TimeInspector: A Static Analysis Approach for Detecting Timing Attacks

Fatih Durmaz[§], Nureddin Kamadan[§], Melih Taha Öz[§], Musa Unal[§],
Arsalan Javeed, Cemal Yilmaz, and Erkay Savas
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
{*fatih.durmaz, nkamadan, melihoz, musa, ajaveed, cyilmaz, erkays*}*@sabanciuniv.edu*

***Abstract*—We present a static analysis approach to detect malicious binaries that are capable of carrying out a timing attack. The proposed approach is based on a simple observation that the timing attacks typically operate by measuring the execution times of short sequences of instructions. Consequently, given a binary, we first construct the control flow graph of the binary and then determine the paths between the pairs of time readings, on which a suspiciously low number of instructions might be executed. In the presence of such a path, we mark the binary as potentially malicious and report all the suspicious paths identified. In the experiments, where a collection of benign and malicious binaries were used, the proposed approach correctly detected all the malicious binaries with an accuracy up to 99.5% and without any false negatives.**

***Index Terms*—timing attacks, side-channel attacks, static program analysis, malware analysis**

1. Introduction

Side-channel attacks leverage the information that is unintentionally leaked by a system, such as execution time, power consumption, and cache access behavior [1]–[3], to exfiltrate secret information processed by the system. These attacks are generally quite difficult to mitigate as, in many of the attacks, neither the algorithms nor the implementations actually leak any information. However, when the systems are forced to share some software and/or hardware resources (e.g., by running multiple systems on the same computing platform), they suddenly start leaking information.

A prominent type of side-channel attacks, which is also the focus of this paper, is the timing attacks [4]–[6]. Timing attacks typically operate by measuring the execution times of short-running operations. The differences between the measurements are then associated with secret information, such as the private key processed by a cryptographic algorithm [7].

Cache-based timing attacks, such as Flush+Flush [8], Flush+Reload [9], and Prime+Probe [10], as well as transient execution attacks, such as Meltdown [11], which incorporate cache-based timing attacks, exploit timing variations in memory accesses to determine if the data associated with a memory address of interest is in the cache. To this end, the time is read before and after the memory access and the difference is attributed to the operation. Since the time required to fetch the requested data from the DRAM is significantly more than fetching it from the cache, smaller execution times would indicate that the data is currently in the cache. This information can then be used in various different ways to exfiltrate secret information. For instance, in Meltdown [11], it is used to read a secret value stored at a memory address, which the attacker should not normally be able to access. More specifically, the secret value is used as an index into an array owned by the attacker. Although the access will eventually result in a segmentation fault, due to the out of order execution, the error occurs after the respective array entry has been brought to the cache. Therefore, figuring out which entry of the previously flushed-out array has been brought to the cache, would reveal the secret value.

In previous work, we have developed a generic approach, called *Detector*$^+$, aiming to detect, isolate, and prevent timing attacks [12]. The proposed approach is, indeed, based on a simple, yet quite effective observation: In a timing attack, the attacker typically needs to carry out (possibly repeatedly) a pair of successive time readings in a short period of time. Therefore, Detector$^+$ intercepts the time readings initiated by the processes. If two consecutive time readings happen to be suspiciously close to each other in time, some random noise is introduced into the readings to prevent any potential ongoing attack. In another work [13], we have developed *HyperDetector*, demonstrating that the same idea can also be effectively and efficiently utilized in virtualized environments by intercepting the time reading requests, including the executions of the *rdtsc* machine instruction, at the level of a hypervisor. In the empirical studies, the aforementioned approaches detected all the malicious time measurements with almost perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives.

Both Detector$^+$ and HyperDetector are, however, dynamic program analysis-based approaches. From this perspective, they are reactive approaches as they let potentially malicious applications to run and then monitor and analyze the executions at runtime with the goal of mitigating the harmful effects of the attacks. In this work, we present a proactive and complementary static analysis approach, called *TimeInspector*, to detect potentially malicious binaries before these binaries are even executed. At a very high level, given a binary, TimeInspector first constructs the control flow graph (CFG) of the binary.

[§]These authors contributed equally

Then, the paths between all pairs of times readings are determined. Next, the number of machine instructions occurring on these paths are counted. Finally, the paths, which contain suspiciously few instructions between two time readings, are marked as potentially malicious.

We empirically evaluated the TimeInspector by conducting experiments on both benign binaries and the binaries belonging to a wide spectrum of timing attacks, namely sweep-counting attacks [14], loop-counting attacks [15], Meltdown [11], Flush+Reload [9], Flush+Flush [9], Prime+Probe [10], and Evict+Reload [16]. In these experiments, the proposed approach correctly detected all the malicious binaries with an accuracy up to 99.5% and without any false negatives. We have also successfully tested TimeInspector on a family of malware, called *GuLoader*, which employs timing-based evasion checks [17].

The remainder of the paper is organized as follows: Section 2 presents the threat model addressed in this work; Section 3 introduces the approach; Section 4 discusses the empirical studies carried out to evaluate the proposed approach; Section 5 presents related work; and Section 6 concludes with some potential future work ideas.

2. Threat Model

In the threat model addressed by this work, an adversary provides a piece of malicious code to be executed on a victim system. Although the malicious code could be in the form of a source code and/or a binary code (including the machine-agnostic binaries, such as bytecodes), we, in this work, solely focus on the latter types of codes as it represents more practical attack scenarios. The proposed approach, however, is readily applicable to the former types of codes, too. In either case, the malicious code is made available to the victim system before the execution and the adversary cannot prevent the victim system from analyzing the instructions in a static manner.

The malicious code aims to exfiltrate information, which is indirectly leaked by the system through a timing-based side-channel. The side-channels of interest are formed by measuring the execution times of some short-running operations. To this end, the measurements are made by reading the time before and after each operation of interest and attributing the difference to the operation. All the malicious time measurements are made locally on the victim system by using the timing primitives supported by the system, such as by using the `rdtsc` (or similar) instructions or by making system calls to read the time. Furthermore, the instructions to be executed to carry out the operations of interest are a part of the malicious code.

Consequently, the remote timing attacks where the time measurements are made remotely [18] and the attacks where the operations, the execution times of which are measured for malicious intents, are carried out remotely [19], are out of the scope of this work. Note that, in the former case, the time readings are performed remotely while the instructions being timed may be executed locally. In the latter case, however, although the time readings may be performed locally, the instructions are executed remotely. In either case, the number of instructions executed in between two consecutive time readings may not be determined locally. Similarly, the timing attacks, which operate by measuring the execution times of long-running operations (i.e., the ones, which require a large number of instructions to be executed), such as measuring the time it takes to parse a JavaScript file or a media file to predict the size of an external resource, are out of the scope [19].

Furthermore, we, in this work, assume that the time measurements are carried out by using the native timers provided by the victim system (e.g., by using the timing primitives provided by the underlying system), so that the places in the code where the time is being read can automatically be identified. If, however, the time measurements are obtained by using implicit (e.g., non-native) and often non-clock-based timing sources [20], then TimeInspector may not detect the presence of the time measurements. Interestingly enough, though, we believe that the proposed approach could also be adapted to detect the implementations of certain types of implicit timing sources, which we leave as a future work. Further discussion on the subject can be found in Section 4.4.

3. TimeInspector

TimeInspector takes as input a binary and determines whether the binary is likely to carry out a timing attack or not. In the case of a suspicion, the execution paths (i.e., the sequences of machine instructions) that form the basis for the suspicion, are reported, so that the binary can later be analyzed in a focused manner.

An integral part of TimeInspector is to construct the CFG of a given binary. For this work, without losing the generality of the proposed approach, we use `radare2` [*] – a framework for reverse-engineering and analyzing Linux binaries. Note, however, that the proposed approach is readily applicable to any operating system.

We first construct the CFGs in an intra-procedural manner, such that one CFG is constructed per module, e.g., function. Each node in these CFGs corresponds to a *basic block* – a consecutive sequence of machine instructions, which are all guaranteed to be executed once the first instruction in the sequence is executed. Consequently, branching instructions in basic blocks can appear only as the last instructions of the blocks. We further augment the CFG of a module with a special *entry* and *exit* node, representing the entry and exit of the module, respectively. The references made into statically as well as dynamically linked libraries are also resolved during the construction of the CFGs. Figure A.1a presents a simple CFG, summarizing the attack loop of Flush+Flush [8].

Once the initial CFGs are constructed, we determine the nodes that contain an instruction for reading time, e.g., the `rdtsc` instruction, or that contain a function call, e.g., the `call` instruction. We then split these nodes into multiple nodes, such that every node in the resulting CFG represents a basic block containing a single time reading instruction, or a single function call, or a sequence of other instructions, but not a mixed of them.

We, in particular, represent each time reading instruction in a node of its own, so that the problem of identifying the execution paths between two time readings can be expressed as a reachability problem in graph

[*] Radare2, https://github.com/radareorg/radare2

theory. Similarly, we choose to have a separate node for each function call, so that we can carry out an inter-procedural analysis by linking these nodes to the CFGs of the respective callees. Figure A.1b presents an example where the original CFG in Figure A.1a is modified by splitting node A into 5 nodes ($A_1, \ldots, A_5$), such that each `rdtsc` instruction has its own separate node. Note that, from the perspective of the flow of control, both CFGs are semantically identical.

For the analysis, we first determine the function calls that can directly or indirectly read time. In the remainder of the paper, the nodes having such calls are referred to as *time-reading calls*, whereas the nodes that directly read the time with the help of a machine instruction, such as `rdtsc`, are referred to as *time-reading instructions*.

TimeInspector operates by determining the minimum number of instructions that can be executed between each pair of time readings, so that the paths, which can potentially attempt to measure the execution times of suspiciously few instructions, can be determined. Consequently, we cast the problem to that of finding the shortest path between two nodes (i.e., between two time-reading nodes) in directed weighted graphs (i.e., in CFGs), where the *length of a path* is defined as the number of instructions to be executed on the path.

More specifically, we determine all the ordered pair of nodes (u, v) in the CFGs, such that u is a time-reading instruction and v is either a time-reading instruction or a time-reading function. The reason for the distinction between the types of the source and the target nodes, will become clear shortly. Next, we eliminate the pairs where v is not reachable from u as the respective time readings may not be used for time measurements. We then compute the shortest path for each remaining pair.

To this end, we define the weight of an edge from node z to w as the number of machine instructions included in the basic block represented by w. If, however, w represents a call to a function $func$, the weight is computed as the minimum of the length of the shortest path from the entry node of $func$ to an exit node and the length of the shortest path from the entry node to a time reading instruction (if any), which can directly or indirectly be executed by $func$. Furthermore, the weight for an incoming edge to a function call is computed recursively by traversing all the direct and indirect calls made by the function.

Once the weights are determined, we use a modified version of the Dijkstra's weighted shortest path algorithm to find the shortest path between two time-reading nodes. Note that since the weight for a time-reading call is associated with the incoming edges, to be able to correctly compute the path lengths, we make sure that the source nodes are always time-reading instructions (rather than time-reading calls), while the target nodes can be any type of time-reading nodes. That is, given the way we compute the weights, using a time-reading call as a source node would not account for the instructions to be executed in the call after the time has been read.

One modification we make to the Dijkstra's algorithm is due to the fact that the source and target nodes (u and v, respectively) may not necessarily be distinct in our case. For example, a single time reading can be carried out in a loop. In such a case, there will be a path from the time reading node to itself through the loop (i.e., $u = v$). Therefore, even if the source and the target nodes on a path are the same, the path is capable of making a legitimate time measurement. When $u = v$, the Dijkstra's shortest path algorithm, on the other hand, assumes that the length of the shortest path is 0, which, in our case, is misleading. To overcome this issue, although we start with u and, as a first step, use the weights of the edges to the adjacent nodes as the lengths of the best paths so far discovered to these nodes, we assume u is at distance ∞ from itself and never mark u as visited throughout the search. Therefore, the shortest path from u to itself via other nodes keeps on updated as needed until the search is terminated.

Once the lengths of the shortest paths between every pair of time-reading nodes of interest are determined, we mark the ones that are smaller than a predetermined cutoff value (in our case, $cutoff = 44$) as *suspicious*. And, in the presence of at least one suspicious path, the binary under analysis is marked as *potentially malicious* and all the suspicious paths identified are reported. For this work, we determine the cutoff value by analyzing the benign binaries. As the cutoff value, we use the smallest path length observed, which is greater than the 5% of all the shortest paths obtained from the benign binaries.

4. Experiments

To evaluate the proposed approach, we have carried out a series of experiments.

4.1. Subject Binaries

In these experiments, we used 749 benign binaries from various domains, including systems, networking, and persistent storage, together with 11 malicious binaries. We, in particular, chose the aforementioned benign binaries because they represent the binaries commonly found in modern Linux systems. Furthermore, the malicious binaries used in the experiments implemented various flavors of the sweep-counting attacks [14], loop-counting attacks [15], Meltdown [11], Flush+Reload [9], Flush+Flush [21], Prime+Probe [10], and Evict+Reload [16]. All of the malicious binaries were obtained by using the publicly available implementations of the aforementioned attacks.

4.2. Operational Framework

We first constructed the initial CFGs by using `radare2`, where the external references made into the shared objects were resolved with the help of `ldd` [22]. Next, we determined all the time reading instructions in all the binaries. To this end, we searched both for the `rdtsc` (or similar) machine instructions and the system calls as well as the functions that read time, including `gettimeofday`, `system_clock::now`, `timespec_get`, and `time`. We then modified the CFGs, such that each time-reading instruction or a function call is included in its own separate node. Finally, we identified the weighted shortest paths between each pair of time readings as explained in Section 3. All the experiments were carried out on an Intel Xeon E5606

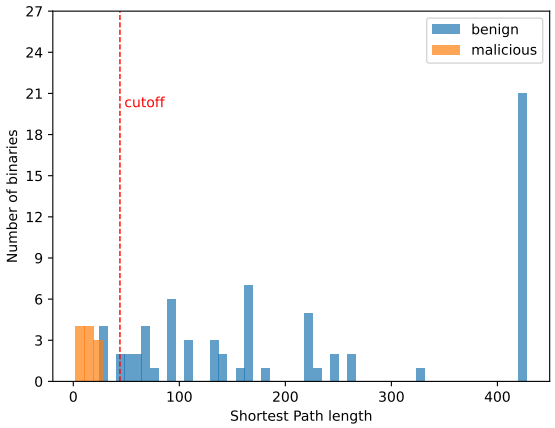


Figure 1: Histogram of the shortest path lengths obtained from the benign and malicious binaries.

2.13GHz GHz machine with 32 GB of RAM running 20.04.1-Ubuntu with kernel version 5.13.0 and `radare2` version 5.6.9.28082 (commit 34ce17da).

4.3. Data and Analysis

We first observed that 677 out of 749 benign binaries did not contain any instructions or functional calls for making time measurements. The remaining 72 benign binaries, on the other hand, had at least one path between two time readings in their CFGs. This observation is, indeed, well-aligned with our previous observations [12], [13] that applications operating in the production environments often do not need to measure the execution times. That is, for many applications, even an attempt to make a time measurement, can, indeed, be considered to be suspicious, especially when measuring time is not required for the application to operate.

We then analyzed the distributions of the shortest path lengths obtained from the benign and malicious binaries. Figure 1 visualizes the results we obtained where the benign binaries without any time measurements are excluded for visualization purposes.

We first observed that the maximum shortest path length obtained from the malicious binaries was 28. For example, the Flush+Flush attack [8] in Figure A.1 measures the execution time of the `clflush` instruction by using two `rdtsc` instructions with 6 other instructions in between, i.e., with a shortest path of length 6.

We then observed that using 44 as the cutoff value as explained in Section 3, correctly identified all the malicious binaries, i.e., with 0 false negative, while resulting in 4 false positives for the 72 benign binaries with the time measurements. That is, when only the 72 benign binaries with the time measurements are considered, the accuracy of the proposed approach is 95%. And, when all the benign binaries are considered, the accuracy is 99.5%.

An in-depth analysis of the false positives revealed that all of the false positives were caused by the binaries, which use the same or a similar approach to check for the existence of a particular clock at runtime. More specifically, although it is possible to check the availability of a certain clock statically at compile-time, the aforementioned benign binaries have a simple while loop, which first attempts to read the time by using the `CLOCK_MONOTONIC` timer. In the case of a failure, the timer is changed to `CLOCK_REALTIME` in the body of the loop, such that the aforementioned timer is probed until a successful time reading is obtained. Indeed, three out of four of the binaries, which were falsely marked as malicious, had exactly the same while loop resulting in a shortest path of length 24, while the remaining binary had a similar loop, resulting in a shortest path of length 26. Consequently, one way to suppress these and similar false positives is to capture the patterns in such benign uses of the timers and filter them out in the static analysis. For example, besides having exactly the same or similar sequences of machine instructions, one common characteristic among these binaries is that, although there is a path between two time readings, the readings cannot be used for any time measurements as at least one of the readings always results in a failure.

In these studies, we used the reference implementations of the aforementioned attacks for the evaluations. It is, indeed, quite difficult to obtain real-life timing attacks as these attacks typically leave no trace, except possibly for some suspicious and hard-to-interpret set of symptoms in the logs. We could, however, find a family of malware, called *GuLoader*, which employs timing-based evasion checks [17]. More specifically, to figure out whether it is running in a virtual machine or not, this malware measures the execution time of the `CPUID` instruction by using `rdtsc`. In a virtualized environment, as this instruction causes a VM exit, it takes longer time to execute, compared to executing it directly on real hardware. Note that, from the perspective of this work, this attack is still considered to be a timing attack since it uses the execution time of an operation to exfiltrate information that is indirectly and unintentionally leaked by the system under attack. When we, indeed, applied the proposed approach exactly the way it is described in Section 3 on this family of malware, we were able to detect it as the length of the shortest path between two timing readings was 10.

4.4. Discussion

One countermeasure against the proposed approach could be to use an implicit timing source, such that TimeInspector fails to detect the places in the code where the time measurements are carried out. A number of such implicit clocks for the web browsers have been introduced in [20]. Interestingly enough, though, we believe that the proposed approach could potentially be used to detect the implementations of certain types of implicit clocks. For example, one type of an implicit clock aims to recover the high resolution of a native clock, the resolution of which is purposefully decreased by the browser to prevent the timing attacks. These approaches, however, rely on using the low-resolution clock provided by the browsers to detect the clock edges. And, this requires the time to be read repeatedly by, for example, using the browser-provided `performance.now()` function, which would make it suspicious to TimeInspector. To test this conjecture, we implemented the code excerpt given in Listing A.1 of [20] in C by replacing the `performance.now()` calls with `rdtsc` instructions as our implementation of

the proposed approach currently supports only the Linux binaries. TimeInspector, indeed, marked the implementation as suspicious as the length of the shortest path between two time-reading calls, was 5.

Other types of implicit clocks, rather than depending on the availability of a native clock, use certain events that are directly supported by the browsers, e.g., the `onmessage` events and the `postMessage` operations, as the clock edge signal. And, the interval between two consecutive events of interest mimics a "clock cycle" [20], which will be referred to as *virtual clock cycles* in the remainder of the document. In each virtual clock cycle, the value of a counter is incremented, ensuring a monotonically increasing sequence of time readings. Note that, in order to obtain the highest-possible resolution, which is typically required by the timing attacks, nothing except for incrementing a simple counter should be performed in a virtual clock cycle. Consequently, as long as the events that are likely to be used to implement the implicit timing sources are known, the proposed approach could be adapted by checking whether a suspiciously few number of instructions are executed between these events. Indeed, the events of interest used in the reference implementations given in [20], were all messaging-related events.

One type of implicit timing source, which cannot be detected by TimeInspector, is counting threads [23], [24], where a dedicated thread keeps on incrementing a counter, the values of which are used as time readings. Note, however, that it could be possible to develop specialized static analyzers to detect the presence of such timing sources by checking for code segments that do nothing, but increment a counter value in a loop.

Another countermeasure against the proposed approach could be to add no-op (NOP) operations, such that the number of instructions between the readings is increased (eventually to a point above the cutoff value) without affecting the actual time measurements. However, this countermeasure could be addressed by filtering out the NOP (or similar) operations (to the extent to which the identification of these operations are possible in a static analysis) before the path lengths are computed.

A similar countermeasure is to add redundant instructions in between the time readings, the execution times of which can be predicted, so that they can be subtracted from the actual measurements. However, in [12], we demonstrated that using this approach to stay stealthier introduces noise into the measurements, which, in turn, makes it more difficult (if not impossible) for the attacker to carry out the attack. More specifically, we kept on adding an increasing number of redundant operations in the time measurements made by the Meltdown attack [12]. We observed that the more redundant instructions added, the less successful the attack typically became [12]. And, after adding a certain number of redundant operations, the attack was rendered useless, i.e., exfiltrating 0 byte of information. We believe that this is because as the number of instructions executed in between two consecutive time readings increase, the noise in the measurements tends to increase. For example, in the cache-based timing attacks, the more time spent for the measurements the more likely it is for the cache lines evicted by other processes. Or, the more it takes for the instructions to execute, the more likely it becomes for the respective processes to be scheduled out by the operating system in the middle of a measurement. Clearly, making more measurements could help statistically factor out the noise [25]. This, however, would certainly make the attacks more difficult to carry out as the experiments need to be repeated, which could, in turn, also make it easier for the runtime approaches to detect the suspicious activities. After all, the cutoff hyper-parameter of the proposed approach can be increased to address this countermeasure at the cost of a potentially increased false positive rate.

Another countermeasure is to use obfuscated binaries. However, in the case of sensitive time measurements, which are the main concern of this work, the obfuscation needs to be carried in a carefully-considered manner not to introduce excessive noise in the measurements, e.g., excessive number of redundant instructions in between the time readings. After all, we always identify the short paths between the readings.

A related countermeasure is to use packed binaries. As is the case with all the similar static analysis approaches, the proposed approach assumes that the instructions in the binaries can be analyzed. If this is not possible, as is the case in the packed binaries, where certain subsets of the instructions to be executed are compressed and/or encrypted, then the static analysis approaches will certainly suffer. From this perspective, the proposed approach is no exception. Therefore, the proposed approach, as a static analysis approach, is complementary to the dynamic analysis approaches developed for detecting the timing attacks at runtime [12], [13], [26]–[28].

5. Related Work

A variety of static analysis approaches have been proposed in the literature to analyze and mitigate timing-based side-channel attacks [12], [29]–[33]. Doychev et al. present an approach, which analyze program binaries to derive quantitative upper bounds on the amount of information leaked through a number of timing-based and other side-channels due to the program interactions with cache memory [34]. Qin et al. reveal that timing channels could emerge during just-in-time (JIT) compilation of sensitive program segments and present a static analysis approach to generate safe JIT compilation policies [35]. Barthe et al. point out that constant-time implementations of cryptographic routines within the scope of high-level programming languages could provide resilience to timing channels [36]. However, compilation itself could generate unsafe machine code prone to timing exploitations. To this end, they present an approach, which turns an existing compiler into a formally-verified secure compiler against such issues. Jancar et al. [37] conducted a survey on why many mainstream cryptography libraries are found to be vulnerable to timing attacks despite being developed with constant-time implementation principles. The authors found that developers are aware of timing attacks and often implement constant-time code, but there are shortcomings in the criterion, tests, and formal models used to evaluate these implementations. To this end, the authors propose a set of serious recommendations to improve current practices in this area. Cauligi et al. [38] discuss the importance of compiler- and verification-tools to defend against Spectre attacks, which require reasoning about

microarchitectural timing details. Despite existing formal foundations and security guarantees, the authors argue that more work is needed. They systematize current knowledge about software verification and mitigation within this context and propose a framework to study security properties. Furthermore, they outline the foundations upon which a security assessment framework should be built. Skorsten et al. present a secure compilation approach for safeguarding the programs against the attacks on program states and control flows [39]. Specialized domain specific languages (DSLs) have also been proposed to implement constant-time cryptographic primitives, guaranteeing the safety against the timing exploitations even within the context of speculative execution [40], [41]. Vassena et al. present an approach to harden programs that are prone to timing exploitation under speculative execution, by detecting vulnerable data flows and patching them with fence-based calls [42]. Wu et al. present an approach to eliminate timing channels from the source code of a given program through static program repair and transformation [43]. Furthermore, Atici et al. combine static and dynamic analysis to identify the root causes of the information leakage in software systems [44]. Our work is different than these existing approaches in that we identify malicious binaries by developing a novel static analysis approach. The aforementioned approaches, on the other hand, aim to harden the benign binaries by analyzing and mitigating timing-based side-channels.

From this perspective, Irazoqui et al.'s work is, perhaps, the closest work to ours. They present Mascat [45] framework, which at its core utilizes the presence of certain instruction features, which are characteristic to known microarchitecture attacks including timing attacks. The features span across from the presence of certain special instructions to specific instruction patterns, such as whether a cache line is being flushed inside a loop performing the measurements. Depending on the presence of these features, an overall weighted score is calculated, ranking a binary on a scale of guaranteed attack to perfectly benign. TimeInspector, however, follows an orthogonal approach. More specifically, rather than looking for the presence of certain instructions, which can be avoided to remain stealthier, TimeInspector counts the smallest number of instructions to be executed between two time readings, regardless of what these instructions are, with the goal of detecting the timing attacks that operate by measuring the execution times of short running operations.

Dynamic program analysis-based approaches to combat timing attacks have also been under active research. Crane et al. rely on control-flow diversity to offer probabilistic protection [46]. Their approach systematically generates a large number of semantically equivalent but unique execution paths at runtime and frequently switches among these paths to leave a unique execution trace each time. Rane et al. adopt a similar approach, but leverage obfuscation in control flows to perturb branch outcomes as a defense mechanism [47]. Hunt et al. adopt a hardware-enclave based sandboxing approach to carry out the executions of sensitive programs in a distributed manner with the goal of minimizing the effectiveness of the timing attacks [48]. Brasser et al. demonstrate frequent location-randomization as an effective defense mechanism, which obfuscates the locality of data against the adversaries utilizing the timing channels [49]. Using neural networks to discover and quantify the information leakage through timing channels has also been explored [50], [51]. Employment of program fuzzing techniques to discover potential timing vulnerabilities hidden in programs have also received considerable attention [52], [53].

One particular, but focused aspect among dynamic analysis-based approaches is runtime detection, isolation, and prevention of microarchitecture-related timing attacks. In this regard, Akyildiz et al. [26] rely on runtime monitoring of segmentation faults occurring at memory addresses that are close to each other to solemnly countermeasure an ongoing Meltdown attack [11]. Kulah et al. monitor the contentions in L1 cache memory and emit warnings about the presence of potential cache-based timing attacks when the contentions arrive at a suspicious level [27]. In a similar fashion, Chiappetta et al. monitor the contentions in L3 cache memory to detect the patterns demonstrated by known attacks or similar types of cache-based timing attacks to raise alarm upon detecting an ongoing attack [28]. Our work is different than these dynamic analysis approaches as TimeInspector is a static analysis approach. From this perspective, TimeInspector serves in complimentary fashion to its dynamic contemporaries.

Unlike compiled malicious binaries, an attacker can mount remote attacks through embedding malicious JavaScript (JS) code on a web page. Timing attacks carrying out microarchitecture attacks in JS are fairly-recent. Although, browser vendors followed mitigation such as limiting the resolution of timing-APIs however, subsequent research revealed a number of alternative ways to either circumvent aforementioned low-resolutions, or a craft implicit timers of ample resolution to mount timing attacks. To this end, Schwarz et al. [20] presented a number of practical approaches to craft such implicit timers meant for JS attacks. In contrast, Rockici et al. [54] systematize and present efforts which could mitigate and aid to develop effective countermeasure against aforementioned implicit timers. Our presented work is exclusively focused on compiled binaries which employ explicit time measurement routines. However, as we discussed in Section 4.4, we do not neglect a future extension of our presented approach tailored for implicit timer attacks.

6. Concluding Remarks

In this work, we have presented a novel static analysis approach to detect malicious binaries that are capable of carrying out a timing attack. To this end, we construct the CFG of a given binary and identify the paths between the pairs of time readings, on which a suspiciously low number of instructions might be executed. In the presence of such a path, we mark the binary as potentially malicious. The proposed approach, being a static analysis, is complementary to the dynamic analysis approaches that aim to detect, isolate, and prevent timing attacks [12], [13], [26]–[28]. One potential avenue for future research is to develop hybrid approaches where the results of the static analysis are used to determine the parts of the executions to be focused on at runtime to further reduce the runtime overheads while increasing the detection accuracy. Another avenue is to adapt the proposed approach to detect the presence of implicit timing sources.

References

[1] S. Zander, G. Armitage, and P. Branch, "A survey of covert channels and countermeasures in computer network protocols," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 44–57, 2007.

[2] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, 2019.

[3] J. Betz, D. Westhoff, and G. Müller, "Survey on covert channels in virtual machines and cloud computing," *Transactions on Emerging Telecommunications Technologies*, vol. 28, no. 6, p. e3134, 2017.

[4] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[6] C. Percival, "Cache missing for fun and profit," 2005.

[7] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[8] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, J. Caballero, U. Zurutuza, and R. J. Rodríguez, Eds. Cham: Springer International Publishing, 2016, pp. 279–299.

[9] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 719–732.

[10] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[12] A. Javeed, C. Yilmaz, and E. Savas, "Detector+: An approach for detecting, isolating, and preventing timing attacks," *Computers & Security*, vol. 110, 2021.

[13] M. S. Unal, A. Javeed, C. Yilmaz, and E. Savas, "Hyperdetector: Detecting, isolating, and mitigating timing attacks in virtualized environments," in *International Conference on Cryptology and Network Security*. Springer, 2022, pp. 188–199.

[14] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+ probe 1, javascript 0: Overcoming browser-based side-channel defenses." https://www. usenix. org/conference/usenixsecurity21/presentation/shusterman, 2021.

[15] J. Cook, J. Drean, J. Behrens, and M. Yan, "There's always a bigger fish: a clarifying analysis of a machine-learning-assisted side-channel attack," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 204–217.

[16] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[17] J. Security, "Guloader's vm-exit instruction hammering explained," Accessed: 4 May 2023. [Online]. Available: https://www.joesecurity.org/blog/3535317197858305930

[18] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 2011, pp. 355–371.

[19] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1382–1393.

[20] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript," in *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers 21*. Springer, 2017, pp. 247–267.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[22] T. L. D. Project, *ldd(1) — Linux manual page*, The Linux Documentation Project, 2021. [Online]. Available: https://man7.org/linux/man-pages/man1/ldd.1.html

[23] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices." in *USENIX Security Symposium*, 2016, pp. 549–564.

[24] J. C. Wray, "An analysis of covert timing channels," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 219–232, 1992.

[25] E. Ronen, K. G. Paterson, and A. Shamir, "Pseudo constant time implementations of tls are only pseudo secure," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1397–1414.

[26] T. A. Akyildiz, C. B. Guzgeren, C. Yilmaz, and E. Savas, "Meltdowndetector: A runtime approach for detecting meltdown attacks," *Future Generation Computer Systems*, vol. 112, pp. 136–147, 2020.

[27] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "Spydetector: An approach for detecting side-channel attacks at runtime," *Int. J. Inf. Secur.*, vol. 18, no. 4, p. 393–422, aug 2019. [Online]. Available: https://doi.org/10.1007/s10207-018-0411-7

[28] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[29] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[30] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[31] A. K. Biswas, D. Ghosal, and S. Nagaraja, "A survey of timing channels and countermeasures," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–39, 2017.

[32] Q. Zhang, H. Gong, X. Zhang, C. Liang, and Y.-a. Tan, "A sensitive network jitter measurement for covert timing channels over interactive traffic," *Multimedia Tools and Applications*, vol. 78, no. 3, pp. 3493–3509, 2019.

[33] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 360–371.

[34] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on information and system security (TISSEC)*, vol. 18, no. 1, pp. 1–32, 2015.

[35] Q. Qin, J. Jiyang, F. song, T. Chen, and X. Xing, "Dejitleak: Eliminating jit-induced timing side-channel leaks," in *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, Singapore, November 14-18, 2022*, 2022.

[36] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving c compiler," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–30, 2020.

[37] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ""they're not that hard to mitigate": What cryptographic library developers think about timing attacks," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, p. 632–649.

[38] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "Sok: Practical foundations for software spectre defenses," in *2022 IEEE Symposium on Security and Privacy (SP)*, May 2022, p. 666–680.

[39] L. Skorstengaard, D. Devriese, and L. Birkedal, "Stktokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–28, 2019.

[40] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 913–926.

[41] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: A dsl for timing-sensitive computation." New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314605

[42] M. Vassena, C. Disselkoen, K. V. Gleissenthall, S. Cauligi, R. G. Kici, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *arXiv preprint arXiv:2005.00294*, 2020.

[43] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

[44] A. Atıcı, C. Yilmaz, and E. Savas, "An approach for isolating the sources of information leakage exploited in cache-based side-channel attacks," 06 2013, pp. 74–83.

[45] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Mascat: Stopping microarchitectural attacks before execution," *Cryptology ePrint Archive*, 2016.

[46] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[47] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital {Side-Channels} through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[48] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

[49] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 788–800.

[50] S. Tizpaz-Niari, P. Černỳ, S. Sankaranarayanan, and A. Trivedi, "Efficient detection and quantification of timing leaks with neural networks," in *International Conference on Runtime Verification*. Springer, 2019, pp. 329–348.

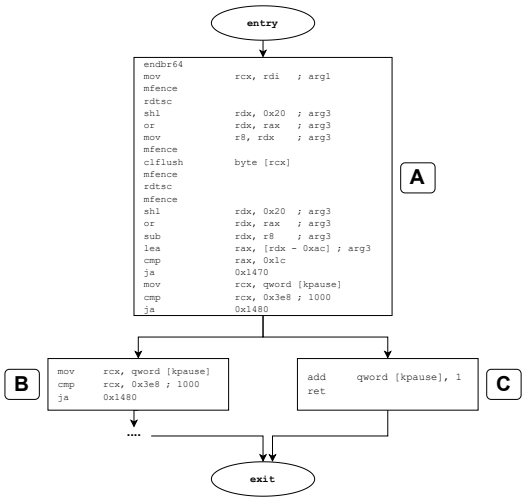
[51] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.

[52] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

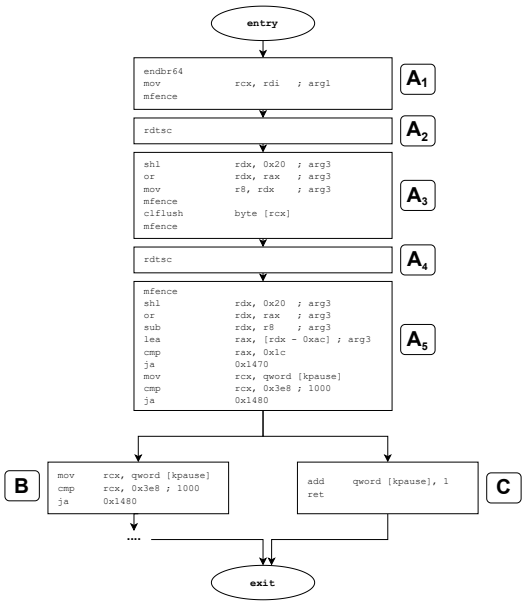
[53] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[54] T. Rokicki, C. Maurice, and P. Laperdrix, "Sok: In search of lost time: A review of javascript timers in browsers," in *2021 IEEE European Symposium on Security and Privacy (EuroSP)*, Sep 2021, p. 472–486.

Appendix



(a) Original CFG



(b) Modified CFG

Figure A.1: Example control flow graphs (CFGs).