Secure Register Allocation for Trusted Code Generation

Priyanka Panigrahi, Vemuri Sahithya, Chandan Karfa, and Prabhat Mishra

***Abstract*—In this paper, we investigate the inherent vulnerability of register allocation in which the variables of a source program are mapped to the registers in hardware. This paper makes three important contributions. Specifically, we show that register allocation is secure if there is no spilling. Next, we show that register allocation with spilling does not preserve the security properties of the source program. Our experimental evaluation using a wide variety of benchmarks demonstrates that register allocation in LLVM is not secure. Finally, we propose a secure register allocation in LLVM that will not introduce additional information leaks in the generated code due to spilling.**

***Index Terms*—Secure Compilation, Register Allocation, Spilling, Taint Analysis, Information Flow, LLVM.**

I. Introduction and Related Work

COMPILATION in embedded system focuses on generating functionally correct assembly under power and performance constraints. It applies various optimizations to improve performance. The primary objective of a compiler is to guarantee that the generated code is functionally equivalent to the source program. However, it is a vital requirement in many embedded and cyber-physical systems (such as safety-critical systems) that the compilation should not introduce any security vulnerabilities. While recent advancements in secure compilation try to ensure that the generated code (target assembly) preserves the security properties of the source program, there is a limited effort in securing compiler transformations. Therefore, it is important to ensure that the generated assembly code preserves the security properties of the source program [1].

One promising direction to ensure secure compilation is to establish that none of the compiler transformations introduce any security vulnerabilities. *In this work, we investigate the security of register allocation.* D'Silva et al. [1] studied various compiler transformations to identify the leaky one. The secure version of dead store elimination and single static assignment (SSA) transformations are proposed by Deng and Namjoshi [2], [3]. Recently, Besson et al. [4] proposed Information Flow Preserving program transformations in which they model the information leak of a program using the notion of attacker knowledge. In [5], they analyze the Information Flow Preservation of register allocation in the CompCert C compiler, where the mapping information of registers and corresponding locations are available. However, such mapping is difficult to obtain in modern compilers like LLVM, in which source variables are renamed, and many temporary variables are introduced in Intermediate Representation (IR). Thus, our work does not take any mapping information from the compiler during the security analysis of register allocation.

We assume the attacker has access to the executable code but not the secret data, and the attacker's goal is to gain information about the secret data. The attacker does have access to the final memory at the end of the program [4] but has no access to direct registers. Assuming the functional correctness of register allocation, this paper tries to answer an important question - *is register allocation secure with respect to preserving information leakage?* Specifically, this paper makes the following major contributions:

- We show that register allocation is secure in the absence of spilling.
- We show that register allocation with spilling is not secure - it introduces additional leaky paths.
- Experimental results using diverse benchmarks demonstrate that register allocation in LLVM is not secure.
- We propose a secure register allocation in LLVM that will not introduce information leakage due to spilling.

II. Security Analysis of Register Allocation

Register allocation is an essential step in a compiler in which the variables of a program are mapped to bluepossibly fewer number of physical registers. Two or more variables can be mapped to a single register if their lifetimes do not overlap [6]. If a single variable is defined multiple times, it may be split to be mapped to two or more different registers for better register usage. However, for moderate-sized programs, it is impossible to map all the variables into registers. Spilling a variable is storing the value of a variable into memory instead of register. In this work, we have analyzed the security issues of register allocation with and without spilling. Note that we are not considering the security effects of register allocation on system cache in this work.

Variables of a program can be partitioned into sensitive, i.e., high security (H), and non-sensitive, i.e., low security (L). An input of a program can be termed as H or L based on its sensitiveness. A program variable or an output is said to be leaky if it leaks any sensitive information. Formally, the information leakage of a program is defined as follows [2]:

Definition 1. (Information Leakage:) Let us have two input variables, I_H and I_L, where I_H is a sensitive input and I_L is a non-sensitive input of a program. Consider two pairs of values ($I_H = a, I_L = c$) and ($I_H = b, I_L = c$) such that $a \neq b$. Suppose for a program, the computation on both the input pairs (a, c) and (b, c) either differ in the sequence of output values or the value of one of the low-security variable differ at their final states when both the computation terminate. In that case, the program is said to leak information, and (a, b, c) is called as a leaky triple for the program.

```
1  foo(m,n)
2  {
3
4    p=get_key();
5    n=m+n;
6    p=p+n;
7    return p;
8
9  }
```

(a) Source Program S

```
1  fooRA(m,n)
2  {
3    r1=get_key();
4    r2=m;
5    r3=n;
6    r3=r2+r3;
7    r1=r1+r3;
8    return r1;
9  }
```

(b) T_1 after RA

```
1  foo(m,n)
2  {
3     p=get_key();
4     store p, pm;
5     n=m+n;
6     load pm, p;
7     p=p+n;
8     return p;
9  }
```

(c) S after Spilling

```
1  fooRA(m,n)
2  { r1=get_key();
3     r2=m;
4     store r1, pm;
5     r1=n;
6     r1=r2+r1;
7     load pm, r2;
8     r2=r2+r1;
9     return r2;}
```

(d) T_2 after Spilling

Figure 1: An example of register allocation

The input of register allocation (RA) is the source program, and the output is transformed program. The inputs (and their security types) and the outputs are the same for both source and transformed programs. In this work, we are concerned about the relative security of RA, i.e., we check if RA introduces any new information leakage to the source program. It may be noted that the source program can still be leaky.

Definition 2. (Relative Security:) A transformed program after RA from a source program is said to be relatively secure if all the leaky triples belong to the transformed program are also belong to the source program.

To identify the information leakage of a program, *static taint analysis* is popularly used. Taint analysis [7] checks the direct and indirect influence of sensitive inputs on each variable at each program location, violating the security properties of the program. It is an over approximate method, i.e. the variables defined inside a tainted condition are also to be tainted due to control flow. Thus, it does not generate false-negative results, i.e., a variable is not tainted but leaking some sensitive content. It may be noted that leaky triple in Definition 1 cannot quantify the leakiness of the program, i.e., a leaky program can be transformed to a more leaky program, but they can have same leaky triples. For a high input that is not leaked in the source program but is leaked in the transformed program, a leaky triple can be obtained, which is not satisfying Definition 2. For such case, taint analysis will identify some new tainted program variable(s). Therefore, using taint analysis in the context of our attack model for the relative security of register allocation is sound.

Motivating Example: Consider the example of a source program segment S in Figure 1a. Program S takes two low-security (L)-inputs m and n and outputs p. The function $get\_key()$ returns sensitive (i.e., high-security (H)-input) information. Live variable analysis [8] of S tells that at least three registers are required to map all the variables of S. Assume that the machine has three free registers. The transformed program after RA (T_1) is shown in Figure 1b. If the machine does not have three free registers, it spills one variable of S. Assume that the variable p has been spilled. Then, the source program segment after inserting the spill code is shown in Figure 1c. The transformed program T_2 after RA with two available registers and spilling variable p is shown in Figure 1d.

The variable p in S and the corresponding mapped register $r1$ in T_1 are not leaking sensitive information since both of them have been redefined. Therefore, T_1 would be as secure as S. However, the memory location pm in T_2 contains the sensitive value of the variable p; thus, pm is tainted. Whereas p has been redefined in S. There is no corresponding leak in S for the tainted memory location pm. Thus, T_2 is not relatively secure as compared to S. It shows RA with spilling leads to information leakage.

A. Security Analysis

We analyze the security issues in RA with and without spilling. There are two scenarios of RA; *Scenario 1 (RA without spilling):* All variables are mapped to registers only, and no memory location is used. *Scenario 2 (RA with spilling):* Some variables are spilled to memories.

Let us denote *the source program before RA as S, the transformed program in scenario 1 as T_1, and the transformed program in scenario 2 as T_2*. Our objectives are to find *(i) Is T_1 as secure as S?, and (ii) Is T_2 as secure as S?*

Scenario 1: During RA, the variables are mapped to registers. Therefore, T_1 is exactly the same program as S, except the variables of S are renamed with corresponding registers in T_1. Let (a, b, c) be a leaky triple for T_1. Thus, T_1 is leaky only if for inputs ($I_H = a$, $I_L = c$) and ($I_H = b$, $I_L = c$): the value of one of the low-security variables (i.e., a register) differs at their final states. The sequence of output values must be the same for both the inputs as S and T_1 are functionally equivalent.

Let us assume that the values of register r differ at the final states for the execution of T_1 with ($I_H = a$, $I_L = c$) and ($I_H = b$, $I_L = c$). If we apply taint analysis on T_1, we can show that the taint value of input I_H is actually propagating to r in T_1. It indicates that there must be a control and/or data flow path from I_H to r through which the taint value of I_H propagates to r. Let us consider that the variable v of S is mapped to register r at the final states. Since the control and data flow of S is the same as that of T_1, there must be a control or data flow path from I_H to v in S, and the taint value of H must be propagated to v in S through that path. Therefore, the values of variable v differ at the final states for the execution of S with (($I_H = a$, $I_L = c$) and ($I_H = b$, $I_L = c$). Hence, (a, b, c) is also a leaky triple for S. *Thus, any leak in T_1 will have a corresponding leak in S. So, T_1 is as secure as S.*

In the case of splitting, a variable is mapped to more than one register. Assume in Figure 1a, the first assignment of variable p is mapped to register $r1$, and the second assignment is mapped to register $r2$. Potentially, $r1$ can leak p if it continues to hold the value of p till the end of the program execution. However, it would not lead to any information

leakage as we assume that the attacker has no access to direct registers.

Scenario 2: There may be information leakage through memory with spilling. Assume that in S, a tainted variable x mapped to register r has been spilled to memory x_m in the assembly. Assume x has been redefined later and becomes untainted in S. RA does not guarantee that x_m will be redefined if x has been redefined in S. Therefore, the memory location x_m has the tainted value of x for the rest of the program's execution. So, x_m leaks information in T_2. *Thus, any leak in T_2 does not induce a corresponding leak in S. So, T_2 is not relatively secure to S.*

III. Securing Register Allocation

There are four register allocators (RA) in LLVM [9], namely, *basic*, *fast*, *greedy,* and *PBQP*. In this work, we show how to secure the Greedy Register Allocation (GRA) of LLVM. We choose GRA since its the default one for optimized code in LLVM. We explain the approach of GRA in brief.

Compiler performs live interval analysis of all the program variables for allocating them to registers. The GRA finds the spill weights of all live intervals based on heuristics such as the number of uses, conflicts, etc. A priority queue is constructed based on *allocation priorities* of live intervals of the program variables. GRA marks all the variables as not to be split initially. The following steps are performed for each live interval in the priority queue until all the live intervals are allocated to registers. In each iteration, the live interval with the highest allocation priority (say x) is de-queued and assigned a register if the machine has a free register. In case of unavailability of a free register and x is not marked to be split, GRA checks if eviction of an already allocated register (say r) is beneficial. If yes, it evicts a cheaper interfering interval (say y) from r based on low spill weight and assigns the current interval (x) to the register r. The evicted interval y again en-queued with the same allocation priority. Otherwise, x is marked to be split and en-queued again with lower allocation priority. When there is no free register and x is marked to be split, splitting is performed on x if it is beneficial. Otherwise, x will be spilled to memory. Splitting divides the live interval x into smaller ones and creates new live ranges. The spill weight is calculated for all new intervals, which are to be en-queued into the priority queue based on allocation priorities. Control goes back to consider the next interval in the queue. When splitting is also not beneficial, it goes for spilling. Spilling stores the value into memory. It also creates new live ranges after inserting spill code. The spill weight is calculated for new intervals, and new live intervals are en-queued into the priority queue based on the allocation priorities. The above steps are shown in Figure 2 where the dotted box demonstrates our proposed spilling approach in GRA.

To make spilling secure, our basic intuition is flushing out the secure information from the memory location used for spilling by storing zero in the earliest possible time, i.e., immediately after the last use of each spilled memory location to reduce the lifetime. In fact, any garbage value can be used to flush out the sensitive content of the memory location. Let the live interval x is to be spilled. The GRA inserts

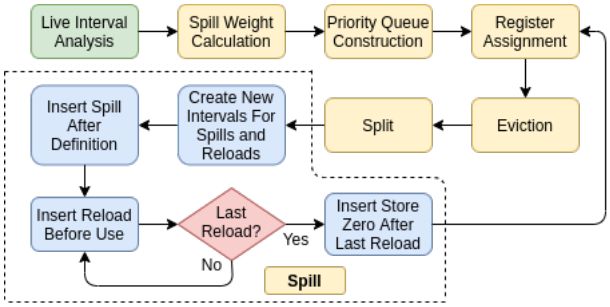

Figure 2: Secure Greedy Register Allocation in LLVM

spill instruction after definition and *reload* instruction before use of x. Consequently, the live ranges of new intervals of x due to spilling are created, and the old live interval does not exist anymore. Whenever GRA reloads the content from memory location (m) into a register, we check whether it is the last reload from the memory location (i.e., last use of m). We find the last reload by live interval analysis. If it is the last reload from m, we store zero into m just after the last reload. This process continues until we store zero after the last reload of each spilled memory location in GRA. It may be noted that a variable may be defined once and used multiple times. Inserting zero after each reload would result in incorrect functionality. The spill locations within the stack frame of a single function are heavily reused. Moreover, the entire stack frame space tends to be reused between different functions while a program executes. Therefore, actual information left in spill locations depends on the unique memory locations used for spilling. Since we insert store zero instruction only once for each spilled memory location, our approach adds minimum spill zero instructions. We also clear the sensitive memory content immediately after last use. Thus, our proposed approach is the best possible solution to make spilling secure.

IV. Experimental Results

We have used an Intel Xeon(R) CPU E5-2620 v4 2.10GHz, 64GB of RAM, running Ubuntu 18.04.3 LTS in our experiments. We ran a wide variety of LLVM test suites using different RAs in LLVM 10.0.1. Each benchmark has been compiled to generate the LLVM IR using Clang. After a few optimizations, the llc tool of LLVM generates the assembly code with the specified RA from the LLVM IR. The number of spills has been analyzed from the generated assembly, and all the performance parameters have been analyzed from the report generated by llvm-mca. We have run our proposed approach for around thirty benchmarks but presented results for ten arbitrarily chosen benchmarks in Table I and Table II due to limited space. Note that we have observed a similar trend from other benchmarks as well.

A. Results with Register Allocations in LLVM

Table I shows the number of total spills (T_s) versus leaks (L) generated by different benchmarks for different RAs in LLVM. The number of leaks (L) for all the existing RAs of LLVM is the unique memory locations used for spilling. We have assumed that all inputs are high-security such that any spill is a potential leak. The second to ninth, eleventh, and twelfth columns provide spills and leaks due to *Basic*, *Fast*, *PBQP*,

Table I: Spills (T_s) and Leaks (L) in Basic, Fast, and PBQP RA, and T_s, L, and Registers (R_n) in SGRA Vs GRA

Benchmark	Basic		Fast		PBQP		GRA			SGRA		
	T_s	L	T_s	L	T_s	L	T_s	L	R_n	T_s	L	R_n
linpack-pc	91	22	313	74	93	21	63	17	163	80	0	163
almabench	115	15	96	59	115	15	47	15	124	62	0	124
n-body	28	8	58	17	28	8	10	7	139	17	0	139
partialsums	30	12	35	25	30	12	22	12	103	34	0	103
fftbench	24	8	226	44	23	9	16	7	109	23	0	109
misr	23	10	80	30	19	10	12	10	168	22	0	168
lpbench	8	6	108	22	9	6	9	7	150	16	0	150
Queens	8	8	22	14	8	8	8	8	114	16	0	114
recursive	10	5	34	11	10	5	7	5	102	12	0	102
chomp	4	3	116	20	4	3	4	3	118	7	0	118

Greedy RA (GRA), and our proposed secure GRA (SGRA), respectively. It may be noted that actual leaks (L) are quite low compared to the total spills (T_s) since the memory space is heavily reused during spilling. Our method inserts 'L' store zero operations in SGRA. The results show that *fast* generates the maximum leaks among existing RA algorithms.

The results in Table I show that register allocation without spilling is impossible for practical test cases. Therefore, register allocation in LLVM is leaky. The leaks for SGRA are zero in all cases because we flushed the sensitive data from spilled memory locations. The number of spills for SGRA increases from GRA because of additional spill zero instructions. We analyzed that there would be no change in the number of registers required at any time for both GRA and SGRA, as our added instructions do not need any register to perform the memory operations.

Table II: Performance overhead in SGRA versus GRA wrt Instructions, Cycles, Block RThroughput, and Resource Pressure

Benchmark	Instrs (K)		Cycles (K)		BlockRT		ResPre (%)	
	GRA	SGRA	GRA	SGRA	GRA	SGRA	GRA	SGRA
linpack-pc	162.4	164.6	101.7	103.7	568.3	573.8	7.27	7.52
almabench	51.8	53.4	44	44.4	202.8	206.8	5.26	7.42
n-body	27.2	27.9	21.6	21.8	96.8	98.5	8.33	7.37
partialsums	20.4	21.7	17.3	17.9	80.8	84.0	8.14	10.5
fftbench	129.8	132.6	137.9	140.4	593.5	607.0	1.67	2.49
misr	41.7	42.7	32.7	33.4	149.0	151.5	2.29	4.15
lpbench	47.2	47.9	35.6	35.8	168.0	169.8	3.10	2.52
Queens	16.2	17.5	14.7	15.1	60.8	64.0	0.77	4.02
recursive	18.0	18.5	18.7	19.1	81.0	82.3	1.87	2.61
chomp	77.2	79.3	79.6	80.4	332.5	338.3	1.63	1.37
Avg.	**1.42%**		**0.82%**		**0.04%**		**0.97%**	

B. Performance Overhead

Table II presents the performance overhead of SGRA over GRA concerning the total number of instructions (Instr in thousands (K)), the total required execution cycles (Cycles in thousands (K)), the block RThroughput (BlockRT), and the resource pressure (ResPre in percentage) for the target assemblies generated from the benchmarks. We have run each benchmark for X86-64 architecture in llvm-mca for 100 iterations and recorded the results for these performance parameters. The average overhead of all the ten benchmarks for each parameter is shown in the last row of Table II. Block RThroughput is the reciprocal of the block throughput. Block throughput is computed as the maximum number of blocks that can execute per simulated clock cycle. Resource pressure is the maximum number of instructions executed in parallel in each cycle. Total instructions, cycles, and block RThroughput are increased marginally for all cases in SGRA due to the added spills by our approach. The resource pressure varies for benchmarks as the instruction scheduling getting changed in SGRA. The results suggest there is no significant impact of our proposed SGRA on overall performance.

V. Summary and Future Directions

In this paper, we investigated the inherent vulnerability of register allocation in the presence of spilling. Our experimental studies revealed that all LLVM RAs create information leaks due to spilling. This work shows one simple way of making register spilling secure by flushing the memory content after last use. Our attack model is also practically relevant because: (i) the attacker needs to first find few cycles where secure data processes (say an encryption routine) which is a small fraction of the actual program of millions of cycles. (ii) Once the secure routine is identified, the attacker has to identify the exact timing between register spill and store zero which is extremely hard even with the state-of-the-art side-channel analysis. (iii) Dumping of memory content randomly during execution is also not trivial and involves many cycles since the I/O speed is significantly slower than execution speed. The existing approach [4] also assumes a similar attack model.

Our work opens up many questions on the security of the register allocation process, as discussed here. (i) If the attacker has access to the registers as well, splitting could also be leaky. (ii) The security issues of RA need to be analyzed properly under other attack models as well. (iii) Our implementation can be enhanced to store zero only in the tainted spilled memory locations. (iv) Another approach of making spilling secure could be by avoiding the spilling of tainted variables to memory. (v) If spilling performs into cache instead of memory, we need to mitigate the leaks in the cache. (vi) A generic translation validation approach is needed to ensure the relative security of the compiler optimizations. These open problems need to be explored further for a better analysis of secure compilation.

References

[1] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[2] C. Deng and K. S. Namjoshi, "Securing a compiler transformation," in *Static Analysis*, X. Rival, Ed., 2016, pp. 170–188.

[3] ——, "Securing the SSA transform," in *Static Analysis'17*, F. Ranzato, Ed., 2017, pp. 88–105.

[4] F. Besson, A. Dang, and T. Jensen, "Securing compilation against memory probing," in *PLAS '18*, 2018, pp. 29–40.

[5] ——, "Information-flow preservation in compiler optimisations," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 230–242.

[6] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.

[7] D. Ceara, L. Mounier, and M. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in *ICST'10*, April 2010, pp. 371–380.

[8] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. USA: CRC Press, Inc., 2009.

[9] LLVM Register Allocation, https://llvm.org/docs/CodeGenerator.html#register-allocation, Accessed February 23 2021.