program is not duplicated at all. Naturally, this leads to an even bigger reduction of the resource requirements. In this case, labeled DDSME Low in Figure 3, not only can we save roughly 40% for small inputs but the reduction is amplified for larger inputs. Where traditional SME adds 765 instructions per increment, DDSME only adds 460 instructions for each chunk of unclassified input. Because of these additional savings, DDSME is better than ideal SME for up to 60 MB of unclassified input.

Finally, note that many of the additional instructions can be parallelized on a machine with multiple cores. Hence, the impact on the actual runtime may be even less. However, this is hard to measure, as the runtime also depends on a variety of other factors such as the current workload on the machine. Therefore, we focused on user instructions, as they are very stable across multiple runs.

C. Context-Insensitive Results

To demonstrate the effect of precision of the critical section on the runtime of our enforcement system, we compare our enforcement mechanism using the context-insensitive and the context-sensitive boundaries. Note that in both cases the boundaries are sound. Hence, the results always provide

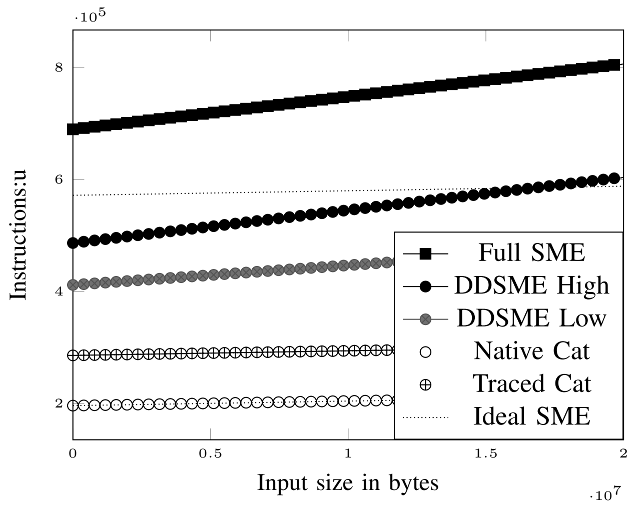


Fig. 3: `cat` DDSME enforcement cost visualization

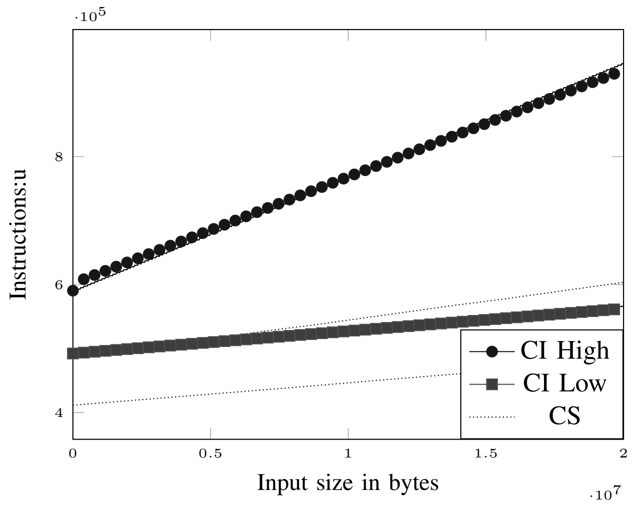


Fig. 4: Context-Insensitive (CI) and Context-Sensitive (CS)

transparency, meaning that the results of the enforcement mechanism are correct for all types of flows. However, as mentioned above, the result of our context-sensitive analysis not only leads to tighter boundaries but also to boundaries with less entries. Consequently, we not just save on multiplication but also on monitoring overhead. The cumulative effect is visualized in Figure 4. Instead of the 769 instructions per chunk for DDSME with classified inputs and our context-sensitive (CS) analysis, the context-insensitive boundary leads to 2334 instructions per chunk (CI High). In the unclassified case, the the increments is the same but the additional entries in the boundary lead to additional overhead in the setup of our engine.

D. Tighter Bounds

As explained above, `cat` processes the input files sequentially and in chunks. Consequently, the `write` becomes unreachable only after `cat` has checked that all files have been processed. Because this check is performed later in the program, we remain in Secure Multi-Execution mode even after the last chunk has been processed.

To evaluate this, we manually created tighter bounds. Using these, the SME mode is left once the end of a file is reached. Our experiments show that this saves about 2.000 user instructions. Naturally, if more than one file is passed to `cat`, we enter SME mode once per file. This comes with additional monitoring overhead of roughly 15.000 instructions, which goes beyond the 2.000 saved instructions. Hence, while this tightening of the bounds can achieve better performance, it very much depends on the specific setting. Only if more instructions can be saved than are added due to monitoring, should this be done.

Overall, our results show that not only is our static analysis very fast, it also consistently leads to correct results and a significant reduction of the required enforcement resources. While our evaluation seems to discourage the use of more sophisticated analyses, we still think it is a worthwhile research direction. The optimal solution must balance the costs of more duplication and the monitoring overhead. In the next section we present related works and then conclude this paper in Section VI.

V. Related Work

In this section, we discuss works related to ours. Since there are few works specifically targeting Secure Multi-Execution on machine code, we include the broader concept of noninterference enforcement.

a) Static Approaches: A prominent tool for static enforcement of noninterference is JOANA [13]. It uses Program Dependence Graphs (PDGs) to reduce information flow analysis to reachability. The authors note that the construction of such graphs is absolutely nontrivial, yet the tool provides impressive results for Java code. Due to the focus on syntactical analysis, it may raise false positives. Unfortunately, the approach is also not trivially transferable to machine code [14].

The approach by Balliu et al. accounts for this, by using semantics-based analysis [15]. Instead of enumerating all possible inputs, they enumerate all possible paths of a program using symbolic execution through BAP [16]. Silent actions are removed from the traces and they are combined into an Symbolic Observation Tree (SOT), similar to the interaction trees introduced by [17]. Using an SMT solver, the SOT is compared to a renamed version of itself in a manner similar to the self-composition introduced by Barthe et al. [18]. While the approach gives precise results and works on machine code, it suffers from the typically poor scalability of symbolic execution. As a result, the case studies use less than 100 lines of assembly code.

b) Dynamic Approaches: As a dynamic counterpart to the approach by Balliu et al. [15], Kwon et al. proposed a dynamic system for causality inference [19]. Based on Dual-Execution [20], they run a program twice in synchronization and alter distinct inputs to assess variability at outputs. Their approach scales better and uses more precise semantics-based analysis. However, their technique requires source-code analysis and does not repair noninterference violations. In contrast to this, Shadow Execution [21], a predecessor to Secure Multi-Execution [5], actually repairs information leaks as they occur. However, their approach requires not only the duplication of the process but of the complete system. Similarly, ThightLip [22] provides a form of multi-execution with various repair options. However, as they do not have knowledge about the future use(s) of scrubbed information, they cannot guarantee transparency. Finally, Schmitz et al. recently proposed Faceted Secure Multi-Execution [9], a more efficient extension for SME based on Multiple Facets [23]. They use a barrier-synchronization scheme, combined with termination-sensitive timeouts and provide an implementation in Haskell. As their technique requires changes to the language semantics, it is not applicable to hardware-defined machine code.

VI. Conclusion

In [10], we presented our solution to confidentiality enforcement for machine code. Our goal is to make noninterference enforcement practical through our more efficient Demand-Driven Secure Multi-Execution (DDSME) approach. By implementing our approach on machine code level, we additionally demonstrate the wide-range applicability. The key idea is to restrict the expensive Secure Multi-Execution enforcement to *critical sections* in the code. A limiting factor for both efficiency and transparency of our approach lies in the precision of these critical section. In this paper, we introduced a novel, fully automatic approach to extract sound critical sections from binaries. To achieve this, we focused on the reachability relation implied by the control-flow graph. As the control-flow graph of binary code is usually context-insensitive, a straightforward analysis would result in unnecessarily large critical sections. To counter this, we introduced a fast context-sensitive approach. The efficiency of the analysis comes from a purely graph-based method that does not require tracking of a call stack. We used the results to evaluate our DDSME engine and have shown that the extracted critical sections are correct and lead to a significant increase in performance. In the future, our analysis could be improved through the use of more advanced binary analyses. For example, feasibility analysis may be used to create stricter boundaries. Ideally, static information flow analysis could be used to predict future uses of inputs, which could be used to leave the critical section early. However, both techniques are quite challenging and probably require program-wide points-to analysis, which is highly nontrivial. Our results also show that the ideal size of the critical section depends on the use case.

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Security and Privacy, 1982 IEEE Symposium on*. IEEE, 1982, pp. 11–11.

[2] N. Bielova and T. Rezk, *A Taxonomy of Information Flow Monitors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 46–67. [Online]. Available: https://doi.org/10.1007/978-3-662-49635-0_3

[3] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, vol. 3672. Springer, 2005, pp. 352–367.

[4] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[5] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 109–124.

[6] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," *Journal of Computer Security*, vol. 24, no. 1, pp. 39–90, 2016.

[7] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing-and termination-sensitive secure information flow: Exploring a new approach," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 413–428.

[8] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 1, pp. 175–205, 2006.

[9] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, "Faceted secure multi execution," 2018.

[10] T. Pfeffer, T. Göthel, and S. Glesner, "Efficient and precise information flow control for machine code through demand-driven secure multi-execution," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '19, no. 12. New York, NY, USA: ACM, März 2019, pp. 197–208.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[12] T. Pfeffer, P. Herber, L. Druschke, and S. Glesner, "Efficient and safe control flow recovery using a restricted intermediate language," in *VSC Track on Validation of Safety critical Collaboration systems at the IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2018)*. IEEE Computer Society, Juni 2018, pp. 235 – 240.

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

[14] V. Srinivasan and T. Reps, "An improved algorithm for slicing machine code," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 378–393.

[15] M. Balliu, M. Dam, and R. Guanciale, "Automating information flow analysis of low level code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1080–1091. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660322

[16] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[17] D. Zanarini, M. Jaskelioff, and A. Russo, "Precise enforcement of confidentiality for reactive systems," in *2013 IEEE 26th Computer Security Foundations Symposium*, June 2013, pp. 18–32.

[18] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 2004, pp. 100–114.

[19] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 503–515. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872395

[20] D. Kim, Y. Kwon, W. N. Sumner, X. Zhang, and D. Xu, "Dual execution for on the fly fine grained execution comparison," *SIGPLAN Not.*, vol. 50, no. 4, pp. 325–338, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694394

[21] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla, "Preventing information leaks through shadow executions," in *2008 Annual Computer Security Applications Conference (ACSAC)*, Dec 2008, pp. 322–331.

[22] A. R. Yumerefendi, B. Mickle, and L. P. Cox, "Tightlip: Keeping applications from spilling the beans." in *NSDI*, 2007.

[23] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178.