Efficient and Safe Control Flow Recovery Using a Restricted Intermediate Language

Tobias Pfeffer*, Paula Herber*, Lucas Druschke†, Sabine Glesner*
Technische Universität Berlin, Berlin, Germany
*name.lastname@tu-berlin.de
†l.druschke@campus.tu-berlin.de

***Abstract*—Approaches for the automatic analysis of security policies on source code level cannot trivially be applied to *binaries*. This is due to the lacking high-level semantics of low-level object code, and the fundamental problem that control-flow recovery from binaries is difficult. We present a novel approach to recover the control-flow of binaries that is both safe and efficient. The key idea of our approach is to use the information contained in security mechanisms to approximate the targets of computed branches. To achieve this, we first define a *restricted control transition intermediate language* (RCTIL), which restricts the number of possible targets for each branch to a finite number of given targets. Based on this intermediate language, we demonstrate how a safe model of the control flow can be recovered without data-flow analyses. Our evaluation shows that that makes our solution more efficient than existing solutions.**

I. Introduction

The increasing integration of collaborating devices into our everyday lives comes with a threat to the security of our data. Potentially all data provided to a device could be leaked to an unauthorized third party. Trust in the appropriate handling of sensitive user data could be increased if a specification of the behavior of the software running on the devices was given. Unfortunately, commercial devices are usually not shipped with high-level specifications such as source-code. As a result, analyses of their compliance with security policies must be based on the machine code.

Since auditing the machine code requires skills not available to the average user, an automatic approach is to be favored. Yet, the great number of potential paths in a typical binary leads to a path explosion in dynamic analyses, such as symbolic execution. One solution to this problem lies in the creation of abstract models first, which can be used to rule out portions of the code to focus on critical paths. For example, if a security policy constitutes that a specific function must always be called before another function is executed (e.g. for access control), it can be validated without executing expensive data-flow analysis, solely based on the inter-procedural control flow. Unfortunately, machine code lacks high-level semantics such as functions that provide a readily accessible level of abstraction. Hence, abstractions such as control-flow models must be recovered first [1]. Yet, due to the presence of computed branching instructions that depend on the memory state, control flow and data flow are naturally intertwined in binary programs. Existing approaches for the reconstruction of the control-flow graphs aim to resolve data flow and control flow at the same time. This leads to expensive analyses that scale poorly and thus complicate validation even of simple policies like the one mentioned above. To overcome this problem, we propose a data-flow free approach to control-flow recovery. The key idea is to reuse information embedded in the protection mechanisms in modern binaries. As data-dependent control-flow transfers often make compiled binaries vulnerable to arbitrary control redirection, it is also a concern for defensive security [2]. This has led to the introduction of various countermeasures, including control-flow integrity (CFI). Under CFI defense, binaries are enriched with mechanisms that restrict the possible targets of indirect jumps to a set of potential targets, collected at compile-time. For our novel control-flow recovery algorithm, we use this information to resolve indirect jump targets. The resulting control-flow model is necessarily sound, as CFI mechanisms terminate the execution if the control flow diverges from the embedded targets. As the resolution works without nontrivial data-flow analyses, the approach scales better than data-flow based solutions.

The paper is structured as follows. In the next section, we give a brief introduction into preliminaries, summarizing related work in Section III. In Section IV we introduce our approach to the recovery of control-flow models from protected binaries. As a first step, we define our *restricted intermediate language* in Section IV-A and show how machine code can be expressed in it. Then, we describe our control-flow recovery algorithm in Section IV-C and evaluate it in Section V. We show the resulting control-flow model and demonstrate the scalability of our approach. Finally, in Section VI, we conclude and give directions for future work.

II. Preliminaries

In this section, we briefly introduce the preliminaries, namely the simple intermediate language (SimpIL) [3] and control-flow integrity as a protection mechanism.

A. Simple Intermediate Language

To define the ideas expressed in this paper, we use a simplified version of the simple intermediate language (SimpIL) introduced in [3]. It is general enough to express the control-flow semantics of typical machine code, yet abstract enough to hide technical details. A program in SimpIL is given by

©2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

$$\textsc{Store}\ \frac{\mu \vdash e_1 \Downarrow v_1 \qquad \mu \vdash e_2 \Downarrow v_2 \qquad \mu' = \mu[v_1 \leftarrow v_2] \qquad \iota = \Sigma[pc+1]}{\Sigma, \mu, pc, \text{store}(e_1, e_2) \rightsquigarrow \Sigma, \mu', pc+1, \iota} \qquad \textsc{Goto}\ \frac{\mu \vdash e \Downarrow v \qquad \iota = \Sigma[v]}{\Sigma, \mu, pc, \text{goto } e \rightsquigarrow \Sigma, \mu, v, \iota}$$

$$\textsc{TCond}\ \frac{\mu \vdash e \Downarrow 1 \qquad \iota = \Sigma[v]}{\Sigma, \mu, pc, \text{if } e \text{ then goto } v \rightsquigarrow \Sigma, \mu, v, \iota} \qquad \textsc{FCond}\ \frac{\mu \vdash e \Downarrow 0 \qquad \iota = \Sigma[pc+1]}{\Sigma, \mu, pc, \text{if } e \text{ then goto } v \rightsquigarrow \Sigma, \mu, pc+1, \iota}$$

Fig. 1. Simplified SIMPIL semantics

$\Sigma : \mathbb{N} \mapsto \mathit{Instr}$, which maps the addresses in the code to specific instructions. The variable $pc \in \mathbb{N}$ models the program counter, describing the currently executed address. We assume a fixed instruction length of 1, hence the starting address of the next instruction is given by $pc+1$. This is a simplification from real architectures with varying encoding lengths. However, since these lengths are instruction-specific, the definition could easily be extended to respect values depending on the instruction. The memory state is described by $\mu : \mathbb{N} \mapsto \mathbb{N}$, mapping addresses to their respective value. Together with the currently executed instruction $\iota \in \mathit{Instr}$, a machine state is described by a tuple (Σ, μ, pc, ι). The simplified operational semantics of the SIMPIL is presented in Figure 1. Expressions are modeled as e, which is an arbitrary complex combination of the operators usually found in machine languages. The expression e may also use the values stored at memory locations, hence its evaluation requires a specific memory state. $\mu \vdash e \Downarrow v$ denotes that the expression e is evaluated to the value v, assuming the data state μ. The STORE instruction models the evaluation of a target expression e_1 and a value expression e_2 and stores the result v_2 in the memory μ at location v_1. After execution of this instruction, the program counter *falls through* to the succeeding instruction, given by $\Sigma[pc+1]$. Conditional branch instructions have two possible evaluations, modeled by TCOND and FCOND. In case the condition expression evaluates to zero (False), the jump is not taken and the program counter falls through to the next instruction. In case the condition expression evaluates to one (True), the jump is taken and the execution continues at the constant target v. Finally, the GOTO instruction models an indirect branch, such as computed jumps in machine code. Its target is computed by evaluation of e and may depend on the memory state μ. This gives rise to problems both regarding the security of the control flow as well as its reconstruction, which we explain in the following sections. As proposed in [3], we omit specific high-level instructions such as calls and return here but will reintroduce them when defining our intermediate language in Section IV-A.

B. Control-Flow Integrity

While the successor of a conditional branch instruction depends on the outcome of the evaluation (and thereby on the memory state μ), its successors are limited to v and the next instruction, $pc+1$. In contrast, the successor of the GOTO instruction may depend on the memory state μ directly, meaning it could be any value from $\mathbb{N}$. As a consequence, the control flow can be hijacked after the memory state has been corrupted [2]. Consider a STORE instruction where the attacker has control over the values v_1 and v_2, constituting a *write-what-where* attack. She may alter μ in such a way that a succeeding GOTO instruction evaluates to an address the attacker wishes to execute. Various countermeasures have been proposed that aim to protect the control flow, including control-flow integrity (CFI). Under CFI protection, potentially insecure control transitions are augmented with target checking code [4]. The target checking code dictates that execution must follow a path of an inter-procedural control-flow graph, determined ahead of time. Figure 2 presents an example of CFI protection in the compiled code. Based on information inferred from the source code, individual labels are added for all call sites and indirect branch targets. For example, the indirect call in the `sort` function can only transfer control to targets with a matching tag (i.e., 17, which corresponds to `lt` and `gt`). Vice versa, the `gt` and `lt` functions can only return to call sites with a matching tag (i.e., 23, within the `sort` function). This means that control flow can only be redirected to the predefined addresses, a finite subset of $\mathbb{N}$. The original authors note that, because CFI concerns only static control flow, it cannot ensure that a function call returns to the call site most recently used for invoking the function. As an enhancement, they propose to complement CFI with a dedicated call stack, usually referred to as shadow stack.

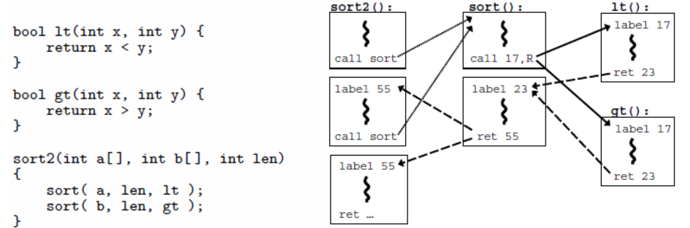

Fig. 2. Control-flow integrity example [4]

III. RELATED WORK

For the same reasons that GOTO is vulnerable to control-flow hijacking, it also thwarts precise control-flow analysis. As its successor depends on the actual memory state given by μ, a recovery analysis must resolve all reachable memory states for each GOTO instruction. Therefore, control-flow analysis depends on the results of a data-flow analysis step, which in turn depends on the results of the control-flow analysis.

```
1   int a(void) {return 5;}
2   int b(void) {return 6;}
3
4   int main(int argc, char **argv) {
5       int (*procs[])(void) = {a,b};
6       int count = 0;
7       for(int i = 0; i < 100; i++) {
8           if(argv[1][i] == '0') break;
9           if(argv[1][i] == 'Z') count ^= 1;
10      }
11      return procs[count]();
12  }
```

Fig. 3. Example code

This cyclic dependency has been coined the "chicken and egg" problem of control-flow recovery for machine code.

Approaches that aim to perform these analyses sequentially, such as the pipes-and-filter-based decompilation introduced by Cifuentes and Gough in 1995 [5], or heuristics-based approches [6], [7], are incapable of precisely resolving indirect jump targets [8], [9]. Instead, modern approaches interweave data-flow and control-flow analysis. In their paper on Bitblaze [10], Song et al. connect unresolvable GOTOs to a special node, that allows Bitblaze to recognize unsound results in later stages. In CodeSurfer/x86 [11], Balakrishnan et al. use their value set analysis (VSA) approach first described in [12]. In VSA, the machine code is interpreted in an abstract domain that overapproximates all reachable memory states. Kinder et al. proposed a similar approach with their work on Jakstab [13]. Based on the work by Flexeder et al. on inter-procedural control-flow reconstruction [9], Miahila devised a hierarchical structure of cofibered domains to increase the precision of VSA-based control-flow and data-flow anlyses [14]. Additional binary analysis frameworks exist, such as Binoca [15], [16] or BAP [17], [18]. Note that, as Andriesse et al. present in their in-depth study of disassembly techniques, perfect control-flow recovery is not generally achieved by any of the existing tools [1]. Lately, approaches based on symbolic execution have gained attention [19], [20], but they are mostly used to prove the presence of vulnerabilities and do not aim for full coverage of the control flow. All data-flow based approaches mentioned above typically scale poorly for larger binaries.

IV. RESTRICTED CONTROL-FLOW RECOVERY

The goal of our work is to reconstruct a model of the control flow from a given binary. To enable security analysis, the model must be sound, meaning that all potential control flow present in the binary is also included in the control-flow model. While a control-flow model that connects each instruction with every other instruction would trivially be sound, it would also be of little use to further analyses. Hence the quality of the model also depends on the precision of the control flow, meaning that it should contain as few infeasible transitions as possible. Unfortunately, precise resolution of indirect jump targets requires computation of all possible reachable states, which in general is infeasible.

To illustrate this, consider the example code in Figure 3, adapted from [20]. Since the indirect call in Line 11 depends on the value of `count`, the resolution of the potential targets of the call require analysis of the possible values of `count`. However, since the values stored in `argv` depend on user input, the preceding loop creates an exponential number of paths during analysis. This leads to the path explosion problem, rendering precise analyses infeasible.

To overcome this problem, we propose to use information from control-flow integrity (CFI) protections to aid indirect target resolution. With CFI protection enabled, modern compilers perform a coarse-grained version of program-wide analysis and include the results in the machine code [21]–[23]. Reconsider the example in Figure 3. During compile-time, the symbols `a` and `b`, as well as their addresses can easily be resolved. Additionally, the boundaries of `procs` are known, making it possible for the compiler to determine a conservative approximation of the potential targets of the indirect call in Line 11. This information is used to augment the code with CFI checking code and is thus contained in the final output. SIMPIL does not offer a nice way to represent the CFI information extracted from binaries, which is why we define a slightly altered intermediate language in the next section. Based on this *restricted* intermediate language, we then show how the control flow of CFI-protected binaries can be safely approximated without applying data-flow analyses.

A. Restricted Control Transition Intermediate Language

As explained in Section II-A, most of the SIMPIL instructions have finitely many successors and their control-flow behavior can be determined without consulting the memory state μ. However, the indirect branch instruction, GOTO, is problematic for control-flow recovery. The concrete successor of these indirect branch instruction relies on the state that μ may assume during execution. As described in Section III, a precise static analysis must consider all reachable memory configurations, which is infeasible in the general case.

To overcome this problem, we replace the GOTO instruction with the restricted RGOTO in our RCTIL. The key idea behind the RGOTO instruction is to restrict its possible targets to a predefined set, denoted by T. The set is added to the machine state and referenced by the RGOTO instruction after computation of the target. Execution aborts if the evaluation of the target expression e leads to a value that is not within the target table. Thus, it is impossible to redirect the control flow to a location that is not contained in T. Therefore, T acts as a safe overapproximation of the possible successors of RGOTO.

While this instruction is still general enough to represent inter-procedural control flow, combining both return and indirect call targets in the target table T would make it unnecessarily large. The target of return instructions is usually taken from the top of the stack, after being stored there by the most recent call instruction. Since the stack is part of the memory, it can be subsumed by the data state μ. However, this allows

$$\text{RGOTO}\frac{\mu \vdash e \Downarrow v \quad v \in T \quad \iota = \Sigma[v]}{\lambda, \Sigma, \mu, T, pc, \text{goto } e \rightsquigarrow \lambda, \Sigma, \mu, T, v, \iota}$$

$$\text{RCALL}\frac{\mu \vdash e \Downarrow v \quad v \in T \quad \iota = \Sigma[v]}{\lambda, \Sigma, \mu, T, pc, \text{call } e \rightsquigarrow (pc+1) :: \lambda, \Sigma, \mu, T, v, \iota} \qquad \text{RET}\frac{\iota = \Sigma[pc']}{pc' :: \lambda, \Sigma, \mu, T, pc, \text{return} \rightsquigarrow \lambda, \Sigma, \mu, T, pc', \iota}$$

Fig. 4. Changes in the RCTIL semantics

the STORE instruction to manipulate return targets, introducing another vulnerability and thwarting analysis. Hence, we follow the argumentation from [3] and introduce a specific stack context, λ, as well as specific instructions for inter-procedural control flow. The RCALL instruction behaves equivalent to the RGOTO instruction, but pushes the succeeding address on the stack. Conversely, the RET instruction pops the address on top of the stack and transfers control to this address. Note that λ is distinct from μ and cannot be altered by the STORE instruction. It can only be manipulated by execution of RCALL and RET instructions.

Together, our changes ensure that control flow of RCTIL instructions can safely be approximated without computation of μ. Hence, if a program can be expressed using RCTIL, we achieve a data-flow independent overapproximation of the control flow, allowing for efficient recovery. In the next section, we show how CFI-protections implemented in Visual Studio and Clang support our assumptions made for RCTIL.

B. RCTIL Implementation

To make use of our intermediate language, programs given in machine code form must be translated into it. However, since RCTIL is stripped of the potentially insecure indirect jump and call instructions, a direct mapping is not possible. Additionally, the separation of λ and μ has to already be present in the semantics of the original machine language. Fortunately, with the adoption of control-flow integrity and shadow-stack protections, these obstacles are solvable.

Both Visual Studio and Clang integrate a form of CFI in their current version. Microsoft's Control Flow Guard works by adding lightweight security checks to the compiled code [22]. Before executing the indirect jump, a direct call to a checking routine is inserted, as depicted in Figure 5. The `_guard_check_icall` routine checks if the target of the following instruction is part of the Guard CF Function Table (GCFFT), which is stored in the binary and contains all valid indirect call targets [23]. Only if this check is passed, the routine returns and executes the indirect call in the next step. Combined, both calls are jointly represented by a RCALL instruction, where the GCFFT is used as the target table T. Similarly, Clang embeds a trampoline table of potential targets in the code, as depicted in Figure 6. Before an indirect call is followed, the value of the target register is compared to this table to ensure that a valid target is addressed. Yet, instead of calling a specific routine, Clang inserts the validation logic preceding the indirect call. Hence,

```
call    _guard_check_icall
call    rcx
```

Fig. 5. MS Control Flow Guard augmentation

```
0x400c58: jmp $0x400210 <a.cfi>
0x400c60: jmp $0x400220 <b.cfi>
```

Fig. 6. Trampoline table inserted by Clang

to translate the indirect call, the trampoline table functions as the target table T for a single RCALL instruction that covers the semantics of the code. Finally, the separation between μ and λ must be enforced. To achieve this, we exploit the shadow stack protection, introduced to Clang as part of the code-pointer integrity policy [24]. This shadow stack ensures that the targets of return instructions are managed on a separate stack. Note that the translation into RCTIL is purely based on syntax analyses and thus does not require expensive data-flow analysis. Our RCTIL may also be implemented in hardware. For this, Intel has proposed compatible changes to their instruction set with the introduction of the Control-flow Enforcement Technology [25]. Furthermore, control-flow tagging support is also planned for the RISC-V instruction set. Thus, we conclude that various modern hardware architectures and compiler frameworks support translation into RCTIL.

Based on our RCTIL, we outline our control-flow recovery approach in the next section. We then demonstrate the effectiveness and efficiency of our solution in Section V and conclude in Section VI.

C. Control-Flow Recovery

Our proposed control-flow recovery is executed in two consecutive steps. First, we use a recursive traversal algorithm, starting from the entry point of the program. To simulate the advancement of the program counter pc, an instruction is disassembled and executed according to the RCTIL semantics. Due to the design of our RCTIL, the algorithm does not need to track the memory state μ. Hence STORE instructions can be skipped and the considered machine state can be reduced to the tuple (Σ, T, pc, ι). Note that this first step also ignores the call stack context described by λ. As Σ and T do not change during the execution, visiting an address twice leads to the same state and thus must not be traversed again. Due to the lack of μ, we overapproximate the successors of some

instructions. Conditional branch instructions are connected to both their possible successors. Whenever indirect branch instructions are reached, we resolve their respective targets using the information stored in the target tables. This overapproximation is sound, as no additional control flow is allowed during execution of the binary. The exclusion of λ from the machine state implies that backwards edges introduced by RET instructions cannot be resolved during this stage. Instead, we advance the algorithm by conservatively assuming that every call instruction is eventually followed by the next instruction. To represent this fact, we use fake returns as a special edge type, similar to existing CFR frameworks [20]. The resulting intermediate control-flow model contains all intra-procedural control flow and inter-procedural forward edges, but lacks successors to RET instructions. These are resolved in the second step. During the second recovery step, we need to consider the call stack context modeled by λ. However, since the RET instruction only uses the topmost entry of λ, we do not need to reconstruct it completely. Instead, we traverse the intermediate CFG in reverse order for each RET instruction. Once we approach a call instruction, we can be certain that this instruction most recently altered the call stack on this path and that its succeeding instruction is the intended return site. Naturally a RET instruction may be reachable from various calls, leading to various return sites. Vice versa, a call may be followed by multiple returns. Consequently, subsequent analyses may continue at the wrong destination, leading to incorrect function boundaries. To counter this, we enrich our inter-procedural edges with tags, identifying matching call and return sites. To enable later identification of function starts, edges that represent inter-procedural control flow are marked as such. The resulting control-flow graph contains both intra- and inter-procedural control flow, which can be used to determine function boundaries. This is done by collecting all nodes following the intra-procedural control flow from each call site until RET instructions are encountered. This process is a simple graph traversal algorithm that requires no expensive analysis but enables validation of function-level security policies. We evaluate the effectiveness and efficiency of our approach in the next section.

V. Evaluation

We evaluate our approach based on the example introduced in Figure 3 and use all binaries from the `coreutils` package to benchmark its performance. For this purpose, we implemented our analysis as an extension to `angr` [20], a recent binary analysis framework based on symbolic execution. The framework provides a control-flow recovery algorithm that aims to resolve indirect jump targets through data-flow analysis. Our analysis is based on the provided algorithm, but replaces all data-flow analysis steps with our RCTIL-based target resolution. As explained above, the quality of the recovered control-flow model depends on its precision. Unfortunately, precision is generally hard to measure as it would require a perfect model for comparison. Instead, we evaluate the quality of our solution using the example given in Figure 3.

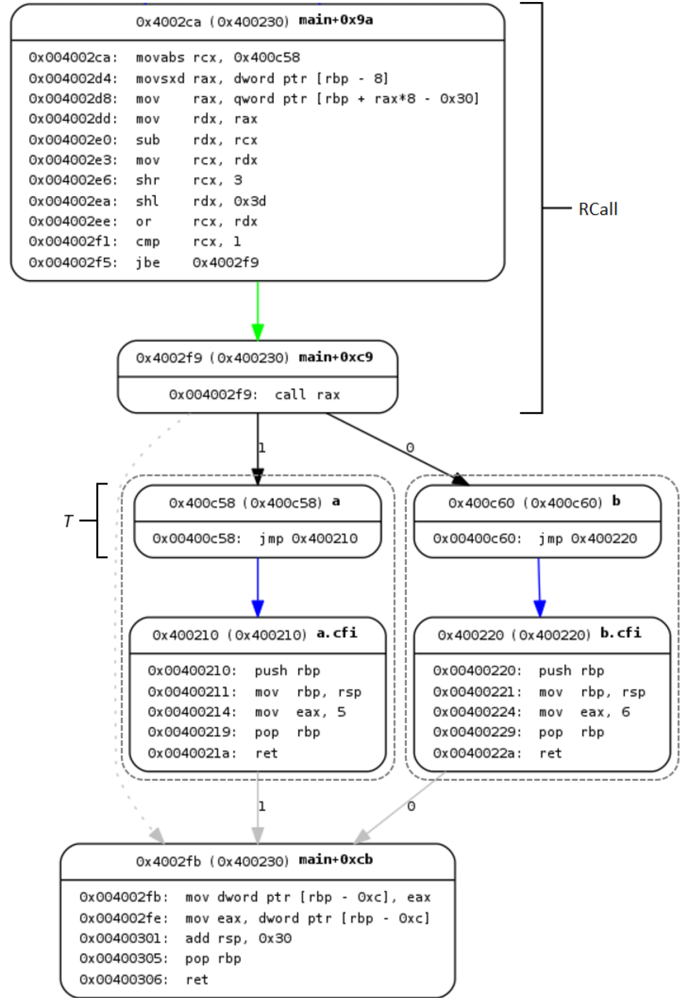

Fig. 7. Resolved indirect control flow from the example

Most of the control flow can be recovered straightforwardly without using data-flow analysis. Since loops and branches are implemented using direct jump instructions, they can be resolved without additional information. In contrast to this, the indirect call can not trivially be resolved, as its target depends on the values that `rax` may assume during execution. Using `angr`'s default CFR algorithm, these indirect jump targets are not resolved, leading to an unsound result. In contrast, Figure 7 shows the resolved indirect control flow using our solution. The first node contains the CFI checking code inserted by Clang before the indirect call in the second node. The successors of the call are the trampoline jumps that together constitute the target table T (here $T = \{0x400c58, 0x400c60\}$). Both targets have been identified correctly, leading to a sound (and in this case precise) control-flow model. Additionally, the inter-procedural edges are tagged with equivalent numbers (0 and 1, resp.) and the function boundaries of both `a` and `b` are identified. Note that, due to our definition of intra-procedural control flow, the trampoline jumps introduced by Clang are considered part of the function.

To demonstrate the efficiency of our approach, we compared

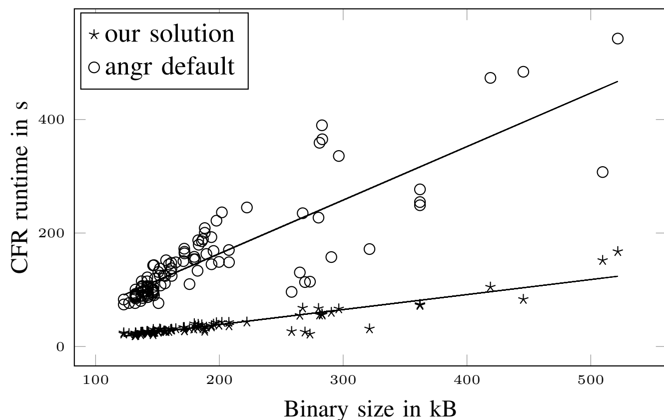

Fig. 8. Runtime comparison

the run time of our solution to the run time of the default algorithm, using the `coreutils` package as a benchmark. The results are shown in Figure 8. On average, our solution is over four times faster. More importantly, the acceleration increases with growing complexity of the samples, demonstrating the efficiency benefits of our data-flow free approach.

VI. Conclusion

In this paper, we have presented a novel approach for the reconstruction of control-flow graphs from CFI-protected binaries. To overcome the problem that statically resolving indirect branch instruction targets during control-flow recovery is either expensive or imprecise, we have defined the *restricted control transition intermediate language* (RCTIL). The key idea is that indirect branch instructions are restricted to a set of predefined targets. We have shown that the recent integration of control-flow integrity (CFI) and shadow stacks in major compilers support our approach. This enables us to compute a safe overapproximation of the control flow, without using expensive data-flow analysis. Our evaluation shows that our algorithm has the potential to yield precise control-flow models and scales better than a data-flow based approach. Our resulting control-flow model can be used to recover function boundaries and inter-procedural control flow, which enables lightweight validation of control-flow related security policies. Future research will show, how well our approach integrates with hardware integration of control-flow integrity protection and how well it can handle virtual function calls. Additionally, some problems regarding dynamic control-flow mechanisms that are not protected by CFI (such as `setjump/longjump`) still remain open.

References

[1] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," 2016.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[3] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[5] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[6] B. De Sutter, K. De Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," in *Proceedings of the international conference on parallel and distributed processing techniques and applications*, vol. 2, 2000, pp. 1013–1019.

[7] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *Reverse engineering, 2002. Proceedings. Ninth working conference on*. IEEE, 2002, pp. 45–54.

[8] A. Lakhotia and P. K. Singh, "Challenges in getting formal with viruses," *Virus Bulletin*, vol. 9, no. 1, pp. 14–18, 2003.

[9] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl, "Interprocedural control flow reconstruction," in *Programming Languages and Systems*. Springer, 2010, pp. 188–203.

[10] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, *BitBlaze: A New Approach to Computer Security via Binary Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89862-7_1

[11] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, pp. 23:1–23:84, #aug# 2010. [Online]. Available: http://doi.acm.org/10.1145/1749608.1749612

[12] ——, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[13] J. Kinder, "Static analysis of x86 executables," Ph.D. dissertation, Technische Universität Darmstadt, 2010.

[14] B. Mihaila, "Adaptable static analysis of executables for proving the absence of vulnerabilities," Ph.D. dissertation, München, Technische Universität München, Diss., 2015, 2015.

[15] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The bincoa framework for binary code analysis," in *Computer Aided Verification*. Springer, 2011, pp. 165–170.

[16] E. Fleury, O. Ly, G. Point, and A. Vincent, "Insight: An open binary analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 218–224.

[17] D. Brumley, I. Jager, E. J. Schwartz, and S. Whitman, "The bap handbook," 2013.

[18] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 353–368. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[19] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," *NDSS*, 2015.

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

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, #Aug# 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[22] M. D. Network, "Control flow guard," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

[23] J. Tang and T. M. T. S. Team, "Exploring control flow guard in windows 10," *Available at ht tp://blog. trendmicro. c om/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10*, 2015.

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 147–163.

[25] Intel, "Control-flow enforcement technology preview," Intel, Tech. Rep., 2016.