Compiler-Based Approach to Enhance BliMe Hardware Usability

by

Xiaohe Duan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

Outsourced computing has emerged as an efficient platform for data processing, but it has raised security concerns due to potential exposure of sensitive data through runtime and side-channel attacks. To address these concerns, the BliMe hardware extensions offer a hardware-enforced taint tracking policy to prevent secret-dependent data exposure. However, such strict policies can hinder software usability on BliMe hardware.

While existing solutions can transform software to make it constant-time and more compatible with BliMe policies, they are not fully compatible with BliMe hardware. To strengthen the usability of BliMe hardware, we propose a compiler-based tool to detect and transform policy violations, ensuring constant-time compliance with BliMe. Our tool employs static analysis for taint tracking and employs transformation techniques including array access expansion, control-flow linearization and branchless select. We have implemented the tool on LLVM-11 to automatically convert existing source code.

We then conducted experiments on WolfSSL and OISA to examine the accuracy of the analysis and the effect of the transformations. Our evaluation indicates that our tool can successfully transform multiple code patterns. However, we acknowledge that certain code patterns are challenging to transform. Therefore, we also discuss manual approaches and explore potential future work to expand the coverage of our automatic transformations.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Outsourced computing provides efficient and stable platforms and services for individuals and organizations to process data. A common setting is that clients of outsourced computing send sensitive data to service providers and receive the processed results. However, the underlying software of outsourced computing infrastructure can be malicious or vulnerable to external adversaries. Hence client data can be exposed by run-time attacks, or even more covertly by side-channel attacks. Addressing clients' concern on sensitive data leakage is a major challenge in outsourced computing.

As a solution to this challenge, the BliMe [18] hardware extensions enforce a taint tracking policy to track secret-dependent data and triggers a fault if there is an attempt to expose secret-dependent data to external observers, via run-time attacks or side-channels such as cache and timing side-channels. Unlike Fully Homomorphic Encryption (FHE)[20], BliMe performs computations directly on decrypted data after the data import, bypassing the high performance overhead caused by computing on encrypted data. Also, BliMe ensures that decrypted data on server-side is always tagged as tainted and tracked by the hardware, preventing the potential leakage from vulnerable or malicious server-side software.

While BliMe hardware provides efficient and secured outsourced computation, its strict policies pose obstacles for the usability. Most software is not designed to be constant-time, i.e., avoiding secret-dependent branches and memory accesses. Directly deploying most software on BliMe hardware incurs frequent faults and weakens the availability of the software. Some of these faults are not caused by malicious attackers. Also, a portion of them can be fixed by modifying the code positions that correspond to the faults. Fixing such code positions can strongly enhance the usability of the BliMe hardware. As manually finding potential leakages and writing constant-time code is difficult and error-prone, a compiler-based tool that automatically generates constant-time binaries from the general source code is required to improve the usability of BliMe hardware. The tool needs to meet the following requirements:

- The analysis should be able to cover all the execution paths and find all the potential violations.
- The transformation should transform both the secret-dependent branches and memory accesses into a constant-time version that is compatible with BliMe.

Prior work has developed tools to automatically transform programs into a functionally-equivalent constant-time versions. Some of the previous solutions [43][14] cover only the control-flow related side-channels, and are unable to fix the violations from secret-dependent memory accesses. SC-Eliminator [53] use static analysis to track the secret-derived data and fix the secret-dependent data-flow using data pre-loading techniques. Each lookup table is preloaded before actual accesses so each memory access is a cache hit. However, BliMe hardware still considers this access a violation. Constantine [9] uses data-flow and control-flow linearization techniques to remove side-channels. However, it tracks tainted data using dynamic profiling tools. As a result, it can fail to cover all the execution paths and miss potential faults when running on the hardware. In addition, all the existing works require specific hardware such as TSX [56][21]. Therefore, existing works cannot fulfil all the requirements for compiler support for BliMe hardware. They either cannot detect and transform some violations or generate executables incompatible with BliMe hardware or the enforced policy.

As the limitations of prior work show, compiler support to strengthen the usability of BliMe hardware should use analysis that covers all the execution paths and generates the code compliant with the BliMe hardware policies. Accordingly, the taint-tracking analysis should be an analysis that detects all code positions that potentially cause policy violations when running on BliMe hardware. Also, the transformations should be able to transform both the secret-dependent branches and memory accesses such that they remain functionally-same but compatible with BliMe policy.

In this work, we aim to develop a compiler-based tool that fulfils all these requirements: it utilizes sound static analysis to figure out all the potential violations, and uses control-flow and data-flow linearization techniques to transform the secret-dependent branchings and memory accesses to be compliant with the BliMe policy. We also classify programs that we cannot transform and suggest manual changes and future compiler transformations as discussion.

We implement the compiler-based tool on LLVM-11. We use SVF [44][2] value-flow analysis for taint tracking, as it generates sound points-to analysis and inter-procedural value-flow graph. We then implement our LLVM passes for array access linearizations. We Also forward-port some Constantine passes for control-flow linearization. We then evaluate the transformation and analysis by running the transformed results on BliMe hardware. The tool requires user added annotations to indicate the sensitive data that represent the sensitive client data while running on the BliMe hardware.

Our main contributions are as follows:

- The design and implementation of a compiler-based tool that detects potential violations to BliMe policy with static analysis and transforms the code positions with potential violations into a constant-time/policy-compliant functional-equivalent version (Chapter 4, Chapter 5). The tool expands array accesses and some branches, but cannot transform all the code positions that potentially cause violations on hardware.

- For the code that our tool cannot transform, we propose the classification of the violations we cannot transform automatically and suggest how to transform them manually (Chapter 6).

Chapter 2

Backgrounds

2.1 Side Channels

Computer system security has long been threatened by determined and smart adversaries whose aim is to hijack programs and perform unauthorized behaviors, such as modifying or exposing sensitive data or disrupting system services. To achieve these goals, adversaries identify program vulnerabilities such as buffer overflows or use-after-free and exploit them to alter the normal program control-flow or data-flow. After breaking the normal program behavior, adversaries can manipulate the program to perform malicious tasks, such as exposing secret data. For instance, by finding a buffer overflow vulnerability, an adversary can manipulate control flow to gain shell access using techniques, such as return-oriented programming (ROP)[40][12]. Even with much more efforts on strengthening the defenses against such attacks[45], adversaries aiming to leak secrets from a system are increasingly turning to more covert methods, such as side-channel attacks.

Side-channel attack is a type of the attack in which adversaries observe the secret-related observable states of a system i.e., side channels, rather than directly accessing the secret, to leak the secret. For example, in a side-channel attack, the adversary can measure the power consumption of a RSA decryption [26] to recover the secret key as the power consumption for a modular exponentiation operation varies depending on the value of each bit in the secret key. If the secret key bit is `0`, then modular exponentiation operation is only a shift; but if the bit is `1`, then an extra multiplication is required. So the power consumption on encryption on a `1` bit of secret key is more than a `0` bit. In contrast, traditional run-time attacks tend to directly expose the secret key to peripherals. While there are various side-channel attacks in modern computer systems, such as cache

[22], execution time [27], energy consumption [23] and other microarchitectural [33] side-channels, we focus on cache and timing side-channels in this thesis. Specifically, we focus on the timing and cache side-channels stemming from secret-dependent branchings and memory accesses.

Secret-dependent Memory Access. Latency for memory access is a performance bottleneck in modern CPUs. To mitigate this, a practical approach is to use a cache, a smaller but faster buffer to store recently accessed data. Accessing the data in the cache (cache hit) is much faster than fetching the data from memory (cache miss).

When secret-dependent memory accesses exist, attackers can exploit the difference in memory access time to recover the secret data. For example, the implementation of AES algorithm uses lookup tables for encryption; the attacker can measure the memory access time to infer the pattern of secret key. Below is the AES encryption implementation from WolfSSL [5]. In the code snippet, there are four lookup table (`Te`) accesses for each `si`. Each access is dependent on a value (`ti`) dependent on the secret key. If `ti` remains the same for `si` in each round of the iteration, the memory access time is much lower because there are more cache hits. However, if the `ti` varies, there would be more cache misses, thus higher memory access time.

```
for (;;) {
    ...
    if (--r == 0) {
        break;
    }
    // t0 - t3 are secret-key-dependent
    // Memory access Te[i][GETBYTE(ti, j)] is secret dependent.
    s0 = Te[0][GETBYTE(t0, 3)] ^ Te[1][GETBYTE(t1, 2)] ^ Te[2][GETBYTE(t2
  , 1)] ^ Te[3][GETBYTE(t3, 0)] ^ rk[0];
    s1 = ...
    ...
}
```

Listing 2.1: Secret-dependent memory access in WolfSSL

Adversaries can passively measure the memory access time [8] to infer the secret bits. They can also launch active flush+reload[54], prime+probe[36], in which they manipulate the cache by evicting the stored data and then monitor the memory access behavior. Data preload is a defense to passive attackers. For the example above, the programmer can carefully load all the data that can be accessed before actually accessing them for encryption and decryption. wolfSSL [6] also preloads the `Te` table to make the memory access cache side-channel resistant. However, this defense fails to protect the program against active attackers as active attackers can evict the preloaded data.

Secret-dependent Branching. Branching is the process of selecting one execution path among several paths that a program can follow based on a condition. In programming languages, conditional branching is the basis of if-else statements, loops and switches. In hardware level, branching is implemented as an instruction that takes in a condition and a jump address. If the condition is true, then program counter (PC) will be updated and the control-flow will be switched to the jump address. A branch is secret-dependent if its result depends on a secret condition.

Secret-dependent branches can be exploited if they affect the computation time, branch predictor [30] or caches. The attacker can use the timing or cache status differences as a side channel to infer which sequence of branches have been taken, thus inferring the value of the sensitive data. For example, the listing below is a code snippet of modular exponentiation, a common component of cryptography algorithms. In this algorithm, `exp` is often a sensitive value. However, in this implementation, an attacker can infer the value of secret bits of `exp` because when the bit is 1, the program takes more execution time as an extra multiplication is needed.

```c
uint64_t expmod(uint64_t base, uint64_t exp, uint64_t modulus) {
    uint64_t result = 1;
    base %= modulus;
    while (exp > 0) {
        // if the last bit is 1, then multiply and mod needed
        if (exp & 1) {
            result = (result * base) % modulus;
        }
        base = (base * base) % modulus;
        exp >>= 1;
    }
    return result;
}
```

Listing 2.2: Square and multiplication implementation of modular exponentiation.

Defenses. To prevent such side-channel attacks, one solution is for programmers to write constant-time programs, of which control-flow and data-flow cannot be secret-dependent. However, writing constant-time code is challenging and error-prone. Some existing works employs compiler-based method to automatically transform the program to be constant-time [9][56][43].

2.2 BliMe

As the demand for computing resources increases, individuals and companies resort to cloud service providers and outsource data processing. However, outsourced computation requires the client to send the data to the service provider, hence the confidentiality of client data becomes a realistic concern. Two major solutions to protect the client data confidentiality are FHE [20] and Trusted Execution Environment (TEE). FHE performs computations on client-encrypted data and send back the result, client then decrypt the received data to get the result. FHE secures the confidentiality but with a high performance overhead. On the other hand, TEE guarantees data integrity and confidentiality policies on a specific area of the CPU using hardware. This solution more efficient but still vulnerable under run-time attacks against TEEs [11][39].

Blinded Memory (BliMe) [18] is an recently proposed architecture aiming to efficiently prevent the leakage of client data in outsourced computation settings. BliMe enforces the policy with hardware to prevent secret-dependent data from being observable in plaintext. It protects the confidentiality even under run-time attacks and side channel attacks while being not expensive as FHE, as it computes directly on data rather than encrypted data.

The workflow of BliMe hardware can be summarized as data import, safe computations and data export. In the context of BliMe hardware, blinded means not observable. BliMe hardware imports the encrypted data from the client, decrypts the data and blinds the memory that contains the plaintext data. After data import, arbitrary *safe* operations are allowed on blinded data. *safe* operations are defined as operations that do not expose data to observable status, such as arithmetic operations. To guarantee the confidentiality of data during computation, BliMe hardware uses taint-tracking and memory tagging to track the blinded data. The rule of the BliMe taint-tracking policy is that, the outputs of operations using blinded inputs are also tagged as blinded. Before blinded data flows to an observable output, a hardware fault will be triggered to prevent the potential leakage. Once the hardware completes all the operations, data can be exported with an encryption. To securely support such functionality, a system using BliMe hardware consists of the following components:

1. **Hardware Security Module (HSM)**: a fixed-function hardware module that provides remote attestation and key agreement utilities. In remote attestation, HSM verifies that the server is using BliMe hardware. Then HSM makes key agreement with the client and stores the key into the encryption engine via a secure channel.

2. **BliMe encryption engine**: an encryption engine that provides atomic data import and export operations for client data. Internally, each atomic import/export consists

of two operations: 1) decryption/encryption using the key from HSM; 2) data blinding/unblinding. Atomic operations ensures that plaintext data are guaranteed to be blinded, while the unblinded data are encrypted thus confidential even if exposed to malicious adversaries.

3. Application Software: BliMe provides environments for both TEE and Rich Execution Environment (REE) application software. With all the memory dependent on client data tagged as blinded, the operations that potentially expose plaintext client data always trigger faults. As a result, the computations are guaranteed to be safe from data leakage.

The definition of observable state determines the categories of data leakage attacks that the BliMe hardware can defend against. In BliMe, observable states are states that can be inferred or observed by external observers such as the value of Program Counter (PC) and cacheline status, peripherals, the addresses of memory operations sent to main memory, etc. Since PC, cache status and other components cannot be marked as blinded during the taint tracking, they are also defined as observable state. Any attempts to change the PC and cache status based on blinded data will cause fault as a result. Therefore, BliMe hardware is also resilient against timing and cache side-channel leakages.

2.3 LLVM

2.3.1 LLVM IR

LLVM[47][29] project is a collection of modular compiler components, toolchains and libraries. LLVM divides compilation into three phases: front-end, Intermediate Representation (IR) optimization and back-end. A compiler under LLVM infrastructure typically follows the highly-modular three-phase design. The front-end takes the input source code and outputs the LLVM IR. The optimizer analyzes and transforms the IR code with passes to improve the performance or strengthen the security. The back-end code generator generates machine code for target architectures.

LLVM IR is the key to the three-phase modular design. A substantial number of LLVM core libraries are designed to generate, process and translate the LLVM IR code. It not only acts as the common language connecting the front-end and the back-end but also serves as the interface for most optimizations. LLVM IR adopts the Static Single Assignment (SSA) form in its representation. SSA form ensures each variable or value in the program is assigned only once throughout its lifetime. In LLVM IR's SSA form, every register is defined exactly once. The adoption of SSA form makes the def-use relationship clear and simplifies the analysis such as data flow analysism thus enhancing the efficiency of code transformation and optimization. In addition, LLVM IR preserves some control-flow structures and data structures information from source languages for analysis and optimization. LLVM IR can be of three functionally equivalent representations: in-memory compiler IR, human readable assembly language representation and bitcode representation (more compact but not human-readable).

We explain three IR concepts to help understanding this thesis:

- **GetElementPointer (GEP).** GEP is an instruction that calculates the memory access address in IR. It calculates a memory access address based on the base pointer and indices.
- **Phi.** Since IR is of SSA form, it is impossible to assign one register twice. Phi is the instruction that merges the control flow and allows different assignments to the same register when coming from different control-flows.
- **Metadata.** Metadata refers to additional data attached to the IR code that indicates the optimization hints, attribute, etc. For this thesis, metadata is a hint for transformations.

2.3.2 LLVM Pass Manager

Same as the modular design of the whole compiler, the optimization phase consists of individual passes that can be flexibly configured and combined to serve different optimization purposes. LLVM introduces pass manager to orchestrate the passes. Pass managers schedule the passes and manage the analysis results. LLVM opt command-line tool utilizes pass manager to run customized optimizations on the IR files.

2.4 Constantine

Constantine[9] is a compiler-based tool that automatically transforms programs into constant-time versions. Constantine hardens the programs using comprehensive control-flow and data-flow linearizations.

To identify the secret values, i.e., the sensitive values that should be protected, Constantine hooks the input functions and marks the input values as secret. It then employs DataFlowSanitizer (DFSan), a generic dynamic data flow analysis tool for LLVM, to track the dependencies of secret data by profiling the program. Through this process, Constantine takes in an executable and a profiling suite consisting of well-crafted input data to cover the execution paths, outputting a list of secret-dependent data locations.

With the secret-dependent data locations identified, Constantine linearizes the branches and memory accesses dependent on these data. Prior to linearization, Constantine applies normalization passes to normalize the IR of the program. These passes transform indirect calls into branches, lower switches into if-else statements, and unify function entry- and exit-points, resulting in a normalized code with single-entry and single-exit regions for linearization. Normalization simplifies later control-flow linearizations.

For branches, Constantine removes conditional branches and forces the program to execute both branches. To ensure that the original program functionality is maintained, Constantine introduces a `taken` predicate to indicate whether the branch is taken. All data accesses within the branch are altered based on the `taken` predicate. When `taken` is `false`, memory accesses are executed as dummy operations, as their results will not be selected and will have no effect on the program.

For variable-size loops, Constantine produces estimated loop-bounds with profiling. The loop only exits when it has reached the condition to exit and reached the loop-bounds. The constant-time guarantee is built on the coverage of the profiling suite and the profiling result.

For memory accesses, Constantine accesses possible objects of a memory access and stride the array cache lines. Constantine also maintains a list of live objects to reduce the performance overhead.

2.5 SVF

SVF[2][44] is a scalable and well-maintained inter-procedural static analysis tool. SVF provides comprehensive APIs to generate useful graphs for program analysis such as value-flow graphs, call graphs and control-flow graphs and several pointer analysis.

SVF analysis takes LLVM bitcode as input. Before the graph generation, SVF builds Program Assignment Graph (PAG) by converting the LLVM IR into PAG nodes. PAG nodes represent the pointer or the object, while PAG edges represent the load, store, copy, addr, GEP, call and return instructions in LLVM. SVF solves the constraint of PAG edges to build the points-to sets. The user can decide the analysis to use by implementing or using different solvers.

SVF then constructs the memory SSA based on the points-to information. SVF annotates each load, store, callsite and the entry and exit points of functions with the def-use of the address-taken variables. For example, a load p = *q is annotated with the use of all the points-to objects of pointer q. For inter-procedural def-use for address-taken variables, SVF annotates the callsites, entry points and exit points with def-use of non-local variables and parameters.

Finally, SVF constructs the value-flow graph called SVFG by connecting the def-use chains of top-level and address-taken variables. The edges thus represent the value-flow. Since LLVM IR is in SSA form, the def-use chains of top-level variables (IR registers) are already available before the analysis. These top-level variables are connected with direct edges by def-use chains from LLVM. Meanwhile, address-taken variables are connected with indirect edges based on the def-use chain of the constructed memory SSA.

Except for the nodes representing the instructions in LLVM IR, SVFG has memory region nodes such as ActualIn (entry node of callsite), ActualOut (exit node of callsite), FormalIn (entry node of defined function) and FormalOut (exit node of defined function). They not only represent the function parameters and return values of the top-level variables, but also the pointer parameters to the address-taken variables and other global variables involved into the def-use chain in the function. For example, in the following code snippet, SVF constructs the value flow from `global_arr` to the ActualIn node of `writeGlobal`, and then to the ActualOut node of `writeGlobal`.

```c
int global_arr[1024];

void writeGlobal(char* from_buffer) {
  for (int i = 0; i < 1024; i++)
    global_arr[i] = from_buffer[i];
}


int main(int argc, char* argv[]) {

  char buffer[1024];
  /* write the local buffer */

  /* write global buffer with local buffer */
  writeGlobal(buffer);

  /* use global buffer */
  useGlobal();

  return 0;
}
```

Listing 2.3: Code example to explain SVFG `ActualIn` and `ActualOut` nodes.

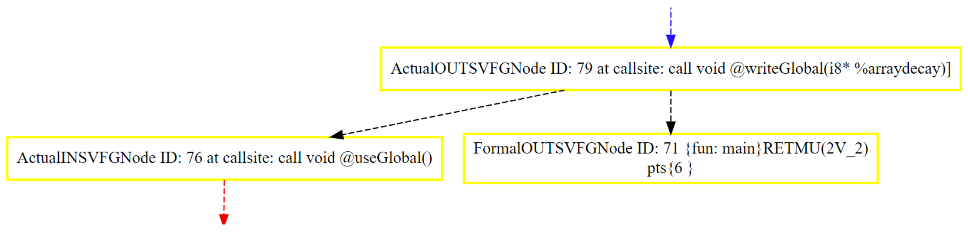

ActualOUTSVFGNode ID: 79 at callsite: call void @writeGlobal(i8* %arraydecay)]
ActualINSVFGNode ID: 76 at callsite: call void @useGlobal()
FormalOUTSVFGNode ID: 71 {fun: main}RETMU(2V_2) pts{6 }

Figure 2.1: Part of the SVFG for Listing 2.3

Chapter 3

Problem Description

3.1 Problem Statement

As discussed in 2.2, based on the designed security policy, BliMe triggers faults on secret-dependent branching or memory access instructions, and end the execution. For a program to run on BliMe, it must be side-channel resistant with respect to blinded input data. However, writing constant-time code manually requires strong background knowledge on side-channel attacks and careful consideration. Even mature cryptography libraries such as OpenSSL [5], in which developers put considerable efforts to detect and eliminate side channels, still suffer from cache side channels [3]. As a result, it is necessary to explore compiler supports to automatically detect the potential violations and transform the code.

In this work, we aim to improve the usability of BliMe hardware with compiler support. The compiler support should enumerate the potential code positions that cause unintended violations on BliMe hardware and automatically transform the IR code to avoid violations when running on BliMe hardware. The program analysis must cover all the execution paths, while compiler transformations should strive to transform both secret-dependent branchings and memory accesses to be constant-time. There have been several compiler-based transformation tools to make the code constant-time. However, they either use an analysis that is unable to cover all the execution paths, or attempt to fix only specific types of side-channels.

Besides, existing solutions are incompatible with BliMe due to a common issue: programs that appear constant-time to software can still potentially violate the BliMe policy and lead to faults. The root of this issue lies in the difference between the hardware and the program analysis tools in their perspectives on the software. Program analysis tools have a more comprehensive understanding of the analyzed software, including cache access patterns throughout the entire program, program semantics, etc. In contrast, the hardware is limited to knowledge about the executed instructions. For example, one approach to make the secret-dependent cache accesses constant-time is to preload the data that can be potentially used in a memory access into the cache before accessing the secret-dependent positions. This modification ensures constant-time memory access since the access will always be a cache hit. However, BliMe is agnostic to the the overall memory access pattern. Instead, it identifies the access as a violation since the access instruction itself is secret-dependent, even if the memory access is constant-time from a software perspective. Hence we need to consider the discrepancies of the observation of the hardware and software analysis tools or programmers to ensure that the transformed code does not violate the BliMe policy, i.e., compatible with the BliMe policy.

3.2 Requirements

Based on the investigation of existing tools that transform code to be constant-time, we have proposed the following requirements to achieve the goal of enhancing BliMe's usability:

3.2.1 Analysis Requirement

R-1 Soundness: code reported as safe for BliMe by the analysis should have no violations during the run-time.

3.2.2 Transformation Requirements

R-2 Correctness: the functionality of the code after the transformation should remain the same as before.

R-3 Minimal source code changes: the transformations should only require changes to the source code for the purpose of identifying sensitive data and nothing else.

R-4 Coverage: the compiler transformation should be able to cover a wide range of program patterns. If unable to transform some code patterns, manual approaches to transform such code should be provided.

3.2.3 Compatibility

R-5 Compatibility with RISC-V: the compiler in this work must generate RISC-V executable that is compatible with BliMe hardware.

R-6 Compatibility with BliMe taint tracking policy: the generated executable should run without unintended violations on BliMe hardware.

Chapter 4

Design

In this chapter, the term *user* refers to the user of our compiler-based tool, i.e., the user that uses our tool to enhance the compatibility of their software on BliMe. As illustrated in Figure 4.1, the workflow of our compiler-based tool begins by feeding in source code from users, where blinded user input data are explicitly annotated as tainted. Our **taint tracking** component then operates on the IR of the code, going through the data flow and identify dependencies on user-marked tainted data. Subsequently, our tool uses **transformations** including array access expansion, Control Flow Linearization (CFL) and branchless `select` to modify the IR code to make programs constant-time. The bitcode can then be compiled into executables for BliMe.

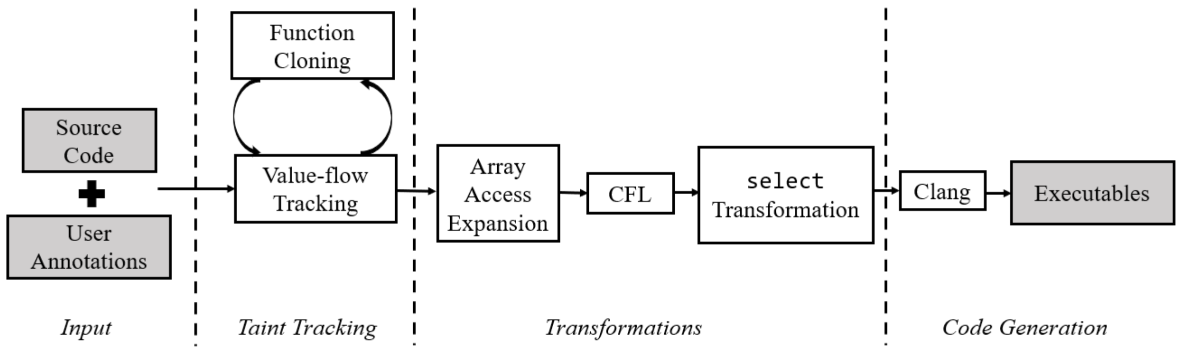


Figure 4.1: The overall workflow of our compiler-based tool.

4.1 Taint Tracking

The goal of our analysis is to identify any data- or control- dependencies on input data that will be blinded at run-time in BliMe hardware. We use compile-time taint tracking technique to achieve this goal.

Taint tracking/analysis is the technique that tracks dependencies of the sensitive or untrusted data, i.e., *taint*, and their interactions with other components of the program, such as variables, control-flow, etc. In the context of taint tracking, taint sources are the starting points of taint propagation, which in our case, means the data expected to be client input data to BliMe. Taint propagation refers to the process that taints spread in the program. Tainted data are data expected to be blinded after taint propagations.

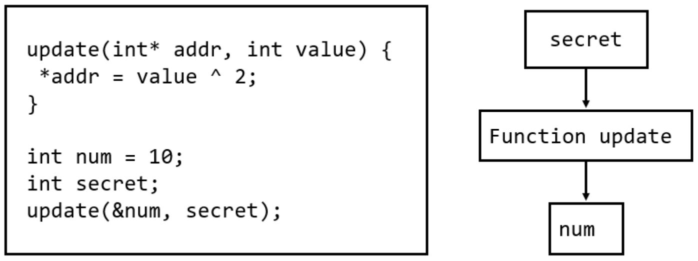
When describing the taint tracking we designed and implemented, we use terminologies from taint analysis, such as taint source, taint propagation, etc. We require users to annotate expected client input data, which should be blinded data on BliMe hardware. Annotated blinded data are taint sources for taint tracking.

4.1.1 Value-Flow Tracking

As explained in Section 2.2, BliMe employs *hardware* taint-tracking at runtime to enforce a policy that prevents blinded data (i.e., secret-dependent data) from influencing the PC, cache status, or directly observable output. Our goal is to track secret-dependent data in programs using our compiler-based tool to prevent violations caused by secret-dependent memory accesses and branches. As a result, our compiler taint-tracking policy should resemble the taint-tracking on the hardware level, i.e., the output of an instruction with tainted input(s) is also tainted.

To achieve this, we utilize an inter-procedural value-flow graph, which is a directed graph representing program entities such as variables, function calls, etc., with edges capturing the def-use relationships between these entities. Figure 4.2 depicts an example program with def-use relationships between variables and its corresponding inter-procedural value-flow graph. In this graph, we can observe that the variable `num` is defined by the value of `secret` through a function call to `update`.

The analysis on def-use relationships allows us to identify data dependent on the taint sources. In our taint analysis, taint propagates through the edges on the value-flow graph via a traversal of the value-flow graph.



```
update(int* addr, int value) {
 *addr = value ^ 2;
}

int num = 10;
int secret;
update(&num, secret);
```

Figure 4.2: An example of a program with def-use relationships between variables and its corresponding value-flow graph.

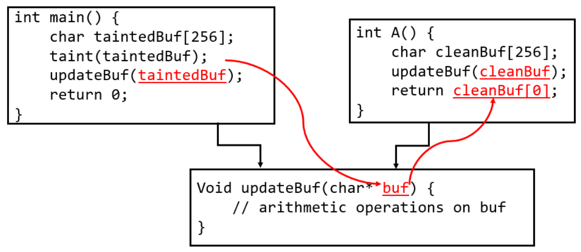
4.1.2 Function Cloning

For programs with multiple calls to the same function, considering all these calls as the same instance leads to a loss of context-sensitivity and reduces the analysis accuracy.

In Figure 4.3a, function `updateBuf` is called twice. The `main` function calls the `taint` function to taint the `taintBuf` array, followed by a call to the `updateBuf` function, which updates the `buf` array using arithmetic operations. In the context-insensitive analysis scenario, the output of `updateBuf`, namely the elements in `buf`, becomes tainted. Consequently, when another function (function `A`) calls `updateBuf` with an untainted array (`cleanBuf`) as `buf`, `cleanBuf` becomes tainted as well, leading to a tainted return value from `A`. This causes the taint propagation to continue excessively, resulting in over-tainting.

Also, transformations to secret-dependent memory accesses and branchings will introduce performance impact, so reducing these transformations on unnecessary positions improves the performance. In Figure 4.4, function `main` calls `accessComp` with different combinations of tainted arguments. In the first call, both arguments are tainted; while in the second call, only the second argument is tainted. Protecting both sensitive memory accesses in both function calls is more costly than required to prevent data leakage, as the first memory access in the second call doesn't require protection.

Having different versions of the same function for different combinations of taint status of arguments can help solve the problems above. We make use of function cloning for this purpose. Function cloning is the technique that generates multiple copies of the same function based on certain properties of the function, such as the context, properties of the input/output, etc. Function cloning technique was used in program analysis to improve the accuracy [9]. As for the problems above, function cloning improves both the analysis accuracy and performance.

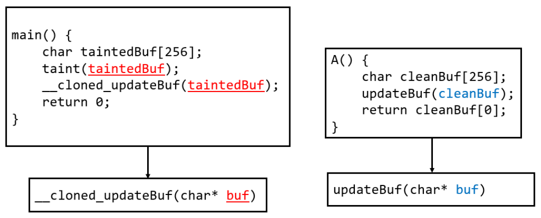


```
int main() {
    char taintedBuf[256];
    taint(taintedBuf);
    updateBuf(taintedBuf);
    return 0;
}
```

```
int A() {
    char cleanBuf[256];
    updateBuf(cleanBuf);
    return cleanBuf[0];
}
```

```
Void updateBuf(char* buf) {
    // arithmetic operations on buf
}
```

(a) Before the function cloning: taint propagates from `updateBuf` function to `A` function.



```
main() {
    char taintedBuf[256];
    taint(taintedBuf);
    __cloned_updateBuf(taintedBuf);
    return 0;
}
```

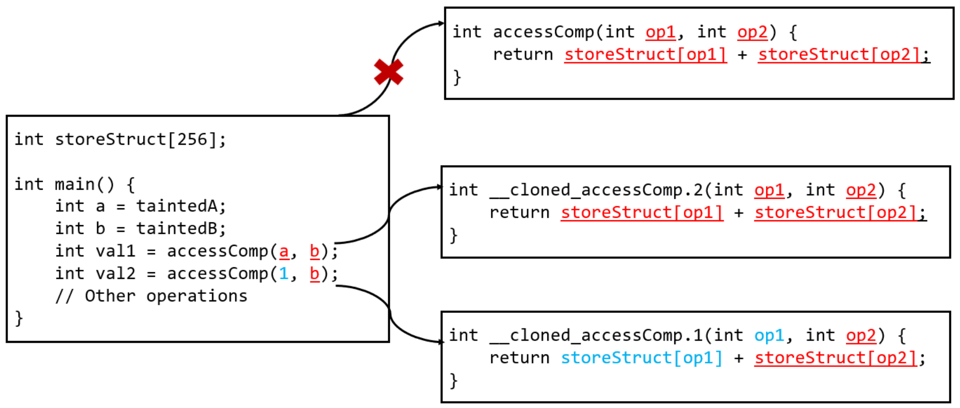
```
__cloned_updateBuf(char* buf)
```

```
A() {
    char cleanBuf[256];
    updateBuf(cleanBuf);
    return cleanBuf[0];
}
```

```
updateBuf(char* buf)
```

(b) After the function cloning:

Figure 4.3: Function cloning can improve the accuracy of the analysis.



```
int storeStruct[256];

int main() {
    int a = taintedA;
    int b = taintedB;
    int val1 = accessComp(a, b);
    int val2 = accessComp(1, b);
    // Other operations
}
```

```
int accessComp(int op1, int op2) {
    return storeStruct[op1] + storeStruct[op2];
}
```

```
int __cloned_accessComp.2(int op1, int op2) {
    return storeStruct[op1] + storeStruct[op2];
}
```

```
int __cloned_accessComp.1(int op1, int op2) {
    return storeStruct[op1] + storeStruct[op2];
}
```

Figure 4.4: Function cloning can improve the performance of transformed program. Multiple versions of cloned functions adopt different transformations. Underlined function arguments are tainted; underlined memory accesses are secret-dependent and require transfomations.

As shown in Figure 4.3b, after the function cloning, only the `__cloned_updateBuf` function continues the taint propagation. The output of `updateBuf` is no longer tainted, and consequently, the return value of function `A` is also untainted. This prevents unnecessary taint propagations and reduces over-tainting.

Also, for the example in Figure 4.4, two versions of cloned function corresponding to two function calls in `main` function are generated after employing the function cloning. Only one memory access in the function `__cloned_accessComp.1` should be transformed, which reduces code positions that should be transformed and improves the performance. Besides, if there is a call the `accessComp` without tainted argument, none of the memory accesses needs to be transformed.

We combine function cloning with value-flow tracking for our analysis. Using a value-flow graph, we propagate taint while identifying function calls containing tainted arguments. These function calls should call cloned variants of the underlying functions. Therefore, we clone the called functions of these calls. In other words, we only clone functions that are called with tainted arguments. Tainted arguments of cloned functions are marked as blinded and also serve as taint source for the next round of analysis. We clone these functions until no further cloning is necessary. For library functions, we do not clone them but identify output value as sensitive if they have sensitive inputs. Though cloned functions expand the analyzed program, function cloning brings benefits in both analysis accuracy and performance.

Algorithm 1: The algorithm that collects taint sources from user annotations at the beginning.

```
Input: A program module $M$ with user-marked tainted data
Output: $TaintedValues$: a set that contains tainted values. $TaintedPointers$: a
        set that contains tainted pointers. $TaintSource$: a set that contains
        taint sources.
1  Function CollectTaint(M):
2      TaintSource, TaintedPointers, TaintedValues = {};
3      foreach GV : M.GlobalVariables do
4          if GV.isMarkedBlinded then
5              TaintedPointers.add(GV);
6      foreach Func : M.Functions do
7          foreach Para : Func.Parameters do
8              if Para.isMarkedBlinded then
9                  if Para.isPointer then
10                     TaintedPointers.add(Para);
11                 else
12                     TaintedValues.add(Para);
13     TaintSource = TaintedPointers ∪ TaintedValues;
```

4.1.3 Taint Tracking Overall Design

We combine two parts in this section together to form the overall design of our taint-tracking.

At the start of the taint tracking analysis, our analysis tool traverses through each global variable and function parameter in the program module to gather all the blinded data and pointers.

Blinded data refers to user-annotated non-pointer data that are expected to be blinded on the hardware and serve as direct taint sources. Blinded pointers, on the other hand, are user-annotated pointers whose underlying non-pointer data should be used as blinded data. They are considered as `TaintedData` and `TaintedPointers` accordingly in the taint tracking.

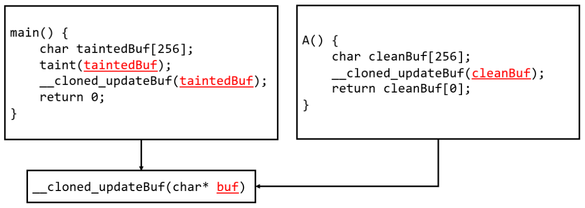
We define the process above as `GetTaintSources` in Algorithm 1, which will be used in other algorithms below. `GetTaintSources` accepts a program module as input, and outputs `TaintSource`, which contains two sets of data: `TaintedData` and `TaintedPointers`.

After obtaining the output of the function in `GetTaintSources`, we do the taint tracking with an inter-procedural value-flow graph, while generating a set of potential callsites that should be considered during the function cloning. We only clone a function if it satisfies both conditions below:

- The function is called with tainted pointers or data as actual arguments.
- The tainted argument is not the output of a function that should be cloned but not cloned yet.

The first condition is set because as discussed in Section 4.1.2, we clone functions to differentiate various combinations of taint status of arguments.

The second conditions is important for the accuracy of the analysis. For example, we expect function cloning to work as Figure 4.3 to improve the accuracy. If the second condition is not considered, since `cleanBuf` is tainted during the taint tracking, `A` uses the cloned version of `updateBuf` after the function cloning. The result of function cloning in this case is shown in Figure 4.5. Both functions call the cloned `updateBuf`. Function cloning fails to differentiate different contexts, thus cannot bring the benefits described in Section 4.1.2. However, if we consider the second condition (assuming `taint` function does not need to be cloned), then the call in function `A` should not be considered as function cloning candidates, since the argument is tainted by `updateBuf`, which should be cloned, but not cloned during the taint tracking.



```
main() {
    char taintedBuf[256];
    taint(taintedBuf);
    __cloned_updateBuf(taintedBuf);
    return 0;
}
```

```
A() {
    char cleanBuf[256];
    __cloned_updateBuf(cleanBuf);
    return cleanBuf[0];
}
```

```
__cloned_updateBuf(char* buf)
```

Figure 4.5: If the second condition is not considered, the call to `updateBuf` in function `A` will call the cloned version.

Taking both taint tracking and function cloning preparation into consideration, we have Algorithm 2.

Algorithm 2: Taint tracking that propagates taint while collecting functions to be cloned.

```
Input: A program module $M$ with user-marked tainted data
Output: $TaintedCallBases$: a set that contains function calls that should be
        considered for function cloning. $TaintedValues$: a set that contains
        tainted values. $TaintedPointers$: a set that contains tainted pointers.
1  Function TaintTracking(M):
2      VFGNodeWorkList = queue();
3      GenerateVFG($M$);
4      CollectTaint($M$);
5      foreach TaintNode : TaintSource do
6          VFGNodeWorkList.push({$nil$, GetVFGNode(TaintNode), $false$});
7      while VFGNodeWorkList not empty do
8          PredNode, CurNode, ReachedFuncCall = VFGNodeWorkList.front();
9          VFGNodeWorkList.pop();
           /* Add VFGNode into the TaintedFuncCall set if it should be
              considered during the function cloning.                     */
10         if CurNode.isCallSiteNode() and ReachedFuncCall is $false$ then
11             ReachedFuncCall = $true$;
12             TaintedCallBases.insert(CurNode);
           /* Collect tainted pointers and values.                         */
13         if CurNode is StoreNode and CurNode.StoredValue() in (TaintedValues or
            TaintedCallBases) then
14             TaintedPointers.add(CurNode.StoredValue());
15         else if CurNode is Pointer and PredNode not in TaintedValues then
16             TaintedPointers.add(CurNode);
17         else
18             TaintedValues.add(CurNode);
           /* Add successor nodes into the VFGNodeWorkList.                */
19         foreach Successor: curNode.successors do
20             if {CurNode, Successor} not visited then
21                 VFGNodeWorkList.add({CurNode, Successor, ReachedFuncCall});
```

At the beginning of each round of taint tracking, Algorithm 1 is called to generate a list of taint sources. A queue consisting of tuples is then built for a breadth-first traversal. The pushed-in tuples contain the information including the node that propagates taint to the `TaintNode` and whether the taint tracking has reached a function call. At the beginning, all the TaintNodes are taint sources, so the predecessor does not exist. Also, the taint tracking has not reached any function calls, so the last value should be false.

Then the taint tracking traverses the value-flow graph. It checks whether the current node (the node popped from the worklist queue) should be considered during the function cloning. If a call should be considered for function cloning, then function calls that are dependent on its output should not be the candidate for function cloning. So the algorithm will set the `ReachedFuncCall` value to be true to stop adding successors of this function into `TaintedCallBases`.

The taint tracking algorithm distinguishes between a tainted pointer and a tainted value (data) to add nodes into different sets. There are two cases where nodes are added into the `TaintedPointers` set:

- The current node is a store node, and the stored value is a tainted pointer or tainted value, then the address of this node is added into tainted pointer set.
- The current node is load, arithmetic operations, assignment, etc., and the result is a pointer, while the predecessor is not a tainted value.

Other nodes are added into the `TaintedValue` set.

Finally, each successor nodes of current node in the value-flow graph are added into the `VFGNodeWorkList` if the combination of it and the current node is not visited before.

The function cloning pass iteratively calls `TaintTracking`, as illustrated in Figure 4.1, and does function cloning until no function cloning can be made. As presented in Algorithm 3, it checks arguments of callbases in `TaintedCallBases` to see if any tainted arguments are not marked by the user or the function cloning. If there is, then it calls `CloneFunction` with the callbase and a set of arguments that should be marked as blinded. `CloneFunction` gets a copy of called function. In addition, it changes the name with a numeric suffix and a prefix to indicate that this is a cloned function. Also, it marks the formal parameters corresponding to `ChangedArgs` to be blinded, i.e., taint sources for the next round of taint tracking.

Algorithm 3: Taint tracking with function cloning.

```
Input: A program module $M$ with user-marked tainted data. $TaintedCallBases$:
       a set that contains function calls that should be considered for function
       cloning. $TaintedValues$: a set that contains tainted values.
       $TaintedPointers$: a set that contains tainted pointers.
Output: A program module $M$ with functions cloned.
Function TTFC(M):
    Changed = true; while Changed do
        TaintTracking(M); Changed = false;
        foreach Instr : TaintedCallBases do
            ChangedArgs = {};
            foreach Arg : Instr.Args do
                if Arg in TaintedValues and Arg not marked blinded then
                    Changed = true;
                    ChangedArgs.add(Arg);
                else if Arg.isPointer then
                    foreach Obj : Arg.pts do
                        if Obj in TaintedValues then
                            Changed = true;
                            ChangedArgs.add(Arg);
            if not ChangedArgs.empty() then
                CloneFunction(Instr.CalledFunction, ChangedArgs);
```

After the combined taint tracking and function cloning, `TaintTracking` can be called again to generate a list of `TaintedValues`. Instructions using elements in the `TaintedValues` for branching or as memory access addresses can be considered as potential violations.

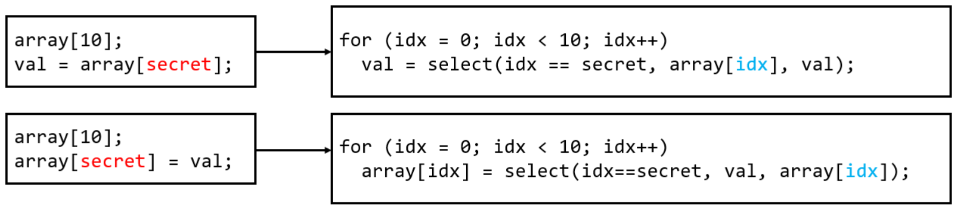
4.2 Transformations

Our transformations include array access expansion, Constantine CFL and branchless select. We use array access expansion to reduce secret-dependent memory accesses and use Constantine CFL to reduce secret-dependent branchings. As these two transformations introduce select instructions that use secrets as conditions, which will be lowered to secret-dependent branchings in RISC-V assembly, we also need transformations to make these select instructions branchless. Our transformations does not cover all the code patterns, such as secret-dependent memory accesses that are not on arrays, loops with secret-dependent sizes, etc.

4.2.1 Array Access Expansion

As discussed in Section 2.1, using a secret as index to access array can expose the secret to cache side channel. Our design expands each secret-dependent array accesses by accessing each of the elements, but only writing the value when the correct index is accessed. Figure 4.6 shows the main idea of array access expansions for load and store when the index is a secret. After the expansion, all the array accesses in the program are not secret-dependent.

For a secret-dependent load, we read each element in the array and select the value only when the index is `secret`. For a secret-dependent store, we store the value only when the index is `secret`. Otherwise, the original value is stored.



```
array[10];
val = array[secret];
```

```
for (idx = 0; idx < 10; idx++)
  val = select(idx == secret, array[idx], val);
```

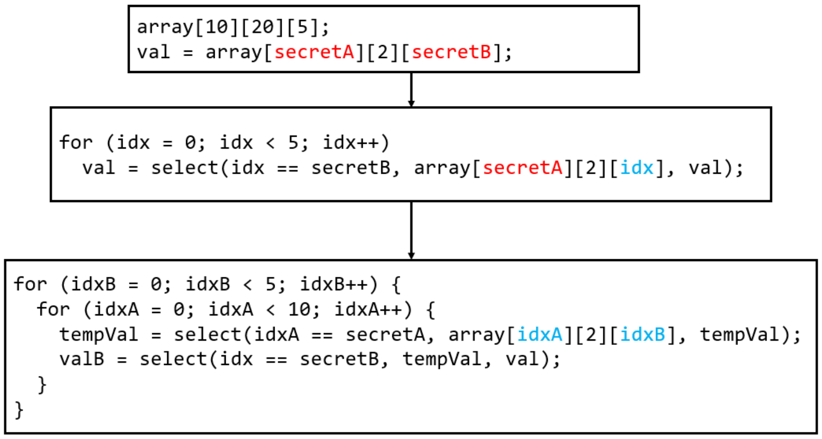
```
array[10];
array[secret] = val;
```

```
for (idx = 0; idx < 10; idx++)
  array[idx] = select(idx==secret, val, array[idx]);
```

Figure 4.6: Array access expansion on secret-dependent load/store: read all the elements in the array and load/store only the element in the correct index.

With information from taint tracking, our array access expansion pass walks through memory access instructions in the program. If the address of memory access instruction can be found in `TaintedValues` set and the memory access is an array access, then this access is expanded.

There are also special cases where array accesses are on multi-dimensional arrays and there are multiple secret-dependent indices. In this case, we iteratively expand each secret-dependent addresses following the main idea described in Figure 4.7. We first expand the access dependent on `secretB`, which generates a new secret-dependent access, but with only one secret index `secretA`. We then expand the new memory access in the same manner as before.

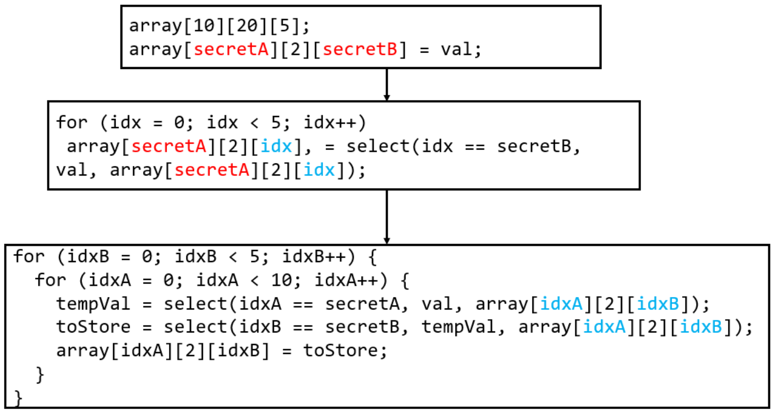


```
array[10][20][5];
val = array[secretA][2][secretB];
```

```
for (idx = 0; idx < 5; idx++)
  val = select(idx == secretB, array[secretA][2][idx], val);
```

```
for (idxB = 0; idxB < 5; idxB++) {
  for (idxA = 0; idxA < 10; idxA++) {
    tempVal = select(idxA == secretA, array[idxA][2][idxB], tempVal);
    valB = select(idx == secretB, tempVal, val);
  }
}
```

(a) Expansion for multi-dimensional array load.



```
array[10][20][5];
array[secretA][2][secretB] = val;
```

```
for (idx = 0; idx < 5; idx++)
 array[secretA][2][idx], = select(idx == secretB,
val, array[secretA][2][idx]);
```

```
for (idxB = 0; idxB < 5; idxB++) {
  for (idxA = 0; idxA < 10; idxA++) {
    tempVal = select(idxA == secretA, val, array[idxA][2][idxB]);
    toStore = select(idxB == secretB, tempVal, array[idxA][2][idxB]);
    array[idxA][2][idxB] = toStore;
  }
}
```

(b) Expansion for multi-dimensional array store.

Figure 4.7: Multi-dimensional array secret-dependent access expansion.

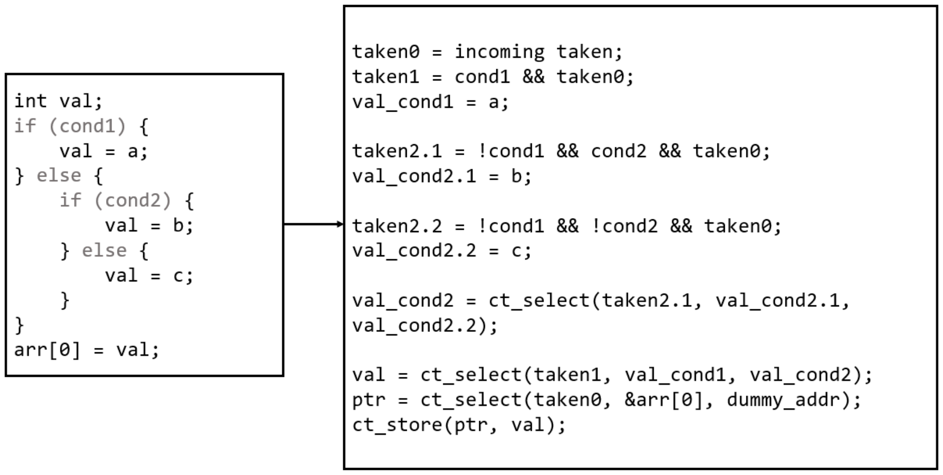
4.2.2 Control-flow Linearization

For control-flow linearization, we make use of Constantine CFL functionalities. As discussed in Section 2.4, Constantine linearizes the control flow to be branchless. An example in Figure 4.8 describes the main idea of Constantine CFL.

When having a sensitive condition for branching, Constantine removes all the branches. Transformed programs complete all the operations on both taken and non-taken branches. To keep the functionality of the original program, all the operations only affect the virtual registers of LLVM IR until exiting both branches.

At the exit point of branches, transformed programs select the result to be effective with a constant-time select (`ct_select`). Effective results then flow to memory and virtual registers that are used later.

Also, address calculations are replaced with a constant-time select based on the taken condition. The path that is not taken get a dummy address in address calculations. Load-/stores using these selected addresses will be wrapped to access the same objects on both take/not-taken branches.



```
int val;
if (cond1) {
    val = a;
} else {
    if (cond2) {
        val = b;
    } else {
        val = c;
    }
}
arr[0] = val;
```

```
taken0 = incoming taken;
taken1 = cond1 && taken0;
val_cond1 = a;

taken2.1 = !cond1 && cond2 && taken0;
val_cond2.1 = b;

taken2.2 = !cond1 && !cond2 && taken0;
val_cond2.2 = c;

val_cond2 = ct_select(taken2.1, val_cond2.1,
val_cond2.2);

val = ct_select(taken1, val_cond1, val_cond2);
ptr = ct_select(taken0, &arr[0], dummy_addr);
ct_store(ptr, val);
```

Figure 4.8: Control-flow linearization of Constantine.

We employ the control-flow linearization passes from Constantine. Constantine records the control-flow instructions that should be tainted by dynamic profiling and marks these instructions with metadata. To provide information to Constantine passes, we mark all the branches with conditions in the `TaintedValues` in the same style as Constantine.

4.2.3 Branchless Select

Though `select` can be lowered as branchless `cmov` in x86 architecture, it still generates branch instructions in RISC-V architecture. We refer to the linearization in [51], which transforms a `select` into branchless arithmetic operations.

A `select(cond, valTrue, valFalse)` outputs `valTrue` if cond is true, `valFalse` if cond is false. Each select instruction in the program is transformed to be the branchless select as shown in Listing 4.1.

```
    neg = -cond;
    /*
    Extend the negated boolean cond to be the same size of values.
    1 -> 111....1
    0 -> 000....0
    */
    extend(cond, sizeof(valTrue);
    /*
     If cond is true, neg & (valTrue ^ valFalse) ^ valTrue = valTrue.
     If cond is false, neg & (valTrue ^ valFalse) ^ valTrue = valTrue ^
   valFalse ^ valTrue = valFalse.
    */
    return neg & (valTrue ^ valFalse) ^ valTrue;

```

Listing 4.1: Linearized branchless select.

The pass that generates branchless selects runs after other transformations to handle `select` instructions generated by transformations.

Chapter 5

Implementation

We implemented our compiler-based tool on LLVM-11. We extended Clang [46] front-end with a new attribute to allow user annotations of blinded data and pointers. We implemented taint tracking analysis, array access expansion and branchless `select` as LLVM passes. For taint tracking, we utilize the SVF library [2] to generate inter-procedural value-flow graph. Also, we forward-ported Constantine passes to LLVM-11 to utilize its CFL passes.

The workflow for using our tool is described in Figure 5.1. The input to the compiler should be the source code with user annotations on blinded data. All the source code are then put into tools such as Whole-Program LLVM (WLLVM)[37] to build a whole-program LLVM bitcode file, which allows using passes on the whole-program. We then utilize LLVM opt tool to apply analysis and transformation passes on the bitcode. The analysis will produce a list of code positions potential violations that cannot be eliminate after all the code transformations, which can be used as a reference for manual changes. The transformed code is then feed into clang to generate the executable compatible with RISC-V.

5.1 Taint Tracking

5.1.1 Taint Source Annotation

As described in Section 4.1, we require users to annotate global variables and function parameters to indicate taint sources. We add a Clang attribute `__attribute__((blinded))`

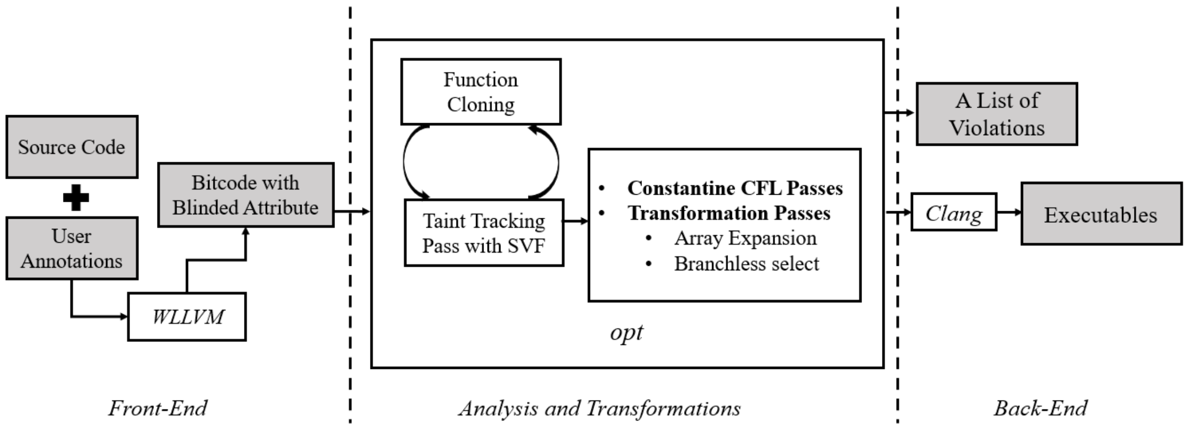


Figure 5.1: Workflow of our compiler-based tool.

to the front-end so users can mark blinded data and pointers with this attribute. This new attribute can be added to global variables and function formal parameters. Users can mark both non-pointer data and pointers, as shown in Listing 5.1. In IR code, attributes can be retrieved using LLVM `hasAttribute()` API.

- **Non-pointer blinded data**: the marked data will be the taint sources, as shown in Line 2 of Listing 5.1.
- **Blinded pointers (also the nested pointers)**: the data pointed by the pointer are taint sources. Intuitively, we assume that users do not intend to directly mark an address as blinded, as blinded address cannot be referenced for load and store when BliMe policy enforced. We always treat the first level of non-pointer data as blinded data, as shown in Line 9 of Listing 5.1.

```
1 /* Global variable keyNum is blinded data. */
2 __attribute__((blinded)) keyNum;
3 
4 /* Global variable blinded_ptr_2 is a blinded pointer. */
5 __attribute__((blinded)) int** blinded_ptr_2 = ptr;
6 /* Both blinded_ptr_1 and blinded_ptr_2 are not blinded data, but blinded
     pointer. */
7 int*  blinded_ptr_1 = *blinded_ptr_2;
8 /* blinded_1 is regarded as blinded data, thus the taint source for taint
     analysis. */
```

```
int    blinded_1 = *blinded_ptr_1;
```

Listing 5.1: Example: user annotations on blinded data and pointers.

5.1.2 Taint Tracking

We rely on SVF to build points-to set and value-flow graph and added SVF library into LLVM-11 framework. Then, we implemented the `BlindedTaintTracking` pass using the functionality from SVF library. `BlindedTaintTracking` pass is an analysis pass that implements the Algorithm 2, which can be invoked by other passes to provide analysis results.

Our implementation generates SVF analysis results including points-to analysis, PAG and SVFG when running the analysis. Listing 5.2 shows the code that build SVFG and uses of each analysis.

```
 // Build SVFModule with the program module M.
 SVF::SVFModule* svfModule = SVF::LLVMModuleSet::getLLVMModuleSet()->
buildSVFModule(M);
 // Build PAG, which is used when converting between SVFG nodes and
LLVM values.
 SVF::PAGBuilder pagBuilder.
 pag = pagBuilder.build(svfModule);
 // Build Andersen's Analysis, which is used to generate SVFG.
Andersen's analysis also helps in function cloning.
 ander = new SVF::Andersen(pag);
 ander->analyze();
 // Build value-flow graph.
 SVF::SVFGBuilder svfBuilder(true);
 svfg = svfBuilder.buildFullSVFGWithoutOPT(ander);
```

Listing 5.2: Code for building SVFG.

We implemented the algorithm described in Algorithm 2. At the start point of the `BlindedTaintTracking` pass, all the analysis results, including taint information and SVF analysis results are cleared. It then builds SVFG, as described in Listing 5.2 and collects taint sources by checking the `blinded` attributes of function parameters and global variables. It then goes through a breadth-first traversal to the SVFG to build the `TaintedValues`, `TaintedObjIDs` and `TaintedCallBases` sets.

Though SVF handles value-flow in most LLVM IR instructions, the flow from the condition to the result of `select` instructions is not reflected. However, it should be considered as our branchless `select` generates an output dependent on the condition. As a result, we also add this dependency during the analysis with the method similar to Line 19-21 in Algorithm 2. As shown in Listing 5.3, since LLVM IR by design preserves def-use relationship for registers, we check all the users of tainted data and propagate to a user if it is a `select` instruction and uses tainted data as condition.

```
for (auto valUser : valNode->users()) {
  Value* NValUserVal = dyn_cast<Value>(valUser);
  if (isa<SelectInst>(NValUserVal)) {
    // Convert a LLVM value to a VFGNode. This is done by finding the corresponding node on PAG, and then on SVFG.
    const SVF::VFGNode* userVFGNode =   LLVMValue2VFGNode(NValUserVal);
    // Check if the pair {vfgNode, userVFGNode} exists in the set. If not, this pair is not visited and should be inserted into the vfgNodeWorkList.
    if (NValUserVal != nullptr) {
      if (!handledNodes.count({vfgNode, userVFGNode})) {
        handledNodes.insert({vfgNode, userVFGNode});
        vfgNodeWorkList.push({vfgNode, userVFGNode});
      }
    }
  }
}
```

Listing 5.3: Code for adding flow of `select` conditions into taint propagation, with some irrelevant checks ommited.

After the taint tracking, the pass goes through all the instructions and check if they are memory access instructions that use tainted values as addresses or conditional branchings that use tainted values as conditions. If so, these instructions will be added into a `Violations` set.

In summary, `BlindedTaintTracking` pass provides following information for further analysis or transformation:

- `TaintedValues`: Tainted IR registers. These registers are collected during the taint tracking described in Algorithm 2.
- `TaintedObjIDs`: SVFG node IDs of tainted objects.
- `TaintedCallBases`: Function calls that should be considered during the function cloning.

- Violations: conditional branching instructions using registers in `TaintedValues` as condition and memory access instructions that use registers in `TaintedValues` as address. These are the instructions that require transformation.

- SVF results: Though violations and taint tracking results are provided, we still preserve points-to analysis and SVFG from SVF. These results help function cloning to handle pointers, as will be described in Section 5.1.3.

5.1.3 Function Cloning

As described in Algorithm 3, at the start of the function cloning, `BlindedTaintTracking` pass is called. Then function cloning needs to find functions that should be cloned and clone these functions properly.

For the first step, we iterate through all the `TaintedCallBases` and check each argument. If at least one of the argument is tainted, but not with a `blinded` attribute, then the called function of the callbase should be considered for cloning. We check the taint status of arguments by checking if they are in the `TaintedValues` set or if their points-to set, which can be queried with `ander getpts(NodeID)`, contains objects in `TaintedObjIDs`.

For the second step, we need to avoid name conflicts. We added a number suffix to the function name to indicate the tainted argument of the function. For arguments from 1 to n, we calculate the number with:

$$\sum_{i=1}^{n} \begin{cases} 2^{i-1} & \text{if } \mathrm{arg}_i \text{ is blinded} \\ 0 & \text{otherwise} \end{cases}$$

E.g., a function call with the first and the third arguments tainted should call a cloned function with suffix 5.

Since LLVM generates numeric suffix for functions with the same name in a module, we also added a prefix `_cloned_` at the beginning of the function name to avoid conflicts from this.

With function name as a reference to which arguments are tainted, we are able to check if a cloned function of a certain version exists already in the program module without having to cache anything. Function calls with the same tainted arguments can be retrieved without having to clone the function again.

We use `LLVM::CloneFunctionInto()` API to clone the function. After the function is cloned with proper name, we add `Blinded` attribute to each parameter corresponding to a tainted argument. Then we replace the called function of the current callbase with the cloned function.

In the subsequent rounds of function cloning, these functions can be identified as exempt from cloning. Also, taint tracking collects these parameters as taint sources.

5.2 Transformation

5.2.1 Array Access Expansion

We implemented array access expansion in a LLVM pass `BlindedInstrConversion`. This pass uses the result from `BlindedTaintTracking` pass.

With all the violations collected as the output of `BlindedTaintTracking` pass, `BlindedInstrConvers` is able to find all the memory access instructions that should be transformed. We collect these instructions into worklist. Then the pass iterates through instructions in the worklist.

The arrays accesses we can transform use an address of GEP instruction or `bitcast` instruction that casts GEP to the wanted type. GEP instruction is the instruction that calculate the memory access address in IR. GEP instructions contain type information, especially for structs and arrays, that we can use to expand the access.

For example, as shown in Listing 5.4, when accessing the element in the array in `struct MyStruct`, we can retrieve the size of the array by analyzing the type of `MyStruct` when we know which index of GEP is tainted:

```
%MyStruct = type { i32, [10 x i32], double }
%myStructPtr = alloca %MyStruct

%arrayElementPtr = getelementptr %MyStruct, %MyStruct* %myStructPtr, i32
    0, i32 1, i32 %secret
```

Listing 5.4: Example: GEP for array expansion.

When having an array access using tainted address, we get the specific tainted indices on the GEP. Then we get the type of the address to be expanded and expand it if we can get the size from the type information:

```
// Get the type of the original pointer or structure from which the GEP
    is derived.
```

```
Type* SourceType = GEP->getSourceElementType();
// Get the corresponding type to tainted index.
// NewIndices is the tainted index.
Type* arrType = GetElementPtrInst::getIndexedType(GEP->
    getSourceElementType(), NewIndices);
```

Listing 5.5: Example: GEP for array expansion.

With the size information, we expand the array access using the LLVM `IRBuilder`. We show the load expansion in Listing 5.6. We use `IRBuilder` to build the loop (using the array size as loop size) we described in Figure 4.6, which contains a constant-time selection to choose the value on the tainted index. We then replace all the uses of the loaded results with the result from our expanded loop. Bitcasts between GEP instructions and load instructions should be added since the loaded type and the GEP type can be different in this case. Array access with multiple tainted indices are supported by adding new generated load/stores into the worklist, which means they will be examined to see if any index is tainted and expanded recursively. Store was implemented in the similar manner.

```
    // Load first element of array for use in loop body PHI
    // If there is a bitcast before load, then the GEP type will be
   different from loaded value. So we need to add another bitcast for
   memory access.
    ...
    if (GEP->getResultElementType() != LIResultType) {
      auto BitCastInst = Builder.CreateBitCast(LoadAddr, LI->
   getPointerOperandType());
      LoadAddr = BitCastInst;
    }
    auto GEPLoad = Builder.CreateLoad(LIResultType, LoadAddr);

    // To handle the case that there are multiple tainted index in a
   memory access.
    LoadWorkList.push_back(static_cast<LoadInst*>(GEPLoad));
    ...

    // Similarly, create load to load elements in the rest of the loop
   and add it to the worklist. Bitcasting omitted.
    GEPLoad = Builder.CreateLoad(LIResultType, LoadAddr);
    LoadWorkList.push_back(static_cast<LoadInst*>(GEPLoad));
    // Select the loaded value if the current load is accesing the
   TaintedIdx.
    auto SelectCmp = Builder.CreateCmp(CmpInst::ICMP_EQ, InducVar,
   TaintedIdx);
```

```
    auto SelectRes = Builder.CreateSelect(SelectCmp, GEPLoad, ArrElement)
   ;

    // increment induction variable
    auto AddRes = Builder.CreateNSWAdd(InducVar, One);
    InducVar->addIncoming(AddRes, LoopBodyBB);

    // Branch to end after we iterate over all array elements
    Value *ArrSizeVal = ConstantInt::get(Context, ArrSizeAPInt);
    auto LoopCondCmp = Builder.CreateCmp(CmpInst::ICMP_SLT, InducVar,
   ArrSizeVal);
    Builder.CreateCondBr(LoopCondCmp, LoopBodyBB, AfterLoopBB);
```

Listing 5.6: Array access expansion implementation with some code omitted.

However, our current implementation is unable to handle the variable-size array. If the array is passed as a pointer, then we are unable to get the array size to expand the array. We leave the discussion to possible solutions to Chapter 7.

5.2.2 Branchless Select

The transformation pass walks through each `select` instructions and converts them into the branchless version. We implement the branchless select in Section 4.2.3 using the `IRBuilder`. The core code is shown in Listing 5.7.

```
    auto *NegCondVal = Builder.CreateNeg(CondVal);
    auto *MaskVal = Builder.CreateSExtOrBitCast(NegCondVal,MaskType);
    auto *TmpXor = Builder.CreateXor(TrueValue, FalseValue);
    auto *TmpXorMasked = Builder.CreateAnd(MaskVal, TmpXor);
    auto *TmpResVal = Builder.CreateXor(TmpXorMasked, FalseValue);
```

Listing 5.7: Example: GEP for array expansion.

5.2.3 Control-flow Linearization

We forward-ported some passes in Constantine from LLVM-9 to LLVM-11 and SVF 2.6, including normalization passes and CFL. Also, we modified scripts in Constantine to use passes we need, including the normalization passes, CFL passes and other optimization passes used by Constantine. For RISC-V compatibility, since CFL requires a compiled CFL library to be linked with the bitcode, we added script to compile the Constantine CFL library into bitcode compatible with RISC-V.

To provide information to Constantine, we added `!t` metadata to conditional branches using tainted data as the condition.

A compatibility issue arises between Constantine and our branchless select due to the transformation performed by Constantine, which replaces many `Phi` and branch instructions with `select` instructions. However, this transformation can result in `select` instructions with `undef` inputs. When expanding these instructions using our transformation, it leads to the generation of arithmetic operations involving `undef` values. To avoid potentially unsafe optimizations on `undef` that disrupt the original functionality of the program, we mitigate this issue by replacing `undef` values with 0.

Chapter 6

Evaluation

In this chapter, we evaluated the effectiveness of analysis and each transformation with experiments. and summarized the extent to which our tool meets the requirements outlined in Section 3.2 with results from the experiment.

We evaluated our tool using two benchmarks: the OISA benchmark [55] and the WolfCrypt benchmark [6] provided by WolfSSL.

The OISA project is a RISC-V hardware design with data-oblivious ISA extension that allows tracking user-marked sensitive data. OISA also includes extended instructions that further enhance the data-oblivious computation in RISC-V, such as conditional move and oblivious load/store. The OISA benchmark tests the applicability of OISA hardware on data-processing programs that require constant-timeness. It consists of simple implementations of basic data processing algorithms, including binary search, matrix multiplication, etc. We chose this benchmark because data processing represents a common use scenario of outsourced computing. Also, OISA has similar user scenario as BliMe.

The WolfCrypt benchmark includes implementations of cryptographic algorithms. This benchmark was chosen because cryptographic libraries are common attack targets of side-channel attacks.We divided WolfCrypt benchmark into separate tests.

We compiled benchmarks into statically-linked RISC-V binaries. The compilation follows the steps in Figure 5.1. We compiled benchmarks into bitcode files. As our compiler and Constantine were built in two separate docker containers, we ran our passes in our container, then ran Constantine passes in Constantine container. As Constantine crashes on all the WolfCrypt benchmark tests, we did not ran Constantine transformations on WolfCrypt benchmarks. We added a function that printed out all the violations reported by the compiler to output files so we were able to analyze the analysis result. After the analysis and transformations, we used Clang to compile the bitcode files into executables.

We ran the benchmarks on a special BliMe Spike implementation. This implementation does not crash upon encountering a violation, but instead prints out all the violations so we can check if our analysis reports all the violations on the hardware.

We evaluate the soundness of the analysis (**R-1**) in Section 6.1 by checking if our analysis indicates all the warnings in the BliMe hardware. We then evaluated the rest of the requirements in Section 6.3.

We did not measure the performance overhead of transformation since we did not get Constantine to transform several tests in each benchmark. As an estimation, we expect significant overhead on software with secret-dependent memory accesses to large arrays, as the expanded array accesses will access all the elements in the array.

6.1 Soundness of the Analysis

To evaluate the soundness of the analysis, we ran the unmodified OISA code on BliMe Spike, collected warnings from the hardware and found source code positions of these warnings by manually examining the disassembled binaries. We then compared these code positions with the result from the analysis pass.

Based on our analysis, the violations caused by pre-compiled libc cannot be caught by our analysis, as our analysis runs on source code IR. The `dnn` uses an exponentiation calculation in libc, which contains secret-dependent branchings (listed in Listing 6.1). The output of this function is, as a result, secret-dependent. We replaced this function with an approximation that contains no secret-dependent branches. Except this, our analysis finds all the violation reported by the hardware. As a result, though we cannot formally prove the soundness of our analysis, the evaluation results support the argument that our analysis fulfills the soundness (**R-1**) requirement.

However, our analysis has certain limitations beyond the case of pre-compiled libraries. Firstly, it may not identify all the code positions that require transformation, as will be discussed in Section 6.3. Additionally, our analysis is an over-approximation, as shown in Table 6.1. The table does not show the number of warnings in libc.

The over-tainting in the OISA benchmark is mostly caused by a lack of field-sensitivity in our analysis. An example is shown in Listing 6.2. In this example, since our analysis is not field-sensitive, though only the `heap` field of `pq` contains secret data, the whole

Test	Results from Analysis		Results from Hardware	
	Memory Access	Branch	Memory Access	Branch
binary_search	1	3	0	2
dijkstra	6	5	6	0
dnn	0	0	0	0
find_max	0	1	0	1
int_sort	0	1	0	1
kmeans	0	2	0	2
matrix_mult	0	0	0	0
page_rank	6	0	2	0
PQ	3	7	0	3

Table 6.1: OISA benchmark analysis and hardware results show the over-approximation of our analysis. This table shows number of warnings from our compiler analysis and the hardware for each tests in OISA benchmark. Warnings from secret-dependent memory accesses and secret-dependent branches are listed in two columns. A select instruction that uses sensitive data as the condition is considered as a secret-dependent branching since select can be lowered to branches in RISC-V binaries.

structure `pq` gets tainted. As a result, our analysis reports warnings when other fields of `pq`, though not containing secret data, are used as branch conditions. Also, though not shown in OISA benchmark, other design factors of our analysis can cause over-tainting, such as the lack of flow-sensitivity, the over-approximation of points-to analysis from SVF, etc.

```
    if (__glibc_unlikely (abstop - top12 (0x1p-54)
      >= top12 (512.0) - top12 (0x1p-54)))

    if (abstop - top12 (0x1p-54) >= 0x80000000)

    if (__glibc_unlikely (abstop == 0))
```

Listing 6.1: A list of secret-dependent branches reported by the hardware in clib function. abstop is a secret value.

```
    // Store a secret value into pq->dp.
    pq->heap[idx] = secret;

    // pq->block_size should not be tainted, but considered tainted in
   our analysis.
    // i < pq->block_size is considered as a secret-dependent branch.
    for(int i = 0; i < pq->block_size; i++)
```

```
        last_item[i] = new_item[i];
```
Listing 6.2: Example: field-insensitivity causes over-tainting in OISA PQ test.

6.2 Effectiveness of Function Cloning

Function cloning incurs multiple rounds of SVFG construction, which is costly for large programs. Other than this, cloned functions also expand the size of the program.

Regardless of these drawbacks, to evaluate the effectiveness of function cloning, we focus on two metrics: first, we aim to determine if fewer functions become tainted as a result of the taint tracking process; and secondly, we want to check whether fewer violations are detected within the same function after the cloning. The first metric can represent the improvement of analysis accuracy through function cloning. This metric can show if less functions are tainted due to over-tainting after the function cloning. The second metric, on the other hand, can represent the improvement of performance: fewer violations in the same function means fewer positions for the transformation, thus less performance impact. More detailed explanation on the effect of function cloning can be found in Section 4.1.2.

For the OISA benchmark, we noticed that though some functions are cloned, function cloning does not reduce the number of tainted functions or violations within the function. Based on manual examination, the call graphs of OISA benchmark tests are simple, with each function being called mostly just once. Consequently, function cloning does not prove to be effective in enhancing the accuracy under such circumstance.

6.3 Transformation

We evaluated coverage (**R-4**) of the transformation by checking how many hardware-reported violations are removed with transformations. Also, we examine the output of tests to verify the correctness (**R-2**) of the transformation.

6.3.1 OISA Benchmark

We evaluated our tool on simpler OISA benchmark first. The results are shown below in Table 6.2. For dijkstra and PQ tests, Constantine crashed during the normalization phase. In `binary_search` test, one of the branch is a lowered to a `select` instruction in LLVM

IR, so it is transformed into branchless with our transformation. Constantine removes one branch in `find_max` and `kmeans` tests.

Test	Selection	Array	CFL	Violations After	
				Memory Access	Control-flow
binary_search	1	0	0	1	2
dijkstra	0	0	-	6	0
dnn	0	0	0	0	0
find_max	0	0	1	0	0
int_sort	0	0	0	0	1
kmeans	0	0	1	0	1
matrix_mult	0	0	0	0	0
page_rank	0	0	0	0	2
PQ	0	0	-	0	3

Table 6.2: Result of transformations on OISA benchmark. Column 2-4 shows the number of hardware violations removed by transformations (branchless select, array expansion and Constantine CFL). The last two columns show the number of hardware violations after all the transformations.

We summarize the types of the code that cannot be transformed with our tool:

- Though the OISA benchmark contains secret-dependent array accesses, the sizes of arrays are unknown to our analysis. These arrays are dynamically allocated at run-time. As a result, our transformation cannot expand these array accesses. An example can be found in Listing 6.3.

```
1 int __attribute__((noinline)) BinarySearch(__attribute__((blinded))
    int arr[], int l, int r, int x) {
2     while (l <= r) {
3         int m = l + (r-l) / 2;
4         if (arr[m] == x)
5             return m;
6         ...
7     }...
8 }
```

Listing 6.3: Example: an array with unknown size. The size of `arr` is unknown to the compiler at the compile-time, so the compiler cannot expand the array access in Line 4.

- Some of the secret-dependent branches are loops with secret-dependent sizes. Constantine unrolls loops with the information of maximum size of loops from dynamic profiling. Also, based on our examination on Constantine loop unrolling, unrolled loops in Constantine exits when the loop reaches both the estimated maximum size of the loop and met the original exit condition. Checking the latter exit condition is a secret-dependent branch. In other words, if the actual loop size is lower than the estimated maximum size, then the run-time loop size of the unrolled loop is the estimated maximum size. Otherwise, the loop size is still dependent on the secret-dependent size. So Constantine transformed code can introduce secret-dependent branches. Therefore, we did not add Constantine loop unrolling into our transformations.

- The transformations also introduced violations. For example, the chosen results of secret-dependent branches and memory accesses are also secret-dependent. However, our analysis does not take it into consideration during the initial analysis. These code positions will not be transformed as they are not marked by our analysis.

We then manually modified the OISA benchmark code. The OISA project authors also modified their code to achieve constant-timeness, though using extended instructions in their hardware. We investigated their changes on the code and modified tests to be constant-time without using OISA extended instructions. The changes made to the code are listed as follows:

- Adding array expansion code with programmers' knowledge. For example, in test `PQ`, programmers know that the size of the array `pq->heap` is stored in `pq->size`. However, the compiler is not able to obtain this information during the analysis.
- Adding fixed loop sizes to variable-size loops. Loops with known maximum sizes are replaced with for-loops of the known maximum sizes.
- Replacing algorithms. Modifying some of the algorithms into constant-time versions is tricky and sometimes leads to much worse performance. `int_sort` test in OISA benchmark is one of the example. This test implements a merge sort algorithm. Unrolling all the loops in the merge sort can be tricky but the complexity is not better than $O(n^2)$. The OISA code replaced this algorithm with bitonic sort.

We were unable to make all the changes following the methods of OISA paper because the BliMe hardware does not contain all the instructions in OISA hardware. As shown in Listing 6.4, hardware implemented `cmov` in OISA selects `item_to_write` to write into `pq->scan_oram` only when the `item_to_write` is not null. However, we cannot select between the value in `item_to_write` and `pq->scan_oram` then write into `pq->scan_oram` because `item_to_write` can be a null pointer.

```c
for(int j = (1 << i); j < (1 << (i+1)); j++){
   ...
    _cmovn(item_to_write != NULL, item_to_write, &pq->scan_oram[pq->
   block_size * j], pq->block_size);
}
```

Listing 6.4: Example: `cmov` instructions used in OISA benchmark.

6.3.2 WolfCrypt Benchmark

We then conducted evaluation on WolfCrypt benchmark. We tainted secret keys (for encryption algorithms) or plaintext (for digest algorithms) and ran all the benchmarks separately on the BliMe hardware. We tainted keys right before each encryption or decryption, and removed all the taint after finishing each encryption or decryption. This can represent the use cases where users send their keys, leaving the remote hardware to execute the cryptographic algorithms, and receive the results.

As shown in Table 6.3, hardware results indicate that most benchmarks ran without any warnings on hardware with only four exceptions: AES-CBC, AES-GCM, RSA and DH. With our array expansion and branchless select passes, AES-CBC ran on the hardware without warnings. AES-GCM requires removing the blindedness of authentication tag to run without warning. However, DH and RSA still triggered hardware warnings after our compiler transformations.

	Tainted Value	w/o Transformations	with Transformations
AES-CBC	Key	n	y
AES-GCM	Key	n	y
RSA	Private key	n	n
Chacha20	Key	y	-
HMAC	Key	y	-
SHA	Plaintext	y	-
DH	Private key	n	n

Table 6.3: Result of transformations on WolfCrypt benchmark tests. The third column shows whether the test runs without hardware warnings without any transformations. The last column shows whether the test runs without hardware warnings after transformations of our compiler.

Since forward-ported Constantine crashes on all the WolfCrypt tests. We are manually transforming the WolfCrypt tests based on the methodology of Constantine. Due to complicated branches in DH and RSA implementation, we failed to got DH and RSA tests correctly running on BliMe hardware, we summarized the main categories of manual transformations we made below:

- **Branches for error handling.** WolfSSL contains branches that check for possible errors during the computation, these error checks are checking the secret-data. Also, after checking the errors, WolfSSL sometimes write the result based on the error. These error handlings introduce multiple secret-dependent branches as a consequence. Our manual transformation moved all the error checks to the end of the function.
- **Secret-dependent loop size.** Since the maximum size of the secret key can be quite large, WolfSSL checks each byte to see if the byte contains data and generate the actual size of the data. Then WolfCrypt only computes on the bytes that actually contains data. This implementation introduces several secret-dependent loop size. Since we have the knowledge of the maximum possible size, we can manually unroll the loops that contains secret-dependent loop size. The main idea of this change is further discussed in Section 7.2.1.
- **Secret-dependent `memcpy`.** There are `memcpy`s of which the content depends on a secret data. We replaced such `memcpy`s with a constant-time version manually.

6.3.3 Summary

For the evaluations above, the generated executables are able to run on BliMe hardware, which fulfills the compatibility with RISC-V (**R-5**). Also, the outputs of transformed code remain the same as original outputs before the transformation, so our tool fulfills the correctness requirement (**R-2**). Though our transformations only require users to identify sensitive data, our tool cannot transform all the secret-dependent code and requires manual changes to code. So minimal source code changes (**R-3**), coverage (**R-4**) and compatibility with BliMe (**R-6**) are not fully fulfilled. We will discuss how relaxing **R-3** helps improve the transformations in Chapter 7.

Chapter 7

Discussion

7.1 Limitations of the Analysis

7.1.1 Dependencies on Branch Conditions

As discussed in Section 6.1, though our analysis shows all the code positions that trigger warnings, our analysis does not list all the code positions that require transformations. For example, in Listing 7.1, if the values in `arr` are secret, our analysis can indicate that Line 4 is a secret-dependent branching. However, it cannot figure out that `l` is dependent on secret value `arr[m]`. As a result, our analysis cannot figure out that the `while` loop condition, which is dependent on `l`, is secret-dependent and requires transformations.

This is deemed acceptable when solely aiming to determine whether the source code results in a crash on BliMe, as normal BliMe instances crashes when encountering secret-dependent branches, i.e., they crash on Line 4 without reaching Line 1 and triggering another fault. However, to point out all the code positions that require transformations, our analysis needs to analyze the dependencies on branch conditions rather than only following the value-flow.

```
1    while (l <= r) {
2        int m = l + (r-l) / 2;
3        ...
4        if (arr[m] < x)
5            l = m + 1;
6            ...
```

```
    }
```
Listing 7.1: Binary search example from OISA benchmark. `arr` is considered as an array filled with sensitive data.

7.1.2 Library Functions

As indicated OISA evaluation results, pre-compiled libraries such as `libc` are not analyzed and transformed. As a result, secret-dependent memory accesses and branches in the pre-compiled libraries are not fixed. Also, the analysis cannot figure out if the return value of pre-compiled library functions are sensitive or not.

Though our current tool can compile libraries, e.g., we compiled WolfCrypt library, we still lack support on a common real-life situation: the library developers develop and compile libraries to be with multiple constant-time instances of sensitive functions; the users, on the other hand, use the pre-compiled library directly. For example, a library exponentiation function has three cases of sensitivity of inputs: either the base or the exponent is sensitive, both the base and the exponent are sensitive. Library developers in this case need to provide three versions of constant-time exponentiation functions to users. Excluding binary analysis and transformation tools, at the compiler level, two potential approaches can address this issue.

Firstly, the compiler can replace common library functions, such as `malloc`, `free`, `max`, etc, with a constant-time version. This approach requires manually creating a constant-time version of the library and configuring all the replacements in compiler-level.

Secondly, an alternative approach is to develop tools that compile libraries to generate multiple instances of functions based on the sensitivity of the input data. To achieve this, the compiler must be extended to allow library developers to annotate the instances of sensitive functions based on their expertise. Consequently, when library users compile the source code that uses the library, the compiler can appropriately select the relevant instances of functions.

However, both of the aforementioned approaches fail to address the issue of the analysis being agnostic to the sensitivity of the return values of library functions. One approach is to add special function suffices to indicate the sensitivity of library functions' return values. Alternatively, users may be required to annotate whether a library function call anticipates a sensitive or non-sensitive return value, drawing from their knowledge and context.

7.2 Limitations of the Transformation

We described some code that cannot be transformed by our tool in Section 6.3. Among these code patterns, we discuss possible solutions to loop unrolling and variable size array accesses.

7.2.1 Annotations for Loop Unrolling

Constantine implements loop unrolling with the maximum loop size obtained from dynamic profiling. Despite not utilizing dynamic profiling, user annotations can be leveraged to specify loop sizes, utilizing users' knowledge of the loops.

Listing 7.2 describes the main idea on how user provided information help to unroll the loop. With user-provided loop size, compiler transformation can safely transform the variable-size loop into a fixed size loop with a secret-dependent if-else branch, of which the condition is the original loop condition. Subsequently, Constantine's Control-Flow Linearization (CFL) can be employed to transform the branch and optimize the loop unrolling process.

```c
    // Code before the transformation
    // max_size is the maximum size of the while loop
    int max_size = log(arr_size);
    while (l <= r) {
        int m = l + (r-l) / 2;
        if (arr[m] == x)
            ...
        if (arr[m] < x)
            l = m + 1;
            ...
    }

    // Code after the transformation
    int max_size = log(arr_size);
    for (int i = 0; i < max_size; i++) {
        if (l <= r) {
            int m = l + (r - l) / 2;
            ....
        }
    }
```

Listing 7.2: Binary search example from OISA benchmark. `arr` is considered as an array filled with sensitive data, `arr_size` is the size of the array, `l` is the start position of binary search, `r` is the end position of binary search.

7.2.2 Dynamically-allocated Array Access Expansion

To handle dynamically allocated variable-size array accesses, one approach is to build an object management data structure at the program's start.

The compiler needs to instrument all the memory allocations to store allocated pointers and sizes into the object management data structure. Then, the compiler can instrument secret-dependent array accesses to retrieve the array sizes for array access expansions. This solution allows the compiler to retrieve array sizes at runtime, enabling handling of variable-size arrays.

7.3 Back-end Issues

As discussed by existing works [41], when lowering the source code to assembly code, compiler optimizations can remove programmers efforts that try to make the code constant-time. Even though our tool operates on IR level, it still suffers from this problem if the optimization level is higher than O1. For future work, we consider adding LLVM intrinsics for important components, such as selection, so we can control the back-end behavior to avoid the situation where the optimizations remove the constant-time property.

Chapter 8

Related Work

8.1 Side-Channel Attacks

As discussed in Section 2.1, side-channel attacks covertly leak sensitive data. To the best of our knowledge, [27] is the first work that discusses timing side-channels on cryptographic algorithms to recover the secret key. Since then, there have been various power [38], cache [54][36][32] and timing side-channel attacks introduced, mostly targeting the cryptographic algorithm implementations. In this category, the vulnerabilities can be mitigated with software engineering efforts such as constant-time programming. This thesis is aiming at partially automating the software mitigations on these attacks.

There are also side-channels introduced by microarchitectural design, such as transient execution attacks. Attackers exploit the transient execution to break the security boundaries set by operating systems and hardware. Transient execution vulnerabilities arise from the speculative out-of-order execution of modern processors. These attacks aim to analyze the timing or power effects caused by the speculative executions. Even though speculative executions should leave no effect on processors, flaws in processor designs fail to guarantee this. With design flaws on hardware, status of microarchitectures, such as L1 cache [13][31], line fill buffer[50], etc, are observed by indirectly using precise timer and power measurement. Attackers then use such status to infer the secret information. For example, in a Spectre [13] attack, the attacker trains the branch predictor to access the secret data and use this secret data as an address to access the memory. After accessing the memory, the attacker can use flush and reload [36] techniques to recover the secret data. BliMe [18] paper mitigates such attacks by preventing the memory tagged as blinded from being observable. As a tool aiming at enhancing the usability of BliMe, we leave the mitigation to such attacks to BliMe.

General hardware countermeasures to side-channel attacks also include turning off precise timing and power measurements [13][31] [39], partitioning resources [57][25] or closing certain optimizations [13][31] [39]. These countermeasures are well discussed in the papers that propose the attack methods

8.2 Side-Channel Attack Defenses

Countermeasures to side-channels stemming from software design can be ad-hoc fixes. For example, for AES s-box, people can use more compact boxes to reduce the information attackers can obtain or use masks [48]. These are not the general defenses we aim to discuss in this thesis. We will discuss general countermeasures to side-channel attacks in this section, including hardware-level countermeasures, software-level side-channel detection and repair.

8.2.1 Hardware-level Countermeasures

Hardware provides programmers with data-oblivious primitives that enhance the constant-timeness of software. Transactional memory guarantees that the execution of critical parts, such as memory accesses, are atomic, thus preventing the data leakage from transient execution or concurrent execution. Existing works [21][56] also make use of this hardware feature to protect data preload. Besides, hardware implementations of Oblivious RAM (ORAM) [19][55] empower programmers with ability to access memory without exposing the memory access pattern.

8.2.2 Software Level Countermeasures

Side-Channel Detection

Existing works on side-channel detection and quantification can be categorized into four types: dynamic profiling [28], symbolic execution [15], static analysis [56][53] and formal analysis [17]. Dynamic profiling methods [28] execute programs after marking sensitive data and track the program points of branchings and memory accesses. Dynamic profiling captures actual run-time behaviors. Without proper inputs to programs, dynamic profiling can fail to cover some execution paths. Symbolic execution [15] simulates program execution using symbolic values and try to solve the constraint. Symbolic execution is more accurate than the analysis in the thesis. However, symbolic execution may suffer from paths explosion and take more than hours to run. Related work on formal analysis [17] derives upper bounds of cache side channel leakages with formal proof. However, this analysis does not detect secret-dependent branching. It also takes hours to finish analysis on large programs. Program repair tools such as [53] and [56] use static analysis to track the data dependencies and find code positions that use secret-dependent data for memory access for branching. We also adopt this method in this thesis. More complete lists of such tools can be found in [24]. For practicality, our tool employs static taint tracking to find code positions, which is not accurate as formal and symbolic execution, but much faster for complex inter-procedural analysis.

Code Transformation

Some of the code transformations add noise to performance counter [34] or balance the secret-dependent branches with dummy executions [52]. Besides, we have discussed compiler-assisted IR-level code transformations that generates constant-time code in Chapter 1 and our conclusion is that, none of the existing tools use sound analysis to find all the potential violations and transform programs to be compatible with the BliMe hardware.

Except the repairs on IR-level, as discussed in Section 7.3, back-end optimizations can cause a loss of constant-time property. Some work [7] in the programming language field discussed how to preserve the constant-time property with secure compilation method. Secure compilation approaches can be considered when lowering from IR to assembly.

Others

Cauligi *et al.*. [10] developed domain specific languages FaCT. The corresponding compiler ensures potentially timing-sensitive high-level code are converted into constant-time LLVM bitcode. This method requires users to re-implement the codebase. As a result, it is not ideal as a tool to enhance the usability of BliMe.

Chapter 9

Conclusion

To strengthen the usability of BliMe hardware, we present the design and implementation of a compiler-based tool to generate constant-time code. Our tool uses static analysis to search for code positions that trigger faults on BliMe. Then, our tool employs code transformation techniques including array expansion, CFL and branchless select to make code positions constant-time.

Our evaluation demonstrates the tool's capability in identifying secret-dependent memory accesses and branches and transforming a subset of the code patterns to enable code execution on BliMe hardware. We analyzed and classified the code patterns that are challenging for automatic compiler transformations and proposed manual changes for each category. In conclusion, our work delved into compiler-based approaches to strengthen the practical usability of BliMe hardware. We pointed out the pitfalls in our current attempts and explored future directions to enable the deployment of side-channel resistant BliMe platform.

References

[1] BENCHMARKING WOLFSSL AND WOLFCRYPT. https://wolfssl.jp/docs-3/benchmarks/.

[2] Svf: Static value-flow analysis framework for source code. https://github.com/SVF-tools/SVF.

[3] CVE-2022-4304. https://www.cve.org/CVERecord?id=CVE-2022-4304, 2022.

[4] Dataflow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html, 2023.

[5] OpenSSL. https://www.openssl.org/, 2023.

[6] wolfSSL. https://www.wolfssl.com/, 2023.

[7] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343. IEEE, 2018.

[8] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[9] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

[10] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.

[11] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[14] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE symposium on security and privacy*, pages 45–60. IEEE, 2009.

[15] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. Cipherh: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations.

[16] Valgrind Developers. Valgrind. http://www.valgrind.org/, 2023.

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on information and system security (TISSEC)*, 18(1):1–32, 2015.

[18] Hossam ElAtali, Lachlan J Gunn, Hans Liljestrand, and N Asokan. Blime: Verifiably secure outsourced computation with hardware-enforced taint tracking. *arXiv preprint arXiv:2204.09649*, 2022.

[19] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious ram controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 215–222. IEEE, 2015.

[20] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[21] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, pages 217–233, 2017.

[22] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[23] Helmut Hlavacs, Thomas Treutner, Jean-Patrick Gelas, Laurent Lefevre, and Anne-Cecile Orgerie. Energy consumption side-channel attack at virtual machines in a cloud. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 605–612. IEEE, 2011.

[24] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin 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)*, pages 632–649. IEEE, 2022.

[25] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204, 2012.

[26] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999.

[27] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[28] A. Langley. ctgrind. `https://github.com/agl/ctgrind`, 2023.

[29] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[30] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security Symposium*, volume 19, pages 16–18, 2017.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown:

Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[33] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys (CSUR)*, 54(6):1–37, 2021.

[34] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH computer architecture news*, 40(3):118–129, 2012.

[35] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[36] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.

[37] The Whole program LLVM Project. wllvm: Whole-program LLVM. https://github.com/travitch/whole-program-llvm. Accessed: June 29, 2023.

[38] Mark Randolph and William Diehl. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography*, 4(2):15, 2020.

[39] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[40] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[41] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018.

[42] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst.*, may 2023. Just Accepted.

[43] Luigi Soares and Fernando Magno Quintãn Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210. IEEE, 2021.

[44] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[45] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[46] The Clang Team. Clang 17.0.0git documentation. https://clang.llvm.org/docs, 2023.

[47] The LLVM Project. LLVM: A compilation framework for lifelong program analysis & transformation. https://llvm.org/, 2023.

[48] Stefan Tillich and Christoph Herbst. Attacking state-of-the-art software countermeasures—a case study for aes. In *Cryptographic Hardware and Embedded Systems–CHES 2008: 10th International Workshop, Washington, DC, USA, August 10-13, 2008. Proceedings 10*, pages 228–243. Springer, 2008.

[49] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[50] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[51] veorq. Cryptocoding. https://github.com/veorq/cryptocoding, 2023.

[52] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 667–682. IEEE, 2021.

[53] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[54] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[55] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. *Cryptology ePrint Archive*, 2018.

[56] Rui Zhang, Michael D Bond, and Yinqian Zhang. Cape: compiler-aided program transformation for htm-based cache side-channel defense. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 181–193, 2022.

[57] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy*, pages 313–328. IEEE, 2011.

Glossary

CFL Control Flow Linearization. 16, 25, 28, 30, 37, 54

DFSan DataFlowSanitizer. 10

FHE Fully Homomorphic Encryption. 1, 7

GEP GetElementPointer. 9, 11, 35, 36

HSM Hardware Security Module. 7, 8

IR Intermediate Representation. 8–11, 16, 28, 31–33, 35, 43, 50, 53

ORAM Oblivious RAM. 52

PAG Program Assignment Graph. 10, 11, 32

PC Program Counter. 8, 17

REE Rich Execution Environment. 8

SSA Static Single Assignment. 8, 9, 11

SVFG Static Value-flow Graph. viii, 11, 12, 32–34, 42

TEE Trusted Execution Environment. 7, 8

WLLVM Whole-Program LLVM. 30