BliMe Linter

Hossam ElAtali*, Xiaohe Duan*, Hans Liljestrand†, Meng Xu* and N. Asokan*
*University of Waterloo, †Intel Labs
{hossam.elatali, x34duan, meng.xu.cs}@uwaterloo.ca, hans@liljestrand.dev, asokan@acm.org

***Abstract*— Outsourced computation presents a risk to the confidentiality of clients' sensitive data since they have to trust that the service providers will not mishandle this data. Blinded Memory (BliMe) [1] is a set of hardware extensions that addresses this problem by using hardware-based taint tracking to keep track of sensitive client data and enforce a security policy that prevents software from leaking this data, either directly or through side channels. Since programs can leak sensitive data through timing channels and memory access patterns when this data is used in control-flow or memory access instructions, BliMe prohibits such unsafe operations and only allows *constant-time code* to operate on sensitive data. The question is how a developer can confirm that their code will run correctly on BliMe. While a program can be manually checked to see if it is constant-time, this process is tedious and error-prone.**

In this paper, we introduce the BliMe linter, a set of compiler extensions built on top of SVF [2] that analyze LLVM bitcode to identify possible BliMe violations. We evaluate the BliMe linter analytically and empirically and show that it is sound.

I. Introduction

Outsourced computing offers efficient and stable platforms and services for individuals and organizations to handle data processing tasks. Typically, in this setup, clients entrust service providers with their sensitive data, which is then processed and the results are returned. However, there is a potential security risk because the software underlying outsourced computing infrastructure can be malicious or vulnerable to external threats. Consequently, client data can be exposed through run-time attacks or more concealed leakage like side-channel attacks. Addressing this concern about the leakage of sensitive data is a significant challenge in outsourced computing.

Many approaches have been proposed to mitigate data leaks in outsourced computing [3]–[6]. One state-of-the-art solution is Blinded Memory (BliMe) [1], which is a set of hardware extensions that enforce a taint-tracking policy to ensure data confidentiality. The policy monitors data that depends on secrets and triggers a fault if an instruction may reveal secret-dependent data to external observers, whether through run-time attacks or side channels. BliMe ensures that decrypted data on the server-side is always tagged as tainted (including in registers, caches and memory) and is tracked by the hardware, thus preventing potential leaks from vulnerable or malicious server-side software. Unlike fully-homomorphic encryption, BliMe allows the processor to conduct computations directly on decrypted data after importing it, avoiding the high performance overhead of operating on encrypted data.

While BliMe hardware provides efficient and secure outsourced computation, its stringent policy aborts any secret-dependent branching or memory access instructions to prevent adversaries from inferring secrets by monitoring execution times. This requirement presents a usability challenge because most software is not designed to be constant-time or BliMe-compliant. In order for such software to run on BliMe, its executable needs to be adapted into a form that avoids secret-dependent control-flow branches and memory accesses. Since manually identifying potential data leaks is a complex and error-prone task, a compiler-based tool is needed to automatically identify potential violations in source code.

In this paper, we introduce the BliMe linter, a set of compiler extensions that analyze LLVM bitcode to identify possible BliMe violations. At the core of our constant-time code linter is a taint-tracking engine that propagates taint statically but with the same policy as enforced by the BliMe hardware at runtime. To detect information flows within the program, we build on a state-of-the-art static analysis tool, SVF [2], that uses value flows to track relations in programs (Section II-C). SVF alone, however, is not enough to detect all BliMe violations. We discuss the reasons for this, as well as how we improve the analysis to account for it, in Section III-C1. Concretely, our contributions are as follows.

1) The BliMe linter, a set of compiler extensions to identify potential BliMe violations in LLVM bitcode (Section III)[1] .
2) An evaluation of the BliMe linter on the oblivious instruction set architecture (OISA) benchmarks and TensorFlow Lite (Section IV).
3) Two case studies that demonstrate the effectiveness of the BliMe linter in identifying the root cause of violations (Section IV-A1).
4) A discussion of challenges and possible improvements for future work, including automatic transformations of non-compliant code (Section V).

II. Background

A. Side channels

Side channels capture side effects of program execution which are not observed through the intended media. Examples of central processing unit (CPU) side channels include execution time, memory access patterns, microarchitectural state (e.g., caches), voltage and electromagnetic radiation [7]–[12]. Side-channel leakage occurs when an adversary can deduce information about secret data by observing the system while it is being processed. One possible cause is vulnerabilities in the underlying hardware, e.g., Meltdown [13]. Another is

[1]Open-sourced at https://github.com/ssg-research/BliMe-linter

secret-data-dependent behavior by the program. For instance, if a program's execution time varies depending on a sensitive value used in a conditional branch instruction, an adversary could infer details about that value by monitoring the branch's completion time. Another scenario is when a sensitive value is used to access an array in memory, causing a change in the memory access pattern, which can be observed through shared caches or memory buses.

Performance optimizations common in modern CPUs can further amplify side-channel leakage [13]–[23]. This behavior can occur transparently under-the-hood, making it difficult for developers to identify vulnerabilities in programs.

B. Constant-time code

Constant-time programming is a programming paradigm widely used in cryptographic libraries to mitigate side-channel leakage. The idea is to prevent sensitive data from being used in ways that affect execution time, which can manifest in two ways: control flow and data flow. The techniques employed to prevent such leakage are called control-flow linearization (CFL) and data-flow linearization (DFL), respectively.

For control flow, state-of-the-art solutions use a program counter security model (PC-security) [24]. PC-security states that the program counter cannot become sensitive-data-dependent at any point during the execution of the program. This effectively means that an adversary cannot use a trace of the program's execution to infer sensitive data values. PC-security prevents using sensitive data to select conditional branches or determine whether a fault occurs (e.g., division by zero). CFL attempts to provide PC-security by executing both paths of a sensitive conditional branch (a real path and a decoy path), while maintaining correct functionality of the program. *Predicated execution*, a form of CFL, does this by maintaining a predicate throughout execution that represents whether this path is a real or decoy path. Every sensitive operation is then masked with this predicate to ensure that it only changes the program state if it is on the real path. *Transactional execution*, another form of CFL, executes both paths as-is but attempts to buffer and then discard state updates from decoy paths.

PC-security alone, however, is not sufficient. Sensitive-data-dependent data flows can also affect execution time. For example, some CPU implementations of floating point instructions have a variable number of cycles that depends on the operand values. If sensitive values are used as operands to such instructions, the execution time (in cycles) can leak the values to an adversary. Furthermore, memory accesses can have different latencies depending on whether the requested address is cached. A leak can therefore also occur if the addresses of memory accesses are sensitive. One simple DFL technique for instructions with a variable number of cycles is software emulation; the unsafe CPU instruction is replaced with a safe instruction sequence that performs the same functionality. For memory accesses, DFL is more difficult. The access latency must be indistinguishable for the set of all possible addresses that could be accessed at this point in the program execution. For arrays, one way to achieve this is simply by fetching the entire array into the cache before each access. This ensures that each access, irrespective of the sensitive address, will always have the cache-hit latency. Another more general approach is to identify, for each memory instruction, the set of all possible memory addresses that can be accessed by the instruction, and simply access the entire set, masking away all values except the desired one.

C. SVF

To facilitate constant-time programming, compilers must determine which values are sensitive and how they affect or propagate to other values. One way to do this is via static program analysis, specifically value flow analysis. Value flow analysis resolves dependencies between variables in the program. SVF [2] is a state-of-the-art value-flow analysis tool. It splits the analysis into two steps that are performed iteratively: pointer analysis, and value-flow construction. Pointer analysis determines the set of locations each pointer can refer to. Value-flow construction takes the results of the pointer analysis and uses it to create a sparse value-flow graph (SVFG), which represents the dependencies between values as a directed graph. If further precision is required, the SVFG can be fed back to the pointer analysis step to improve its precision, resulting in further improvement to the SVFG.

D. BliMe

BliMe enables clients to securely transmit sensitive data to a server for processing, ensuring that the data remains confidential and is not leaked directly or via side channels. It comprises a minimal set of hardware extensions and an attestation architecture. This setup allows a client to send conventionally encrypted data to a remote server, which the processor can decrypt and process without allowing the data or any derived information to be leaked. Results are only returned as ciphertext after encrypting them with the client's key.

In BliMe, the processor prevents client data from being exported from the system by enforcing a taint-tracking policy. Secure import and export of data between clients and servers is provided by using a hardware security module and an encryption engine, which ensure that decrypted client data on the server is always tainted. This allows BliMe to provide its security guarantees *without having to make any trust assumptions about the server code*. Therefore, BliMe ensures client data confidentiality even against run-time attacks and malicious server code.

BliMe raises a fault when there is any attempt by software to leak blinded data. As a result, programs must be "BliMe-compliant" to run without being halted by BliMe.

1) BliMe-compliant vs. Constant-time: BliMe enforces its policy on a per-instruction basis and does not have a high-level view of program semantics. Therefore, it enforces a stricter form of constant-timeness compared to that discussed in Section II-B. For example, traditional constant-time code allows branching on secret data as long as both branches are balanced and no timing difference can be detected between their executions. Another example is when software strides

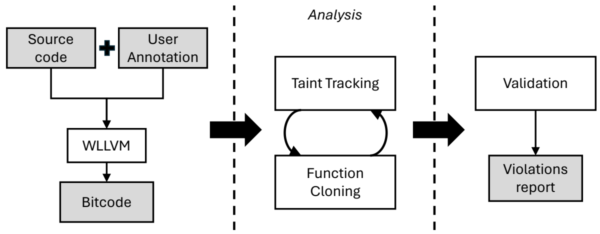

Fig. 1. The BliMe Linter Overview.

arrays with cache-line-sized step and a secret offset within each cache line (Section V-D1). In both cases, data confidentiality guarantees stem from the software and the developer must reason about the software's security to provide these guarantees. BliMe, on the other hand, does not rely on trust in software to provide its security guarantees, and therefore must take a conservative approach. As a result, both cases discussed above are prohibited by BliMe.

2) BliMe vs. compiler taint tracking: Taint tracking in BliMe is dynamic and fully precise. The hardware can track exactly how tainted values are used and how the taint propagates in registers and memory. Consequently, BliMe precisely knows the taint of a program in its current execution state. On the other hand, static taint analysis attempts to determine the taint for all possible execution states at a specific program location. This is an undecidable problem [25], and therefore compilers must make approximations to keep the problem tractable. These approximations must be conservative, which can lead to an over-approximation of the final tainted state. We discuss how the analysis can be made more precise in Section V-B.

III. Design & Implementation

The main requirement for the BliMe linter is that the analysis must be sound but not necessarily complete. This means that the analysis may fail to mark certain BliMe-compliant code as safe, but code reported as safe by the linter must have no violations on the actual BliMe hardware.

The high-level design of the BliMe linter is shown in Figure 1. The input to the BliMe linter is the source code, annotated to mark which data is sensitive, e.g., a secret key. Internally, the BliMe linter is divided into two steps (Analysis and Validation). The Analysis step uses the sensitive data annotated by the developer as *taint sources* and performs static taint-tracking analysis to identify where the taint can propagate throughout the program. The Validation step then identifies all locations in the program that might use tainted data in a non-BliMe-compliant manner, and creates a violations report. Note that since the analysis is not complete, the presence of violations in the final report does not necessarily mean that the executable will not run on BliMe; the violations might be a result of over-approximations of the analysis as discussed in Section V-B. On the other hand, if the violations report is empty, the executable is guaranteed to run on BliMe.

In the following subsections, we detail each part of our design.

```
__attribute__((blinded)) char x = 123;
__attribute__((blinded)) char *y = new char[10];
```

Listing 1. Examples of using the `blinded` attribute.

A. Developer Interface

Since the BliMe linter relies on developer annotations to identify sensitive data, we introduce a new C++ Clang attribute (`blinded`) that the developer can attach to variables in the source code. The attribute works similar to the C/C++ type qualifiers such as `const` and can similarly be applied to primitive types and complex types such as structures and pointers. The example in Listing 1, shows a blinded primtive type (`char x`) and a pointer to an array of blinded characters.

We have extended the Clang front-end to recognize the `blinded` attribute and pass them on in the lowered LLVM intermediate representation (IR), which is then passed to the Analysis step. As the LLVM compiler pipeline may drop unknown attributes, we schedule our analysis pass before other LLVM IR passes that may discard them.

B. Whole-program LLVM

When compiling C++ projects with more than a single `.cpp` file, build systems usually compile each file to a separate object file and then links all object files together at the end to produce an executable. Thus analyses or optimizations *across* modules at the *LLVM IR* level are not possible. To solve this issue, we use whole-program LLVM (WLLVM) [26]. During compilation, WLLVM generates LLVM bitcode for each compilation module and adds it to a custom ELF section in the corresponding object file. When the final executable is linked, WLLVM concatenates all bitcode sections from the linked files and adds them to a section in the ELF executable. A WLLVM tool can then be used to extract and link the bitcode. The result is a bitcode file that represents the entire program. This allows us to easily analyze the entire program (i.e., across modules) using a single input bitcode file.

C. Analysis

As shown in Figure 1, the Analysis step consists of two parts that are also performed iteratively: Taint Tracking and Function Cloning. Taint Tracking uses static value-flow tracking to propagate taint from the taint sources to the rest of the program. If there are any functions with newly tainted arguments, they are cloned, and the compiler then start another round of Taint Tracking. If there are no functions with newly tainted arguments, the compiler exits the Analysis step.

1) Taint Tracking: We adapt the state-of-the-art SVF [2] tool to perform static value-flow tracking in the Taint Tracking step. From SVF, we first obtain a SVFG, which shows the dependencies between values. We then traverse the SVFG breadth-first and propagate taint from the taint sources to the rest of the program along the edges of the SVFG.

a) Implicit Flows: There is one case, however, which is not covered by SVF: *implicit flows*, which are information flows from a condition to value assignments that depend on this condition. As a *partial* remedy for this gap, we propagate taint for LLVM `select` instructions. We use LLVM's internal def-use chains to detect such cases and propagate taint from the condition to the output. However, the BliMe linter intentionally does not handle all implicit flows as this would produce many warnings, making it harder to identify the root branching violation. Instead, we rely on the developer using an iterative "find-and-fix" approach until the BliMe linter reports no violations (indicating that the program is now BliMe-compliant). We discuss this further in Section IV.

2) Function Cloning: We introduce function cloning to improve the context-sensitivity of the analysis. A single function can be called from many call sites in the code. When taint propagates to the arguments of this function, we must track the taint within the function and, if the return value is tainted, propagate the taint back to the call site. However, doing this without function cloning can cause two issues:

1) **Return value over-tainting:** If a function returns a tainted value at only a subset of its call sites, tainted return values will incorrectly propagate taint back to all call sites, even those not within this subset. For example, if a simple `add` function is called with tainted arguments at site ⓐ and untainted arguments at site ⓑ, taint will propagate back to the return value at both sites. With function cloning, the function called at ⓐ will be a taint-propagating clone of that called at ⓑ, and only the return value at ⓐ would become tainted, as expected. Another example is when two calls use tainted arguments, but only one should have a tainted return value.
2) **Forward call over-tainting:** The situation described above can also occur for forward calls. A function called with tainted arguments can propagate these arguments to other functions calls within it. Consider the `add` example above. Without function cloning, if the `add` function calls another function, e.g., `copy`, with `add`'s tainted arguments, every call to `copy` will result in taint propagation, significantly increasing unnecessary taint.

One drawback to function cloning is the increased code size caused by the additional functions. However, this is a reasonable trade-off considering the issues discussed above.

IV. Evaluation

To evaluate the soundness of the BliMe linter, we first analyze the soundness of SVF. While the authors [2] do not claim any soundness guarantees, we believe it is safe to assume that SVF is sound with respect to *explicit value flows* as per its design. Therefore, up until the first time tainted data flows into a condition (e.g., the condition in an if-then-else, or the condition in a select instruction), our analysis is sound. Beyond this point, we can no longer provide soundness guarantees. In other words, we make no claims that if violations are reported, that they are the *only* violations. However, once a developer transforms a violating condition to a BliMe-compliant form, and reruns the analysis, our soundness guarantees will hold further along the program's execution paths. Through an iterative process, we can therefore claim soundness as follows: if the BliMe linter does not report any violations, then the program is guaranteed to work on BliMe.

		Linter Results		Spike Results	
Program	LOC	Memory	Branch	Memory	Branch
binary_search	34	1	3	0	2
dijkstra	188	6	5	5	1
dnn	77	0	0	0	0
find_max	30	0	1	0	1
int_sort	98	0	1	0	1
kmeans	106	0	2	0	2
matrix_mult	58	0	0	0	0
page_rank	101	3	0	3	0
PQ	148	0	4	0	4
freqmine	2205	196	131	541	119
swaptions	1163	3	35	230	199
label_image	1,154,882	4,047	13,101	1	27

TABLE I
RESULTS OF RUNNING THE BLIME LINTER ON THE OISA BENCHMARKS, PARSEC BENCHMARKS AND TENSORFLOW LITE. WE COMPARE THE REPORTED VIOLATIONS WITH RESULTS OBTAINED DYNAMICALLY ON BLIME-SPIKE. LOC INDICATES LINES OF CODE.

We further evaluate the BliMe linter empirically against a debug-enabled implementation of BliMe on the RISC-V Spike emulator, which we call BliMe-Spike. This debug-enabled implementation does not fault when it identifies BliMe violations, but instead produces warnings that identify the offending instructions. This allowed us to run each executable only once to obtain several violations instead of having to "fix" and rerun the executable for each one. Due to its dynamic nature, BliMe-Spike has the following properties.

- It can only identify violations that are on the execution path. This results in a list of violations that is a subset of that produced by the BliMe linter.
- It intentionally does not propagate taint through implicit flows. It will correctly report a violation on a branching condition that uses a blinded value, but will not propagate this violation to assignments performed within the violating then/else branches that would be unreachable with BliMe enforcements enabled. This is because BliMe-Spike does not understand high-level program semantics and therefore cannot reason about where the branches end. The other design choice would have been to propagate taint to *all* assignments after the branching violation, but this would produce many false positives and would reduce the overall usefulness of the output.
- The same violation can result in many duplicate outputs. For example, a branching violation within a for loop that performs 100 iterations will produce 100 warnings instead of one. BliMe-Spike's output therefore requires post-processing to remove duplicates.

We selected several applications for the empirical evaluation: the OISA benchmarks [3], two PARSEC benchmarks, and an image classification example from TensorFlow Lite [27].

A. OISA

Yu et al. use a suite of benchmarks to evaluate OISA by comparing the performance before and after the benchmarks are made constant-time. We therefore use the non-constant-time versions as a starting point for our evaluation. We compile the OISA benchmarks, using the same sensitive inputs as Yu et al and annotating them with the `blinded` attribute to mark them as taint sources. We then compare our analysis results with the violations reported by BliMe-Spike. The results are shown in Table I. We present two case studies from the OISA benchmarks that show the effectiveness of the BliMe linter. We discuss false positives and negatives in the Appendix.

1) `find_max`: Our first case study uses the `find_max` benchmark, which is a simple program that scans an array of secret values and finds the maximum value in that array. We show the source code for `find_max` in Listing 2.

```
1  void FindMax(
2      __attribute__((blinded)) int arr[],
3      int* max_idx,
4      int* max_val){
5    *max_val = -1;
6    for (int i = 0; i < N; i++){
7      if (arr[i] > *max_val){
8        *max_idx = i;
9        *max_val = arr[i];
10 } } }
```

Listing 2. Source code of OISA benchmark `find_max` for finding the maximum value.

As can be seen in the listing, we add the `blinded` attribute to the `arr` pointer parameter. This marks the data *within* the array as blinded. Our analysis correctly identifies a BliMe violation on line 7. The violation is due to the use of a blinded value (`arr[i]`) in the condition of a branch. The BliMe linter has an optional feature to output LLVM IR that contains taint attached to instructions as debugging metadata. The IR output from the `find_max` analysis is shown in Listing 3.

```
%8 = getelementptr inbounds i32,
        i32* %0, i64 %7
%9 = load i32, i32* %8, align 4, !t
%10 = icmp sgt i32 %9, %6, !t
br i1 %10, label %11, label %14, !t
```

Listing 3. LLVM IR of BliMe violation in the `find_max` benchmark.

Instructions marked with a `!t` represent blinded data. `%8` is the pointer to `a[i]`, i.e., `&a[i]`. Note that it is correctly not marked as blinded. The following load fetches the blinded data from the array and stores it in `%9`, which is correctly marked as blinded. Taint is propagated by the analysis to `%10`, which is then used as a condition in a branching instruction on the following line, resulting in a BliMe violation.

2) `page_rank`: For the `page_rank` benchmark, we examine the violation shown in Listing 4. `graph` is a pointer to a blinded struct representing the secret graph of web pages. This causes two violations on line 3. The first is when `numOutEdges` is loaded, and the second is when it stored (after incrementing). The exact taint propagation is shown more clearly in the LLVM IR in Listing 5. `%10` loads the value of `e->src`, which is blinded. A pointer (`%12`) is derived from this value and used to load `numOutEdges` (`%13`). This is marked as a violation due to a load using a blinded address. The same pointer is then used again to store the incremented value on the line 7, resulting in another violation, this time due to a store using a blinded address.

```
1  for(int i = 0; i < numEdges; i++){
2    Edge* e = &graph->edges[i];
3    graph->vertices[e->src].numOutEdges++;
4  }
```

Listing 4. Source code snippet of load/store violation in OISA benchmark `page_rank`. `graph` is a pointer to a blinded struct. There are two violations reported on line 3: a store and a load.

```
1  %10 = load i32, i32* %9, align 4, !t
2  %11 = sext i32 %10 to i64, !t
3  %12 = getelementptr inbounds %struct.Vertex,
4          %struct.Vertex* %5, i64 %11, i32 2, !t
5  %13 = load i32, i32* %12, align 8, !t
6  %14 = add nsw i32 %13, 1, !t
7  store i32 %14, i32* %12, align 8, !t
```

Listing 5. LLVM IR of BliMe violations in the `page_rank` benchmark.

B. PARSEC

We select two benchmarks (`swaptions` and `freqmine`) from the PARSEC suite and mark their inputs as taint sources. They are selected based on easiness of RISC-V cross-compilation, which is not readily supported in PARSEC. For `freqmine`, we taint data as it is read from the input file. `swaptions` generates random inputs instead of reading them from a file; we taint the inputs immediately after they are generated. The results for the BliMe linter and BliMe-Spike are included in Table I. We discuss false positives and negatives in the Appendix.

C. TensorFlow Lite

For TensorFlow Lite, we used the image classification example, `label_image`, marking the input image as sensitive. With BliMe-Spike, we obtained 27 branching violations and 1 memory access violation. 17 of the 27 branching violations were due to uses of the `std::max` and `std::min` function in libc. 17 of all violations were in the `gemmlowp` matrix multiplication library used internally by TensorFlow.

With the BliMe linter, we faced challenges when running our Analysis iterations on the image classification example. Attempting to run the Analysis until no further iterations were required (i.e., function cloning produced no changes) resulted in a run-time greater than 70 hours and memory usage greater than 110GB. We discuss this in more detail in Section V-A. Our analysis was eventually killed due to lack of resources and we therefore decided to disable function cloning and rerun the analysis. Our memory usage was still high ($\tilde{8}$0GB) but runtime was significantly lower ($\tilde{2}$4 hours). As a result, however, the analysis was imprecise and there was a significant amount of taint over-approximation. This is evident in the results shown in Table I. We discuss the challenges of using our approach for large programs in Section V.

V. Discussion & Future Work

A. SVF execution

One challenge we faced when applying our solution to a large executable such as TensorFlow was SVF's very long runtime and high memory overhead, which align with reports on SVF's GitHub page. The cause of this is two-fold. The first is that pointer analysis takes a long time to cover the entire executable. The second is the SVFG generation process itself.

One of the reasons the analysis takes a long time is that SVF builds an SVFG of the *entire* program without considering where the taint sources are located. One possible solution is to limit the analysis to the parts of the program that might contain taint and build the SVFG in tandem with taint propagation.

B. Taint over-approximation

As with any static analysis technique, our analysis must make conservative approximation to be computationally feasible (Section II-D2). We choose to use a conservative approach that over-approximates rather than under-approximates taint. This aligns with our goal of guaranteeing that the program will run if no violations are reported.

While this is sufficient to guarantee BliMe compatibility for a subset of programs, it may result in false negatives. We implemented function cloning to achieve partial context sensitivity (Section III-C2) and eliminate some false negatives but leave further exploration of more accurate analyses as future work. Additional source code annotations could also be added to allow programmers to explicitly denote data as non-blinded to limit the complexity of the analysis.

C. Applicability

Although we focus in this paper on BliMe, the BliMe linter is applicable to any system that requires data-oblivious execution. One prominent example is OISA [3], which enforces a similar policy to BliMe. Furthermore, BliMe linter's analysis is useful even for software that runs on ordinary processors without BliMe or other data-oblivious execution extensions because it can hep developers reason about how their software handles sensitive data and identify potential leaks.

D. Transformations

The BliMe linter provides developers with valuable information about potential side channel vulnerabilities in their programs. Developers can then use this information to manually transform the programs to remove these vulnerabilities. However, with a large volume of vulnerabilities, manual transformations can be cumbersome and time-consuming. Therefore, a natural next step for future work is to extend the compiler to perform these transformations automatically. While this might not be feasible in all cases (e.g., if static analysis cannot identify proper bounds for a blinded pointer), it can significantly reduce the amount of manual labor required, especially if a program contains many easily-transformed violations such as conditional select statements.

One way to integrate transformations into the BliMe linter is by using Constantine [28]. Constantine is a set of LLVM compiler extensions that perform CFL and DFL transformations to produce constant-time code. CFL is done using a form of predicated execution, which computes a predicate for each branch, executes both branches, and masks the results using the predicate to obtain the correct result and discard the result from the incorrect branch. DFL linearizes array accesses that use a tainted index by iterating over the entire array using a stride equal to the cache line size, and selecting only the value at the correct index using the predicate.

BliMe and OISA face two challenges in using Constantine:

1) Array access expansion: Constantine's array access expansion results in binaries that are only constant-time on CPUs with a cache line size matching the chosen stride. Furthermore, the offset of each access from the beginning of the cache line is derived from the sensitive index. On BliMe and OISA, this will cause all the resulting memory access addresses to also be tainted and thus cause the instructions to fault. We can solve this by modifying the array access expansion transform to use a stride of one, which results in no sensitive offset being added to each address and the addresses remain untainted.

2) Select transform: Constantine uses x86-specific optimizations such as the `cmov` instruction. RISC-V does not have such an instruction, which causes IR `select` instructions to be lowered to conditional branching instructions in assembly. Thus integration with Constantine must include special handling for IR select instructions.

VI. Related Work

As mentioned in Section V-D, Constantine [28] is a state-of-the-art constant-time compiler. While Constantine's transformations are complementary to the BliMe linter, Constantine also uses taint tracking to identify potential violations. However, it uses *dynamic* analysis to propagate taint, which involves running the target program several times with different sets of realistic inputs and tracking taint throughout each execution. The downside to this approach is that, contrary to the BliMe linter, Constantine's analysis is not sound as it only considers the observed execution paths and misses information flows that only occur on paths that were not exercised.

VII. Conclusion

Manually identifying side channels in programs is difficult. The BliMe linter facilitates this process by performing static taint-tracking analysis and reporting potential violations. This allows developers to focus only on problematic areas in their software. Through an iterative process, we show that the BliMe linter can help developers eliminate all non-compliant code. However, as with most static analysis techniques, there is room to improve precision. In addition, automatic code transformations are a promising line of future work.

REFERENCES

[1] H. ElAtali, L. J. Gunn, H. Liljestrand, and N. Asokan, "BliMe: Verifiably secure outsourced computation with hardware-enforced taint tracking," in *Network and Distributed Systems Symposium (NDSS)*, San Diego, CA, USA, 2024, ISBN: 1-891562-93-2. DOI: 10.14722/ndss.2024.24105.

[2] Y. Sui and J. Xue, "SVF: Interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*, New York, NY, USA, 2016, pp. 265–266. DOI: 10.1145/2892208.2892235. [Online]. Available: https://doi.org/10.1145/2892208.2892235.

[3] J. Yu, L. Hsiung, M. El'Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, 2019, ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23061.

[4] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Report 2016/086, 2016.

[5] H. B. Lee, T. M. Jois, C. W. Fletcher, and C. A. Gunter, "DOVE: A data-oblivious virtual environment," in *Proceedings of the Network and Distributed System Security Symposium*, Virtual, 2021, ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.23056.

[6] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '09, Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178, ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. [Online]. Available: https://doi.org/10.1145/1536414.1536440.

[7] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *Proceedings of the USENIX Security Symposium*, 2021, pp. 645–662, ISBN: 978-1-939133-24-3.

[8] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proceedings of the International Cryptology Conference on Advances in Cryptology*, Berlin, Heidelberg, 1999, pp. 388–397, ISBN: 3-540-66347-9.

[9] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Berlin, Heidelberg, 2001, pp. 251–261, ISBN: 3-540-42521-7.

[10] J. Longo, E. De Mulder, D. Page, and M. Tunstall, "SoC it to EM: ElectroMagnetic side-channel attacks on a complex system-on-chip," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Berlin, Heidelberg, 2015, pp. 620–640, ISBN: 978-3-662-48323-7. DOI: 10.1007/978-3-662-48324-4_31.

[11] M. Lipp, A. Kogler, D. Oswald, *et al.*, "PLATYPUS: Software-based power side-channel attacks on x86," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2021, pp. 355–371. DOI: 10.1109/SP40001.2021.00063.

[12] T. Kim and Y. Shin, "ThermalBleed: A practical thermal side-channel attack," *IEEE Access*, vol. 10, pp. 25 718–25 731, 2022. DOI: 10.1109/ACCESS.2022.3156596.

[13] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020, ISSN: 0001-0782. DOI: 10.1145/3357033.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing intel secrets from SGX enclaves via speculative execution," in *Proceedings of the IEEE European Symposium on Security and Privacy*, 2019, pp. 142–157. DOI: 10.1109/EuroSP.2019.00020.

[15] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from qualcomm's TrustZone," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2019, pp. 181–194, ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354197.

[16] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "TruSense: Information leakage from TrustZone," in *Proceedings of the IEEE Conference on Computer Communications*, 2018, pp. 1097–1105. DOI: 10.1109/INFOCOM.2018.8486293.

[17] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[18] M. Schwarz, M. Lipp, D. Moghimi, *et al.*, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2019, pp. 753–768, ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354252.

[19] J. Van Bulck, M. Minkin, O. Weisse, *et al.*, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 991–1008, ISBN: 978-1-939133-04-5.

[20] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida, "Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks," in *Proceedings of the USENIX Security Symposium*, 2021, pp. 1451–1468, ISBN: 978-1-939133-24-3.

[21] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proceedings of the European Workshop on Systems Security*, New York, NY, USA, 2017, pp. 1–6, ISBN: 978-1-4503-4935-2. DOI: 10.1145/3065913.3065915.

[22] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proceedings of the USENIX Conference on Offensive Technologies*, USA, 2017.

[23] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proceedings of the USENIX Security Symposium*, USA, 2017, pp. 557–574, ISBN: 978-1-931971-40-9.

[24] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Proceedings of the 8th International Conference on Information Security and Cryptology*, (Seoul, Korea), Berlin, Heidelberg, 2005, pp. 156–168. DOI: 10.1007/11734727_14.

[25] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953, ISSN: 0002-9947. DOI: 10.2307/1990888.

[26] "Whole program LLVM." (), [Online]. Available: https://github.com/travitch/whole-program-llvm (visited on 05/13/2024).

[27] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[28] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic side-channel resistance using efficient control and data flow linearization," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2021, pp. 715–733, ISBN: 978-1-4503-8454-4. DOI: 10.1145/3460120.3484583.

Appendix

False positives/negatives

For the OISA benchmarks, we manually audited the source code and all reported violations to identify any false positives or negatives. For all benchmarks other than `kmeans`, no false positives or negatives occurred. The BliMe linter was able to capture all violations reported by BliMe-Spike as well as other potential violations that would have been reported by BliMe-Spike had their execution paths been exercised.

With `kmeans`), on the other hand, the BliMe linter produced a false negative. Upon further investigation, we discovered that this was caused by a call to the `fabs` C library function. As this function was in an external pre-compiled library, there was no available LLVM bitcode for it and its definition was not included in our whole-program bitcode module. As a result, in the case of `kmeans`, the BliMe linter did not propagate taint through a call to this function, resulting in taint being incorrectly lost. This was also found to be the case in both PARSEC benchmarks and is the cause for the large difference in violations between the BliMe linter and BliMe-Spike in Table I for these benchmarks.

One solution to this is to ensure that bitcode for all parts of the program is included in the analysis. Another solution is to annotate common C/C++ standard library function with taint propagation rules to match their functionality and integrate this into the BliMe linter's taint tracking. We leave this for future work.