SYNTHCT: Towards Portable Constant-Time Code

Sushant Dinesh, Grant Garrett-Grossman, Christopher W. Fletcher
University of Illinois at Urbana-Champaign
{sdinesh2, grantlg2, cwfletch}@illinois.edu

***Abstract*—Recent attacks have demonstrated that modern microarchitectures are fraught with microarchitectural side channels. Constant-time (CT) programming is a software development methodology where programs are carefully written to avoid these channels. In a nutshell, the idea is to only pass secret data to *safe* instructions, i.e., those whose execution creates operand-independent hardware resource usage.**

Yet, current CT programming practices have significant security and performance issues. CT code is written and compiled once, but may execute on multiple different microarchitectures. Yet, what instructions are safe vs. unsafe is fundamentally a microarchitecture-specific issue. A new microarchitectural optimization (or vulnerability) may change the set of safe instructions and break CT guarantees.

In this work, we develop SYNTHCT to address the above issues. Given a specification of safe/unsafe instructions, SYNTHCT automatically synthesizes translations for all unsafe instructions in the ISA using only instructions from the safe set. The synthesized translations can be used as a part of a late-stage compiler pass to generate hardened binaries for a specific microarchitecture. This closes the security hole as the specification, and hence the safe translations, can target each microarchitecture individually. This also allows CT code to reclaim some performance, e.g., use more complex/higher-performing instructions, when they are deemed safe for a specific microarchitecture.

Using the techniques we develop in SYNTHCT, we are able to synthesize translations for a majority of the x86_64 ISA. Specifically, SYNTHCT is able to generate safe translations for 75% of the ISA using only the remaining 25% of the ISA. Interestingly, the majority of the instructions that SYNTHCT was unable to generate translations for are instructions that experts believe are safe instructions on today's x86_64 microarchitectures.

I. INTRODUCTION

Microarchitectural side-channel attacks are a major privacy threat. By observing hardware resource utilization from myriad sources—virtual memory accesses [1], [2], hardware memory accesses [3], [4], branch predictor usage [5], [6], arithmetic pipeline usage [7], [8], [9], speculative execution [10], [11] and more [12]—an attacker can learn substantial amounts of information about a victim program, such as its memory access pattern, control flow behavior, arithmetic operation operands, etc. Given the large number of hardware resources that leak different types of information, it is imperative to develop comprehensive software defenses.

Such comprehensive software-only protection against microarchitectural side channels is achieved through constant-time programming [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [7], [25], [26], [27]. In this paradigm, security critical applications are carefully written and compiled to prevent private information from conditionally influencing hardware resource usage. In a nutshell, constant-time programming ensures: (i) that the program does not have private data-dependent branches, and (ii) that the program does not have *unsafe* instructions with private data-dependent operands. Here, an instruction is *unsafe* if its execution, e.g., timing/hardware resource usage, depends on its operands. Otherwise, the instruction is *safe*.

However, the current constant-time development methodology has a significant security and performance problem. The fundamental issue is that, much like other software, constant-time code is written and compiled once, and reused for a long time across many different microarchitectures. Yet, what instructions are safe vs. unsafe is fundamentally microarchitecture-specific. If an instruction presumed to be safe at compile time is not safe on some current/future microarchitecture, we get a potential security vulnerability. This is especially worrisome now, as Moore's law's slowing incentivizes microarchitects to adopt more exotic optimizations to get performance, rendering potentially many instructions unsafe [28]. On the flip side, if an instruction presumed to be unsafe at compile time is safe on some current/future microarchitecture, we incur unnecessary performance overhead as being able to utilize more instructions tends to reduce overhead.

Prior work on constant-time programming is complementary and has not addressed this issue. Such work either (i) generates *mostly constant-time code* from a higher-level language, e.g., a compiler-based DSL such as FaCT [29], (ii) or hand-writes constant-time code for specific critical operations [7]. (By *mostly constant-time*, we mean a program that treats control-flow and memory instructions as unsafe.) Both of the above assume that which instructions are safe does not change across microarchitectures.

In this work, we develop SYNTHCT to address the above security and performance issues. SYNTHCT enables microarchitects to specify what instructions are safe on a per-microarchitecture basis, called the *safe set*, and uses program synthesis techniques to *automatically* generate translations for every unsafe instruction in a target program to instructions in the safe set. Specifically, SYNTHCT takes as input: (i) a *mostly constant-time target program* in binary form, (ii) the formal instruction semantics for the target program's ISA, and (iii) a safe-set specification for the microarchitecture the target program is to be run on, and automatically replaces any unsafe instructions in the target program with semantically equivalent translations made up of safe instructions. SYNTHCT

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24215
www.ndss-symposium.org

can also improve performance by choosing the best possible set of safe instructions for a given microarchitecure while avoiding applying translations for safe instructions in the target program.

Importantly, SYNTHCT relies on each microarchitecture publishing which instructions are safe vs. unsafe. While this is not common today, there has been both industry [30] and academic [31], [32] traction in publishing similar specifications, suggesting they may become common practice in the future.

SYNTHCT builds on state-of-the-art program synthesis techniques and formulates the problem of synthesizing unsafe instructions as a Counter-Example Guided Inductive Synthesis (CEGIS) problem. Yet, scaling CEGIS techniques to modern ISAs such as x86_64, as needed by SYNTHCT, is non-trivial due to the extremely large synthesis search space.

In SYNTHCT, we design and implement multiple techniques to restrict the synthesis search space and scale CEGIS to modern ISAs. Specifically, the search space grows primarily as a function of two factors: (i) the number of available instructions in the ISA (e.g., 451 non-floating point/vector instructions, or over 1000 instruction variants total, in x86_64) and (ii) the maximum length of allowed synthesized programs. To address (i), we develop *component selection*, a procedure that prunes the ISA down to a manageable subset of instructions which are most likely to result in successful translations for synthesis. The component selection strategy makes use of a key observation: instructions that are semantically similar, and hence, more useful in synthesis, also share a high degree of structural similarity in the instruction's AST. To address (ii), we develop *instruction factorization*, a divide-and-conquer procedure that partitions a complex instruction into smaller pieces, each of which results in a disproportionately smaller synthesis search space. On top of these, we implement multiple optimizations to memoize work and guide synthesis towards more likely solutions earlier—in all cases guaranteeing that translations are semantically equivalent to the input instructions.

Putting it all together, SYNTHCT synthesizes translations for 75% of x86_64 ISA, i.e., is able to translate a large part of the ISA in terms of a small set of core instructions. Interestingly, the instructions that we do not have translations for are extremely simple and happen to match what experts today believe are safe instructions on microarchitectures running x86_64.

In summary, this paper makes the following contributions:

1) We develop SYNTHCT, the first framework to automatically generate translations from unsafe $\rightarrow$ safe instructions in an ISA.
2) We develop techniques to overcome several challenges in framing synthesis of unsafe instructions as a CEGIS problem. In specific, we develop techniques to reduce the synthesis search space and make the problem tractable on commercial ISAs.
3) We systematically evaluate synthesized solutions from SYNTHCT for both performance and security. In our evaluation, we find that SYNTHCT is able to generate safe translations for a majority (75%) of the ISA using only a small set of core instructions (the remaining 25%). In case studies, we show how SYNTHCT is capable of synthesizing safe translations for complex unsafe instructions, e.g., the divide instruction (`DIVL-R32`).

We believe SYNTHCT advances the state-of-the-art in constant-time programming. SYNTHCT is a robust, modular framework which is capable of being re-targeted to other microarchitectures and ISAs with minimal engineering effort. We have open-sourced SYNTHCT as two repositories:

- SYNTHCT: https://github.com/FPSG-UIUC/synthCT
- Synthesized translations: https://github.com/FPSG-UIUC/synthCT-artifacts

II. Threat Model and Scope

We consider a standard threat model for constant-time programming, where a victim program runs on a shared machine in the presence of adversarial software. The adversary's goal is to learn private data in the victim program through microarchitectural side channels. (This is equivalent to the SGX adversary [33], [31], [19], [16], modulo a caveat stated below in "Non-goals".) The program itself is considered public. We trust that, for a given microarchitecture, every instruction marked safe in the safe set specification executes with operand-independent hardware resource usage.

1) Security/Functionality goal: Given a mostly constant-time program P that takes input y and safe set S (a subset of the ISA), our goal is to automatically construct a program P' with the following properties:

1) *Functionality:* We require that $P(y) = P'(y) \ \ \forall y$.
2) *Security:* We require that for each instruction $I \in P'$, we have $I \in S$.[1]

By definition of our threat model, satisfying Property 1 implies non-interference with respect to the program input and the microarchitectural side channel attacker, on a non-speculative microarchitecture. That is, it implies $\forall y, y' \ \ \mathsf{View}(P', y; \mathsf{uArch}_S) = \mathsf{View}(P', y'; \mathsf{uArch}_S)$ where $\mathsf{View}(.)$ returns the program's hardware resource usage trace in space and time when run on non-speculative microarchitecture uArch_S with safe set S.

2) Complementary: Spectre mitigations for constant-time programming: Speculative execution attacks (e.g., Spectre [11]) use a program's *transient execution* to form *accessors* and *transmitters* that access and transmit (leak) a secret [34]. Analyzing what accessors can be constructed for a given program and microarchitecture, i.e., analyzing transient reachability, is out of our scope and is covered in complementary work [35], [36], [37]. Analyzing what transmitters are possible for a given microarchitecture is in scope: 'transmitter' is a synonym for 'unsafe instruction.' In other words, SYNTHCT enables complementary works [26], [27] to produce microarchitecture-specific Spectre hardened code, taking into account that machine's safe set.

[1]Requiring that all instructions are safe is overly strong. More precisely, constant-time programming requires that unsafe instruction operands are not a function of private data. Accommodating programs in this relaxed definition would require a complementary, well-understood analysis that can easily be applied on top of SYNTHCT.

3) Non-goals/limitations: Physical side channels (e.g., power [38] or EM [39]) are out of scope. Similar to other constant-time defenses, we treat SGX-based attacks that monitor analog information, e.g., the RAPL interface [40], as out of scope.

We also do not prevent secrets from leaking through a program's control flow and memory access pattern. That is, we do not block leakage through cache/memory- and control flow-related side channels [5], [3], [4]. We assume the input program is already hardened against these attacks, i.e., is *mostly constant time*. We elaborate on how microarchitecture can *still* undermine the security of mostly constant-time code in Section III. Writing and generating mostly constant-time code is a complementary concern, and has become practical due to a large body of prior work [13], [14], [15], [16], [17], [41], [18], [19], [20], [21], [22], [23], [24], [7], [25], [29], [42].

III. What Instructions May Be Unsafe?

Before describing SYNTHCT, it is important to understand why instructions become unsafe, and which instructions are likely to flip between safe and unsafe, depending on the microarchitecture. To start, *mostly constant-time code* (Section I) assumes that control-flow and memory instructions are unsafe. Due to their significant influence on program execution, it is 'safe' to assume that control-flow instructions are always unsafe. Likewise, due to myriad memory-system optimizations such as caches, which enable cache-based side channels [3], [4], it is safe to assume that memory instructions (loads, stores) are also unsafe on performance-competitive processors.

This paper's focus is therefore non control-flow and non memory operations. Prior work has studied how a small number of "complex" arithmetic instructions, such as floating point [7], divide [25] and multiply [8] are unsafe ("variable time") on certain microarchitectures. Given tools today, the only fix is to assume they are unsafe on all microarchitectures. This is secure, but overly conservative. SYNTHCT can improve performance when code is run on microarchitectures where such instructions are safe.

Worse, there is a large family of ISA-invisible microarchitectural optimizations that may render a significantly larger set of arithmetic instructions unsafe [28]. For example:

- Computation simplification / elimination optimizations (e.g., [43], [44], [45]) have been proposed for many arithmetic operations to take advantage of operation-specific identity and absorption properties. For example, that $x \,\&\, 1 = x$.
- Computation reuse optimizations (e.g., [46], [47], [48]) memoize computation when the same instruction(s) are executed twice with the same operands.
- Value prediction (e.g., [49], [50], [51]) saves cycles when an instruction returns a predictable result.
- Significance compression (e.g., [52], [53], [54]), related to serial computation, impacts performance depending on the position of the high-order on bit in each program word.

These optimizations are a major security concern. They can be implemented on any microarchitecture at any time.[2] Further, they are often implemented on a per-instruction basis as they often require instruction-specific logic (e.g., the identity rule for AND is different than that of OR). This makes it unlikely that which are safe will be consistent across microarchitectures. SYNTHCT can improve security when code is run on microarchitectures where such instructions are unsafe.

[2]Some indeed have been already. For example, [8] is an implementation of significance compression.

IV. Design Overview

In this section, we present an overview of SYNTHCT. Figure 1 shows an overview of SYNTHCT's workflow. At a high-level, SYNTHCT takes as input an ISA specfication and microarchitectural safe-set specification and generates a library of safe translations that implement every instruction in the ISA using only the instructions that are safe on that specific microarchitecture. The generated library may then be used to secure mostly constant-time code (or binaries) using a late-stage compiler pass (or a binary-rewriting mechanism). The final output is a binary that is constant time (with respect to that microarchitecture). Therefore, SYNTHCT operates in three distinct phases: an offline synthesis phase, a microarchitecture targeting phase, and an online deployment phase. Now we discuss each step and artifacts in the workflow in more detail.

1) ISA Semantics: SYNTHCT takes as input the semantics for each instruction in the target ISA. The semantics describe the precise functional operation of each instruction in some intermediate representation. In our current implementation, SYNTHCT takes x86_64 semantics written in the K-framework [55], but the framework is conceptually agnostic to which ISA and semantics IR. The semantics for each instruction describes how the output registers and the flag states are computed. The semantics can be reduced to an abstract syntax tree (AST) representation that uses the K-language (/opcodes). Therefore, there is one AST per output register and flag that the instruction sets. An example of such an AST is shown in Figure 3 for the instruction "Add with Carry" (`ADCQ`) and the output register. Similar ASTs describe how each of the other x86_64 flags are computed by the instruction. Note that the semantics may not describe *how* the instructions are implemented in hardware nor do they indicate what instructions may be safe. These are simply functional specifications.

2) Safe/Unsafe Specification: SYNTHCT also takes as input a microarchitecture-specific input that specifies the set of safe and unsafe instructions for that particular microarchitecture. Safe sets may be provided by the hardware manufacturer, or reverse engineered like unofficial currently-used safe sets [7]. We hope, long term, that this and related work [32] encourages processor manufacturers to publish formal, explicit microarchitecture-specific safe sets.

A. Step 1 (Offline): Synthesis

The synthesis takes as input the semantics for all instructions in the ISA and generates a library of translations. In the first stage, SYNTHCT does not assume a specific safe set and instead tries to find translations from any instruction into any other instruction(s). Therefore, the synthesized set of translations is microarchitecture agnostic. Alternatively, if the target microarchitecture and its safe set are known at

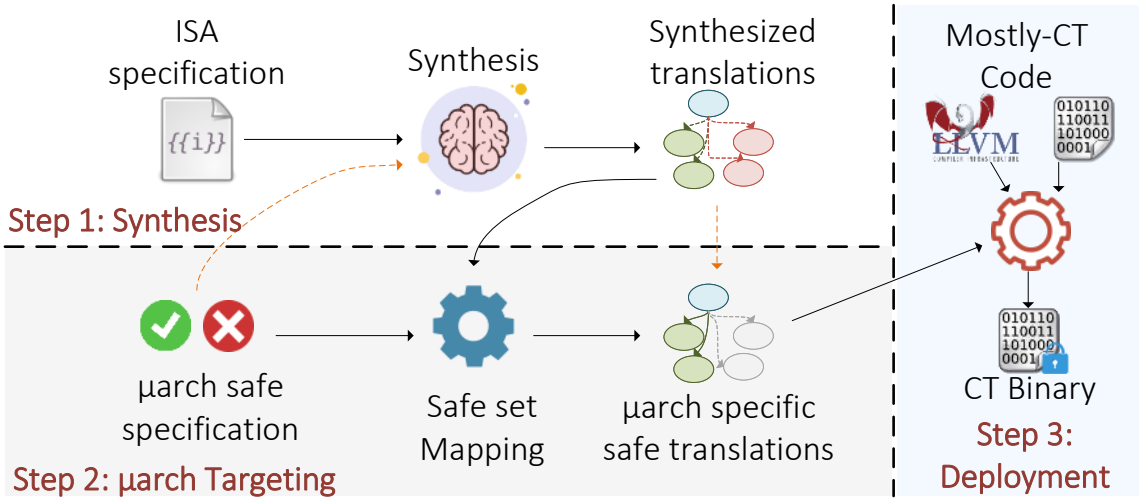

Fig. 1: Overview of SYNTHCT. SYNTHCT takes as input the specification of instructions in an ISA and uses program synthesis to generate a library of translations. When targeting a particular microarchitecture, SYNTHCT uses a safe set specification for the microarchitecture to generate a library of *safe* translations for all unsafe instructions. The set of safe translations can then be used at deployment time to secure mostly CT source code or binaries through a compiler or binary rewriting pass. Parts of the main workflow are shown with filled lines and a possible alternate workflow is indicated with dashed lines.

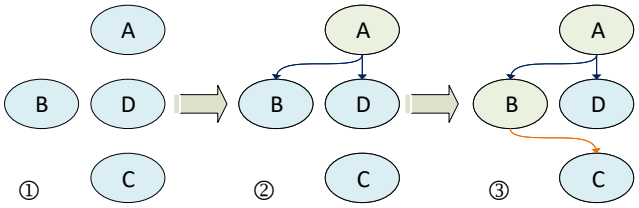

Fig. 2: Iterative Synthesis. SYNTHCT performs synthesis iteratively and updates the synthesis graph with generated translations. ① shows the initial synthesis graph with instructions A, B, C, D having no translations. In ② and ③ SYNTHCT synthesizes translations for A and B and updates the graph.

synthesis time, they may be used in the synthesis step to generate microarchitecture-specific translations. We discuss this alternate work flow at the end of this section.

1) Iterative Synthesis: The synthesis process discussed above is iterative. SYNTHCT constructs a graph using all discovered translations. Such a *synthesis graph* is shown in Figure 2. In Figure 2, SYNTHCT starts of with a graph containing a node for every instruction in the ISA and no edges (①). As SYNTHCT synthesizes solutions, e.g., in ② instruction A is synthesized using instructions B and D, edges are added to the graph. Finally, as shown in ③, SYNTHCT synthesizes a solution for instruction B using instruction C, and therefore, instruction A may also be translated using instruction C and D. This allows SYNTHCT to discover translations that may otherwise not be synthesizable in a single step due to the complexity of the solution.

B. Step 2 (Offline): Safe-set Mapping

SYNTHCT then uses the microarchitecture-agnostic set of translations and the safe-set specification for a given microarchitecture and generates the set of translations specific to that microarchitecture, i.e., every unsafe instruction in the ISA has translations using only the instructions from the safe set. This step ensures that every instruction is expressed with the best possible set of safe instructions, according to some performance model. For safe instructions this is trivial as they need not be translated. For unsafe instructions it boils down to using the best performing synthesized solution that uses only safe instructions. SYNTHCT uses program length as a proxy for performance, but a lower-level, more accurate and microarchitecture-specific performance model may also be used for selecting the best synthesized solutions. The end result from this step is a library of translations that is specialized to harden software for a specific microarchitecture.

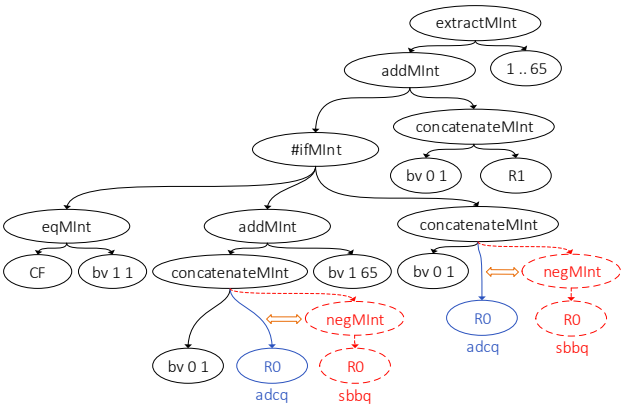

Fig. 3: Semantics AST for the `adcq` instruction's output register (black and blue nodes in the AST), written in K [55]. The figure also shows the structural similarity between `ADCQ` and `SBBQ`: to derive `SBBQ` from `ADCQ`, one replaces the blue nodes with the red nodes.

C. Step 3 (Online/Compile time): Software Hardening / Deployment

During deployment, the library of translations generated by SYNTHCT is used to secure *mostly* constant-time code.[3] We envision this step to be integrated as a late-stage compiler pass in popular compilers, e.g., GCC or LLVM, to replace any unsafe instructions selected by the compiler backend by their safe translations. Optionally, this may also be implemented as a post-compilation pass using binary rewriting to secure mostly constant-time binaries, e.g., binaries produced by DSLs like FaCT or handwritten to be constant time [7], [29], [13], [14].

The key component of SYNTHCT is Step 1, Synthesis. In the sections that follow we will first frame the synthesis problem as a CEGIS problem, then, highlight several challenges in applying the CEGIS formulation directly, and finally, present techniques that SYNTHCT implements to solve these challenges. The deployment step can be implemented with minimal additional engineering effort, e.g., writing a compiler pass that uses synthesized translations stored in a machine-readable format.

D. Alternate Workflow

If the safe set specification and the target microarchitecture is known at synthesis time, SYNTHCT can take such a specification as input in step 1, shown by a dashed line in Figure 1, and directly synthesize safe translations that are specific to the microarchitecture.

In the following section, we highlight several challenges facing both workflows. To summarize these challenges: On the one hand, if the size of the safe set is large, then both workflows face similar scalability issues. On the other hand, if the size of the safe set is small and simple, then the more complex instructions in the ISA may not be synthesizable in one-shot.

V. SYNTHCT: FORMULATION AND CHALLENGES

This section describes the design and implementation of SYNTHCT. First, we formulate the problem of instruction synthesis as a Counter Example Guided Inductive Synthesis (CEGIS) problem in §V-A. Then, we show several challenges that we need to solve to make synthesis tractable for an an ISA like x86_64 in §V-B. Finally, in the sections that follow, we discuss several techniques we develop in SYNTHCT to tackle these challenges.

A. Formulation as a CEGIS Problem

1) CEGIS Sketch: We formulate the problem of instruction synthesis to an off-the-shelf CEGIS tool by specifying: (i) a specification of the synthesis goal and (ii) the sketch used to generate candidate programs. In our setting, the specification is the semantics of the *target instruction*, I_t, in bitvector logic. Candidate programs that implement the target instruction are generated from the sketch shown below. Our sketch is simple and general: A *program* is expressed as a list of x86 instructions, and each instruction is parameterized by zero or more register operands (as needed by the instruction).

```
Program (P): (list Inst*)
Inst (I): (choose* {??})
{??}: x86 Instruction { ??: register operands }
```

The {??} are *holes* that need to be filled in by synthesis to generate concrete programs. Here, the first such hole can be filled in by choosing an appropriate *component*[4], i.e., an instruction, from a set of available components, i.e., all instructions or a subset of instructions from the ISA. The second hole can then be filled in by choosing register operands for the selected instruction from the pool of available registers. Based on this sketch, the CEGIS procedure has three degrees of freedom in generating concrete programs: (i) The length of program to synthesize, (ii) The choice of instructions (components) for each "line" in the program, and (iii) The register operands to each instruction.

2) Verifier: The verifier checks if the synthesized program and the specification are semantically equivalent. To do so, the verifier interprets the specification and the candidate program starting from blank (symbolic) states $\mathbf{S}$ and $\mathbf{S}'$ respectively. Then, it performs an equivalence check over the corresponding register and flags states in $\mathbf{S}$ and $\mathbf{S}'$. If the final states are equivalent, then the candidate program implements the specification and we have a synthesis solution. Otherwise, the verifier produces a counterexample that is used to refine the future solutions. In our setting, the verifier uses the SMT solver, Z3 [56], to perform equivalence checks and generate counterexamples.

3) CEGIS Implementation: For synthesis, SYNTHCT uses Rosette [57] a solver-aided DSL that extends Racket to make it easy to develop various tools, e.g., a symbolic interpreter, synthesis engine etc.

In theory, above formulation is sufficient to synthesize programs for all instructions I_t in an ISA in terms of a safe set of instructions. However, in practice, this formulation of synthesis does not scale to an ISA as large and complex as x86_64 due to the large synthesis search space, as we see below.

B. Challenges

To test the scalability limits of our synthesis problem formulation, we perform several synthesis experiments by varying the parameters: (i) The length of synthesized programs, (ii) Number of available instructions for synthesis, and (iii) Number of registers available for synthesis. For this experiment, we consider the synthesis of the "Add with Carry" (ADCQ) shown previously in Figure 3, with six registers available to use in synthesis. Figure 4 shows the log of synthesis running time vs. synthesis program lengths for different component set sizes.[5] The timeout is set to one day for this experiment. The

[3] That is, code whose control flow and access pattern to data memory is independent of private data, but may pass private data to unsafe instructions.

[4] 'Component' is a synthesis term. In this paper a component is an instruction opcode.

[5] Setting the program length to N instructions or components forces synthesis to find only solutions with that number of instructions. If there exists a solution with fewer than N instructions, say S, synthesis will still need to "find" a solution S' (in a larger search space) that is, e.g., S padded with nops (either literal nops or sequences that are semantically nops) to reach N total instructions.

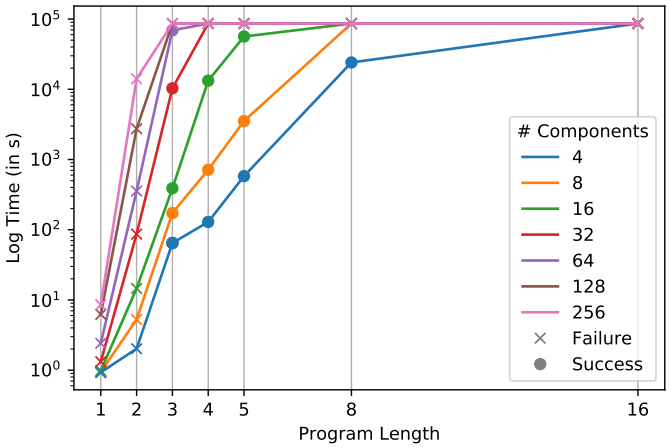

Fig. 4: Synthesis Scalability. The graph shows synthesis time, for the running example ADCQ, vs. program length for different numbers of components. Component sets for synthesis are expert selected such that at least one known solution exists. Points with ● denotes a success and ✕ denote failure, i.e., unsat/timeout. Timeout is set to one day for this experiment.

set of available components are expert chosen to ensure that at least one possible solution exists for ADCQ. Points in the parameter space that lead to successful synthesis solutions are marked by a ● while ones that fail, either due to unsat (no possible solution) or a timeout, are marked by an ✕.

From the graph, we see that synthesis time is a function of both synthesis program length and the number of available instructions to use in synthesis. The synthesis time increases exponentially with the increase in both the program length and the number of available instructions. We see that setting the program length to 3 gives us the first successful synthesis and increasing the program length results in exponential increases in the time to success — only the curve with a small number of available components (4) succeeds for program length 8 and all available instruction set sizes fail due to timeout after one day for programs of length 16. We see a similar trend when exploring different-sized sets of available components — increasing the set size exponentially increases the time taken for success while keeping program length constant, with larger set sizes, e.g., 128, having successes only for very short program lengths (3) and the largest set (# of available components = 256) having no successes.

This experiment makes it clear that both parameters have a 'goldilocks zone' in which synthesis succeeds and takes a reasonable amount of time. On the one hand, we require that the program length, # of available components and # available registers is small. On the other hand, if we do not sufficiently provision each of these, synthesis cannot find a solution.

1) Setting the program length: For example, if we make the program length too short, e.g., points in the graph where program length is 1 or 2, then the synthesis fails as there may be no programs of such short lengths that can implement the target instruction. Making matters worse, given an arbitrary target instruction to synthesize, it is not possible to say ahead of time if such a short solution exists. In fact, in x86_64, we see several such instructions, e.g., Leading Zero Count (LZCNTQ), that are complex and cannot be synthesized with programs of short length. Thus, we need to solve two problems related to setting program length: a) how to choose the right program length for a given target instruction and b) how to scale synthesis to handle complex instructions that require intractably large program lengths.

2) Setting the component set: When looking at the number of components to make available for synthesis, we face a similar issue. Smaller sets of available components lead to quicker synthesis successes. As shown from our experiments, providing the entire ISA to synthesis as choices is infeasible. For example, the synthesis time given a candidate component set size of 256 (256 out of 451 instructions) for programs with lengths > 2 is more than a day. Therefore, the naive solution of using all instructions in the ISA in synthesis is infeasible and we need to develop a way to automatically (without expert guidance) select a good subset.

A strawman idea to address this "component selection" problem is to randomly sub-sample the ISA. The smaller the size of the subset, the lower the probability of picking a 'good' set of components, that contains the components needed for a solution, while a larger subset will exponentially increase the synthesis time. To check whether this simple strategy is sufficient, consider the probability of choosing a good subset of size 32 (as 32 was the largest component size yielding successes in Figure 4). Then there are $\binom{451}{32}$ possible subsets out of which only $\binom{448}{29}$ have the required three components for the solution. Therefore, the probability of picking a good subset is ≈ 0.000326 which is extremely low. Choosing subsets with fewer instructions yields similar results. Therefore, to keep the synthesis time tractable while generating synthesis successes, we need a better way to automatically select component subsets for synthesis based on the target instruction.

3) Setting the number of available registers: In the above experiments, we keep the number of available registers a constant (6). We observe similar scaling trends when varying the number of registers. Due to limitations in the current sketch, synthesized programs cannot spill and restore registers to/from memory. Therefore, having too few registers means that certain solutions that need more registers cannot be synthesized, while having too many registers increases synthesis time.

In this paper we will focus on addressing issues with program length and component sets (see previous two sub-sub sections). Combined with solutions to these issues, we find that the naive strategy for registers—limiting the number of available registers to a smaller number than the actual number of registers in x86_64 (16 general purpose registers)—is sufficient. There may be room to develop heuristics to automatically pick the required number of registers based on the complexity of target instruction or to iteratively increase the number of registers when synthesis fails to produce a solution. We leave developing such heuristics for future work.

VI. SYNTHCT DESIGN

From the above experiments, we must address several challenges.

First, we must coordinate the program length with the number of components we actually expect will be required. This is especially when the program length is large, e.g., as we have described with the LZCNTQ example, as synthesizing programs of this length is intractable. Second, regardless of the program length, we need to devise a way to sample the ISA so as to include components that are likely to lead to successful synthesis. We now give a high-level overview of how we solve these challenges:

1) Component selection: As the naive random sampling is unlikely to generate many synthesis successes, we develop a better strategy to pick a good set of components to maximize the likelihood of synthesis success. The key insight behind our component subset selection is that *structural similarity between instruction ASTs can be an indicator for semantic similarity*. We build on this insight to develop our *component selection* strategy in §VI-A.

2) Instruction factorization: As the program length needs to be short, certain complicated instructions in x86, e.g., Leading Zero Count (LZCNTQ), cannot be synthesized because no loop-free program of short length can implement such instructions. To solve this, we develop *instruction factorization*, a divide-and-conquer strategy to synthesize solutions in parts, in §VI-B. Sometimes, factorization is insufficient to implement certain complicated instructions, e.g., the Divide instruction. To synthesize such instructions we introduce several additional techniques: *node splitting* and *pseudo-instructions* (§VI-C).

3) Register handling: To reduce the synthesis search space, we restrict the number of registers to just 6 registers, compared to 16 available general purpose registers in x86_64. We found that combined with the solutions discussed above, this simple strategy of restricting the number of registers was sufficient for synthesis. In the future, we may explore more sophisticated strategies to tailor the number of registers based on the synthesis need.

4) Miscellaneous issues: Lastly, we discuss other miscellaneous challenges and solutions in synthesizing x86 instructions in §VI-D.

A. Component Selection

To make the synthesis problem tractable we need to restrict the synthesis search space by allowing synthesis to only choose components from a small subset of instructions rather than the entire ISA. From our experiments, it is clear that choosing components at random is unlikely to select a good subset of components for synthesis. In this section, we design the component selector, that for a given instruction to synthesize, picks a set of components that maximizes the likelihood of a synthesis success.

Key Insight. The key insight behind our component selection is that instructions that are semantically similar, and hence more useful for synthesis, have *structurally similar* semantics. In other words, their ASTs are structurally similar to each other. For example, in the ADCQ example from Figure 3, one can form the AST for the "Subtract with Borrow" (SBBQ) instruction by replacing the blue nodes in the AST with red nodes. This makes sense as a subtract is addition with a negation. If we give SYNTHCT a set of components that includes sbbq, it does synthesize the desired solution:

```
adcq R0, R1:
  movq 0x0, R3
  sbbq-r64-r64 R0, R3
  subq-r64-r64 R3, R1
```

To restate the component selection problem: given an instruction I_t to synthesize, return a list of instructions, or components, from the ISA ranked according to their similarity to the synthesis target I_t. Generally, we find that sub-graphs of the target instruction's AST are sub-graphs of a candidate component's AST. For example, the ADCQ↔SBBQ example from Figure 3. Therefore, we need a fuzzy way to estimate similarity between instructions and assign a score based on how similar the instructions are. Performing subgraph matching on all instruction ASTs is too expensive; we therefore need a lightweight way to estimate the degree of similarity between ASTs.

1) Implementation: SYNTHCT implements the fuzzy similarity estimation in two steps:

1) Graph Embedding: to convert variable-sized graphs, i.e., instruction semantics expressed as an AST, to a fixed-size vector representation,
2) Similarity Estimation: that uses vector representation of ASTs to assign similarity scores.

For graph embedding, we use graph2vec [58], an unsupervised machine learning technique similar to word2vec [59] used in NLP settings. graph2vec captures structural similarity between graphs and generates similar embeddings for graphs that are structurally similar to each other. Our implementation uses graph2vec as implemented in the python library karateclub [60]. graph2vec takes a number of hyper-parameters for training, we refer the readers to [58] for full details. Empirically, we found the following learning parameters to be sufficient to find good results for synthesis (§VII-E1): We set the number of WL iterations to 3, the number of training epochs to 30, and the size of the output vectors to 128. We did not try to optimize these hyper-parameters further or do an exhaustive search. We leave optimizing these hyper-parameters as future work.

The output from the graph embedding stage is a fixed-length vector representation of the semantics of each instruction. Then, to find instructions that are most similar to the query instruction we use K-Nearest Neighbors (KNN) search. We use Euclidean distance for the distance metric and set the number of neighbors to 32 ($K = 32$) to get the 32 instructions most similar to the query instruction (as 32 was the largest size that yielded successes in reasonable time; c.f. §V-B).

B. Instruction Factorization

In order to keep synthesis times tractable, we are restricted to synthesizing programs of short lengths (e.g., 4 instructions). However, there is no guarantee that *any* instruction from *any* ISA will be synthesizable with a program of such short length. For example, certain instructions in x86_64 such as the leading zero count (LZCNTQ) instruction have complex semantics and require hundreds to thousands of simpler instructions to

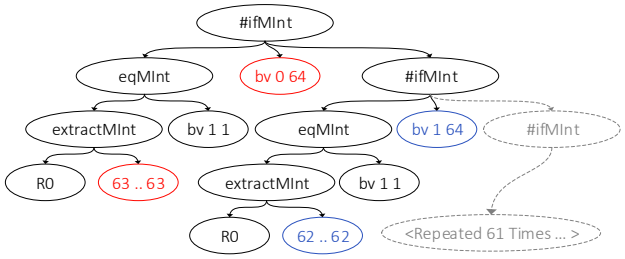

Fig. 5: Simplified Semantics for Leading Zero Count Instruction (`LZCNTQ`). The figure only shows the first two levels of the AST. Observe that the structure of `if-then-else` is repeated multiple times, but with the conditional comparing different bits and the `then` branch returning a different value. The differences in the two levels are highlighted in red and blue respectively.

synthesize (§VII-B). We therefore need additional techniques to scale and generalize synthesis.

In Figure 5 we show the simplified semantics of `LZCNTQ` to illustrate potential complexity in instruction semantics. The original, un-simplified AST has 771 different nodes with a depth of 64. Taking a closer look, the AST has the same substructure repeated multiple times (as shown in dashed gray lines). Indeed, this is because the same operation is repeated on different bits, i.e., bit 0 through 64, of the operand.

To be able to synthesize such complex instructions while keeping synthesis time tractable, we develop *instruction factorization*, a divide-and-conquer technique that breaks down a complex instruction semantics AST to multiple, simpler ASTs. Each of the smaller ASTs are treated as semantics for simpler *factors*. The factors are then synthesized as separate synthesis tasks, like regular instructions in SYNTHCT. Once the intermediate factors have solutions, the original larger instruction is synthesized exclusively using these smaller factors.

1) Factorization Strategies: There are multiple ways to *factorize* an instruction's semantics. For a good factorization, we need to find a minimal-size partition of the semantics AST, such that each of the sub-ASTs that are generated by the partition are synthesizable in parts. In general, it is not possible to say if a particular partition is synthesizable without investing time into trying to synthesize solutions from it.

In this work, we primarily use a bottom-up factorization strategy. The intuition is to try the simplest ASTs first, i.e., starting from trying to synthesize sub-trees from the target instruction's semantic's leaves with height $= 1$, and incrementally combine the simpler solutions to synthesize larger and larger subtrees.

To illustrate this strategy, consider the example in Figure 6 that shows an example AST, the generated factor types, and the bottom-up factorization workflow graph. In the AST (Figure 6a), nodes are labelled such that a lowercase label (letter) indicates a single node in the original AST and an uppercase label refers to a subtree which may have one or more nodes. When the same node label appears two or more times, that means the same node or subtree appears that many times. We differentiate between occurrences via subscripts.

SYNTHCT synthesizes the AST by partitioning it into several types of smaller synthesis tasks (Figure 6b). To synthesize AST leaves, SYNTHCT creates a synthesis task to translate just the leaf. As the AST can be recursively viewed as a single node with subtrees for operands, we create a synthesis task for each node and set the subtrees as stubs. We then recursively factorize the subtrees, eventually reaching the leaves, e.g., the subtree E_1 eventually reaches its leaves and their corresponding subtree of tasks are abstracted behind the task Synth–E_1. To synthesize AST non-leaves (e.g., d_1), SYNTHCT first tries to synthesize the root node (e.g., d_1) in isolation, leaving the children as stubs/residuals (denoted Synth–d_{1r}). It then tries to synthesize the whole rooted AST after both the root node and children have been resolved (denoted Synth–d_1).

Finally, the workflow graph (Figure 6c) shows the order of in which SYNTHCT schedules the synthesis tasks to synthesize the overall AST. SYNTHCT starts synthesis of factors from the leaves of the workflow graph and proceeds to synthesize the parents when all the subtasks return success. Child tasks beneath a single parent are attempted left-to-right; hence, the residual task is synthesized first, followed by tasks for each child subtree. This way, synthesis works bottom-up, synthesizing the simplest factors first before synthesizing larger, more complicated factors, eventually synthesizing the complete instruction. Additionally, when synthesizing non-leaf factors, the sub-factors of the target factor are available as components to use in synthesis.

2) Optimization: Factorization may create a secondary problem: there may be too many simple, overlapping synthesis tasks to run in a reasonable amount of time. Consider the example from earlier (Figure 6) and its corresponding bottom-up workflow graph (in Figure 6c). In this example, notice that the subtrees rooted at d_1 and d_2 are equivalent, and yet, we have separate synthesis tasks, Synth-d_1 and Synth-d_2, in the workflow graph. In fact, we see massive memoization opportunities in the wild when synthesizing several x86_64 instructions, e.g., the aforementioned Leading Zero Count (`LZCNT`) shown in Figure 5, Population Count (`POPCNT`), Trailing Zero Count (`TZCNT`) etc. Indeed, this is because `LZCNTQ` (and similarly the other examples) performs the same repeated operations on bits 1 through 64 of its operands. To reduce the number of synthesis tasks we exploit this observation and memoize the synthesis results. When a new factor needs to be synthesized, SYNTHCT first computes the hash of the subtree and checks if a solution has already been computed by another synthesis task.

C. Node Splitting and Designing Pseudo-Instructions

Instruction factorization solves the problem where the AST is dominated by many vertices, each with relatively simple functionality. We now tackle the opposite problem: where the AST is dominated by few vertices, each with complex functionality, e.g., a divide opcode. These opcodes are not translated in K internally as they have a one-to-one correspondence with opcodes in SMTLIB, thereby enabling direct formal reasoning such as equivalence checking.

The fundamental reason we cannot handle complex intermediate vertices is because SYNTHCT has no structural infor-

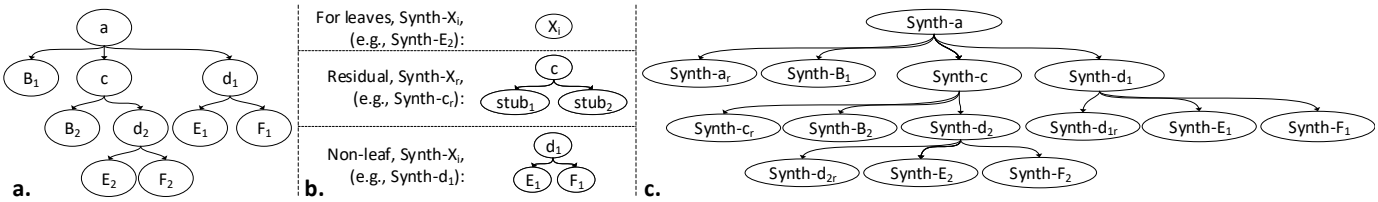

Fig. 6: Instruction factorization example. (a) An example instruction AST to be factorized. Lower-case labels are AST primitive operations; upper-case labels are subtrees of AST operations. Subscripts denote different instances of the same AST subtree. (b) The different types of synthesis tasks spawned to synthesize the AST. (c) The synthesis workflow graph for (a), depicting the order in which synthesis tasks are evaluated (bottom to top).

Pseudo-Instruction	Description
movq-imm-r64 i, r0	Move an immediate value `i` into the register `r0`
pmovq-r64-r64 r0, r1	Move from register `r0` to `r1`
pmov-flag-r64 f, r0	Move value of flag `f` into register `r0`
pmov-r64-flag r0, f	Move value in `r0` into flag `f`
psplit-r64-r64 r0, r1, r2	Split value in `r1` into `r0` and `r1` at index specified in `r2`
pconcat-r32-r32 r0, r1, r2	Combine `r0` and `r1` into `r1` by shifting value in `r0` by value in `r2`
pcmov-r64-r64-r64 r0, r1, r2	Conditionally move `r1` into `r2` depending on if the condition `r0` is true
pnot-r64 r0	Logical not
por-r64-r64 r0, r1	Logical or
pand-r64-r64 r0, r1	Logical and
pxor-r64-r64 r0, r1	Logical xor
pnop	No-op (NOP)

TABLE I: List of pseudo-instructions implemented in SYNTHCT.

mation in the semantics to exploit during component selection, i.e., since each individual vertex is complex and abstracts away multiple operations, picking a good set of components to implement the functionality of that said vertex is not possible. Therefore, SYNTHCT needs additional assistance in order to be able to exploit structural information and synthesize such complex instructions.

In the following two subsections, we develop two techniques to handle this source of complexity: (i) expert-written pseudo-instructions to provide synthesis with some useful primitives and encourage creative solutions, and (ii) node splitting to manually represent complex intermediate K-operations as simpler operations.

1) Pseudo-Instructions: The intuition behind pseudo-instructions is to have some extremely useful expert-chosen primitives, e.g., generating bitmasks, extract bits `i` through `j` from a register etc., be available in all synthesis tasks. These expert-chosen primitives are treated as normal unsafe instructions and are synthesized by SYNTHCT as usual. They can then be used as subroutines in subsequent synthesis jobs. Well chosen pseudo-instructions solve three problems: (i) re-discovering implementations for basic primitives wastes synthesis cycles, (ii) by abstracting away useful primitives that need multiple instructions to implement behind a pseudo-instruction, we effectively compress the program lengths of synthesis solutions that need these primitives, and (iii) if chosen correctly, pseudo-instructions will encourage creative solutions that synthesis may not have found otherwise. Pseudo-instructions are ISA independent and therefore can be used in synthesis of instructions in other ISAs as well, although, their usefulness may vary depending on instructions and complexities of the ISA.

The current set of pseudo-instructions we designed for SYNTHCT along with their descriptions are shown in Table I. Just like regular instructions, pseudo-instructions are synthesized by SYNTHCT, except that the set of instructions used as components during the synthesis is a set of simple, fixed instructions. These need not be in the safe set for a specific microarchitecture and, if not, will themselves require translations to the safe set. In other words, the pseudo-instructions do not go through the component selection process. This is currently an engineering limitation[6] which can be addressed in future work.

2) Node Splitting: We use node splitting to simplify complicated K intermediate opcodes, e.g., the K `divide_64` opcode in the division family of instructions, `ROL/ROR` in the rotate family of instructions, and `MUL` in the multiply family of instructions. K-opcodes are finite and shared across ISAs, i.e., instructions from any ISA need to be expressed in terms of these fundamental K-opcodes to be compatible with formal reasoning in the K-framework. Thus, node splitting is a one-time manual implementation effort, per complex K-opcode, and need not be re-done if SYNTHCT is re-targeted to a different ISA.

Once such translations are implemented in SYNTHCT, ASTs of instructions that contain the complex opcodes are first simplified by replacing the opcode with the translation before proceeding with the rest of SYNTHCT's workflow. We implement such translations in SYNTHCT for the the rotate K-opcode, `rol`, and the K-opcodes used in the divide instruction, `div_quotient_int32` and `div_remainder_int32`. This will be discussed in our evaluation §VII-C. More such translations may be implemented in SYNTHCT on-demand with minimal effort.

The overall algorithm uses node splitting with instruction factorization and component selection as follows. First SYNTHCT uses node splitting to simplify complex vertices in an instruction AST using translations built into SYNTHCT. This process converts an AST with a few complex vertices into an AST with simpler, but larger number of vertices. Second, we use instruction factorization to synthesize the now larger AST through the divide-and-conquer mechanism that we described earlier. We note that due to program length constraints, node splitting would likely be ineffective without factorization.

[6]Pseudo-instructions in SYNTHCT are directly implemented in racket (/rosette) rather than K and therefore cannot go through the usual flow through SYNTHCT that other instructions written in K can go through. Therefore, with the current prototype implementation we cannot perform the usual component selection described in §VI-A.

Pseudo-instructions vs. node splitting. Both pseudo-instructions and node-splitting help SYNTHCT to synthesize complex instructions, albeit in slightly different ways. Pseudo-instructions are intended to be more generic and help synthesize more creative solutions that would otherwise not be possible, while node-splitting is specific to certain complex K-opcodes that are opaque and need to be simplified.

D. Other Challenges

Synthesizing x86_64 instructions, in particular, also introduces several additional challenges.

1) Synthesizing Multiple Solutions: The earlier sections describe how SYNTHCT generates a single solution. But to develop a rich set of translations so that they may be applicable to a wide-range of safe sets we ideally need to generate multiple synthesis solutions for a single instruction, each that utilizes minimally- or non-overlapping subsets of components.

To achieve this diversity, on a synthesis success, SYNTHCT generates multiple new synthesis tasks for the same instruction as a feedback mechanism. Each of the new tasks are generated by removing one of the instructions used in the synthesized solution. To do so, SYNTHCT uses the same K-Nearest Neighbor (KNN) heuristic developed in §VI-A to pick the K most similar instruction to the target instruction and then filters out the instruction(s) that were used in the synthesized solution. Therefore, a solution that uses `N` distinct instructions in the solution generates `N` new synthesis tasks as feedback. This sampling-without-replacement strategy encourages diversity by not re-using instructions across solutions.

2) Side-effects: Equivalence of Flags: Many x86 instructions also set ISA flags as side-effects, i.e., `RFLAGS`, in addition to the output registers(s). For example, the running example, the `ADCQ`, instruction sets the following flags: `OF`, `SF`, `ZF`, `AF`, `CF`, and `PF`, according to the result. To achieve complete equivalence to the instruction, not only does the output register need to be equivalent, but the flag state needs to be equivalent as well.

We treat synthesis of flags as a separate synthesis task where the objective is to synthesize these side-effects and ignore the output register. Once a solution that sets the flags is synthesized, it can be combined with the solution for the output register(s) to achieve complete equivalence. Glue code saves register values before the main computation, then, saves the contents of the output registers after the main computation, and restores the initial operand values for the flag computations. Lastly, it restores the saved result from main computation to the output register using a single `MOV` with no side-effects.

Of course, it may be more efficient to synthesize the flags with the output registers in one-shot in a single synthesis task. Yet, this may prevent synthesis from discovering solutions that only implement the computations needed to set the output registers of the target instruction correctly. Such partial solutions may be sufficient most of the time, as described below.

Including code to set the flags correctly introduces additional overhead. However, in many cases, the instruction side effects are ignored, i.e., the flags are never used before being clobbered again. Therefore, we store both solutions separately: the main solution that sets the output registers, and the additional code needed implement side-effects. When translating instructions in *mostly constant-time code*, we choose from one of the two variants depending on if the flags are live at that program point or not. This allows us to reduce the overhead and not needlessly emulate the instruction side-effects when the flags are never used before being clobbered.

3) Exceptions: Some instructions may generate operand value-dependent exceptions. For example, the divide instruction (`DIVQ`), generates an exception when the divisor is 0. These exceptions are problematic for SYNTHCT, and constant-time code in general. On the one hand, throwing the exception breaks constant-time programming. On the other hand, masking the exception breaks semantic equivalence (which is an important goal in SYNTHCT). By default, we can adopt well-established strategies (e.g., [16]) for masking exceptions in constant-time code, but leave implementing these solutions to future work.

VII. Evaluation

1) Framework: SYNTHCT is implemented in python in about 8000 lines of code. This code is responsible for parsing semantics, performing AST transformations, factorization, generating individual synthesis tasks, and orchestrating the feedback for the synthesis. Additionally, another 700 lines of code is implemented in racket to set up synthesis, implement a machine model, implement pseudo-instructions and interpret the generated K programs. The implementation uses formal semantics for x86_64 written in K [55]. For synthesis, SYNTHCT uses Rosette [57] a solver-aided DSL that extends Racket to make it easy to develop various tools, e.g., a symbolic interpreter, synthesis engine etc. Currently, SYNTHCT only synthesizes instructions with register operands. To handle memory operands, or instructions with immediate operands, we can augment the synthesized translations with an additional `mov` instruction to load (or store) from memory or load immediate operands into registers. Alternatively, the register operands in synthesized translations may be replaced by a memory expression when the target instruction uses a memory operand (assuming the instruction behaves identically when operating on registers v/s memory, albeit performing an additional load/store to memory).

2) Experimental Setup: All experiments below were performed on a server machine running Ubuntu 18.04 within a docker container. The two machines used had 80 cores each, with 128GB/256GB of ram respectively. Synthesis tasks were run in parallel to use as many cores as possible. Synthesis runs were performed in batches with our longest run lasting 3 days. Instructions requiring factorization were run separately in isolation. Over the course of all our experiments, SYNTHCT generated a total of 2617 synthesis solutions, generating at least one translation for 242 (/366)[7] non-vector, non-floating point instructions. We did not try synthesizing vector and floating point instructions. While we cannot show all translations for space, we show several representatives in Table V

[7]The remaining instructions (451-366) are all `mov` + `cmov` instructions that we do not run synthesis tasks for.

in the appendix. All results that we discuss below consider full equivalence of translations, i.e., output registers + flags (§VI-D2).

In the subsections that follow, we evaluate SYNTHCT for: (i) security, to understand and assess the safe sets resulting from SYNTHCT's synthesized translations, (ii) performance, to quantify the overhead introduced by using SYNTHCT's safe translations, (iii) two case-studies to illustrate the use of node-splitting (§VI-C) and factorization (§VI-B) in synthesis of the rotate left (`ROLL`) instruction and the divide (`DIVL-R32`) instruction, and (iv) lastly, highlight some secondary metrics of the synthesis process.

A. Security Evaluation

In this section, we evaluate the security implications of the generated set of translations. More specifically, we analyze the potential safe sets that are emergent from SYNTHCT synthesized translations. Then, we compare these emergent safe sets with those introduced in the literature.

1) Methodology: Deriving Safe Sets from the Synthesis Graph: We represent SYNTHCT's translations as a synthesis graph (§IV). To recap, the synthesis graph (shown in Figure 2) has a node for every instruction (and by extension, pseudo-instructions and factors encountered in the synthesis process). For a solution synthesized for a target instruction, I_t, the graph records the instructions used by the solution by adding an edge from instruction I_t to all the instructions used in the solution. We store additional metadata for each edge, e.g., a unique solution, the line number in the synthesized program, and concrete values of operands to use. We iteratively build up the synthesis graph as SYNTHCT synthesizes translations for all instructions in the ISA.

Once we build a synthesis graph corresponding to a set of translations, the next step is to identify the emergent safe sets. Specifically: we define SYNTHCT's safe set as those instructions that have no translations, i.e., outgoing edges in the synthesis graph (conversely, instructions that have at least one outgoing edge may not be needed in the safe set). By definition, these instructions cannot be rewritten in terms of any other instruction. Thus, any microarchitecture using SYNTHCT will require that its microarchitectural safe set be a superset of the SYNTHCT safe set (more details are given in §VII-A3).

Deriving the safe sets from the synthesis graph is non-trivial because the synthesis graph may contain cycles. For example, consider synthesis of instructions A and B where solution to A may use instruction B and vice-versa, thereby creating a cycle in the synthesis graph. In this case, neither A nor B are leaves, but at least one of them needs to be included to form the safe set. In other words, the safe set is not unique. Therefore, the first step is to reduce the synthesis graph to a directed acyclic graph (DAG) by eliminating such cycles in the graph. To do so, we first identify the cycles in the graph by using Tarjan's algorithm [61] to identify the strongly connected components (SCC) in the graph. Next, we replace the nodes in the cycle with a single node. The new node is labeled using a disjunction of node labels (instructions) that form the cycle. We then redirect any incoming edge to the cycle, i.e., an edge from a node not in the cycle to a node that is a part of the cycle, to the new node. Similarly, any outgoing edge from the cycle is redirected from the new node. We apply this procedure multiple times until all cycles in the synthesis graph are eliminated. The leaves of the newly generated DAG representation of the synthesis graph gives us the list of potential safe sets: every leaf represents a single instruction, or a choice between several instructions (represented by the disjunction as a result of replacing cycles in the graph).

2) SYNTHCT safe set: Using the methodology described in the above subsection, we derive safe sets for SYNTHCT synthesized translations. Specifically: Table III represents the core safe set that must be included in all SYNTHCT safe sets. Table II shows additional safe set instructions involved in synthesis graph cycles (§VII-A1). By combining the instructions in Table III with one instruction per group in Table II, we can form one of many final safe sets. The core safe set contains 53 instructions and we need to select 1 instruction from each of the 37 groups of instructions to form a complete safe set, resulting in a total of 90 instructions in a safe set. Therefore, using only 90 / 366 (25%) of the instructions we can implement the remaining 276 (75%) instructions in the ISA.

3) Comparing to a microarchitecture safe set: The next question we want to answer is: what are the implications of these generated safe sets and how do they compare to a microarchitecture specific safe-set specification? Recall that in the second phase of SYNTHCT, we use a microarchitecture specific safe-set specification to generate safe translations of all instructions in the ISA. The goal is to find translations for every instruction in the ISA to instructions in the uarch safe set. Therefore, when given such a safe-set specification, we can have two different scenarios:

Case 1: Microarchitecture safe-set specification and SYNTHCT safe sets are compatible. In this case, at least one of the SYNTHCT-generated safe sets is a subset of the microarchitecture safe set. Since, by definition, all instructions in the ISA can be translated using only the instructions in SYNTHCT's safe set and the microarchitecture safe set is a superset of SYNTHCT's safe set, every instruction in the ISA can be translated using only the instructions in the microarchitecture safe set. Having more instructions in the microarchitecture safe set simply means more choices of instructions that may be used in the safe translations. This is a success case as now any program compiled for the given ISA can be secured by using the set of safe translations from SYNTHCT.

Case 2: Microarchitecture safe-set specification and SYNTHCT safe sets are incompatible. In this case, none of SYNTHCT's safe sets are subsets of the microarchitecture safe set. Since instructions in the SYNTHCT safe set are leaves in the synthesis graph, this means there is one or more instruction that cannot be translated to the microarchitectural safe set (namely, the SYNTHCT safe set instructions that are not contained in the microarchitectural safe set). This is a failure case. Not all instructions (and hence programs) in the ISA can be expressed using the safe set of instructions for that microarchitecture. Further synthesis tasks and expert assis-

ADCB*(2)	SUBB*(2)	SUBB*(2)	ADCQ or SBBQ	ADCL or SBBL
ADCW or SBBW	ADDB*(4)	NOTL or ANDL	ROLL or RORL	
CMPXCHGB*(4) or XCHGN*(4) or ANDB*(2) or ORB*(2) or XORB*(2)	ANDB*(2)	SALQ or SHLQ	SHLL or SALL or BLSRL	BSRL or BSFL
BSRQ or BSRW	ORB*(2)	XORB*(2)	NEGB*(2)	INCB*(2)
ROLW or RORW	RCRB or RCLB	RCLW or SHLW or SALW	ROLB*(2) or RORB*(2)	ROLL or RORL
SALB*(2) or SHLB*(2)	SALL or SHLL	SALW or SHLW	SARB*(2)	SET*(9 sets, 60 total)

TABLE II: Safe set instructions involved in cycles. One instruction from each group needs to be chosen to be included in the final safe set. '*' denotes multiple instruction variants shown in a compressed form. The number of instructions represented/compressed is indicated with parentheses. For example, the "SALB*(2) or SHLB*(2)" group contains 4 instructions; 1 of which must be included in the final safe set.

Category	Opcode	B	W	L	Q
Bitwise Operations	NOT	✓	✗	✗	✗
	AND/XOR	✗	✗	✗	✓
Divide and Multiply	DIV/IDIV	✓	✓	✗	✓
	IMUL/MUL	✓	✓	✓	✓
Bit Rotates	RCL/RCR	✓	✓	✓	✓
	ROL	✓	✗	✗	✗
	ROR	✓	✓	✗	✗
Bit Shifts	SAR/SHR	✓	✓	✓	✓
	SARX	-	-	✓	✓
	SHL	✗	✗	✗	✓
	SHLX	-	-	✓	✗
Bit Counting	TZCNT	-	✓	✗	✗
	POPCNT	-	✓	✗	✓
	LZCNT	-	✓	✗	✗
Exchange	XCHG	✗	✗	✓	✗
	CMPXCHG	✗	✓	✓	✓
Miscellaneous	BEXTR	-	-	✗	✓
	BTRW	-	✓	✗	✗
	BZHI	-	-	✗	✓
	BSWAP	-	-	✓	✓
	CMOVE	-	-	✗	✓

TABLE III: SYNTHCT core safe set. Rows represent different opcodes. Columns represent bitwidths: **B:** Byte, **W:** Word (2 Bytes), **L:** Long (4 Bytes), and **Q:** Quad Word (8 Bytes). ✓indicates that an instruction variant is a part of the safe set. ✗ denotes that the instruction variant is not a part of the safe set. ✓ denotes that the opcode is common to the LibFTFP safe set, while ✓ denotes that the instruction is absent in the LibFTFP safe set. - indicates that the instruction does not operate on the corresponding bitwidth.

add	and	cmp	imul	mov	
movabs	movsd	movsx	movsxd	movzx	mul
neg	not	or	sar	sbb	seta
setae	setbe	sete	setg	setl	setle
setne	shl	shr	sub	test	xor

TABLE IV: Safe Instruction Set from LibFTFP. Control-flow and memory instructions are excluded for clarity. Opcodes present in LibFTFP and not in SYNTHCT's safe set are highlighted in red. As the table only shows the LibFTFP safe set, opcodes safe in SYNTHCT but not in LibFTFP are not shown. Opcodes common to both LibFTFP and SYNTHCT safe sets are highlighted in green. Opcodes with partial overlap, i.e., only some variants of instruction are in SYNTHCT safe set, are highlighted in orange. Instructions in the mov* and set* family are excluded from coloring as SYNTHCT has a choice in selecting between some of these variants, but does not necessarily need to contain all. See Table III to compare the SYNTHCT safe set against the LibFTFP (this) safe set.

tance may be needed to synthesize translations for instructions that are missing in the microarchitecture safe set to make the two sets compatible.

4) Comparison to Prior Work: In this subsection, we compare the SYNTHCT safe set with safe sets from prior work, e.g., LibFTFP [7]. The list of safe instruction is directly taken from their work and reproduced in Table IV. We have excluded all control-flow and memory instructions, since those are likely to be unsafe across microarchitectures (§III). Their work does not explicitly list the different variants of instructions, i.e., the different bitwidths of operands, but rather only the opcodes. Therefore, we assume that an opcode encodes all variants of that particular instruction.

The key takeaway is that most instructions that are considered safe in SYNTHCT are also what LibFTFP (and hence experts in constant-time programming) considers safe, this corresponds to almost all instructions in Table II and all ✓ in Table III or alternatively the green + orange + black opcodes in Table IV. Notable exceptions to this trend are the instructions in the bit-counting and miscellaneous categories: these are complex x86_64 instructions for which SYNTHCT was unable for synthesize translations for certain variants of the instruction. This is likely due to strict timeout limits and imperfect component selection during our factorization strategy. We aim to improve this to shrink the safe set further in future work.

That said, SYNTHCT need not have translations for every variant of every instruction. For example, SYNTHCT's safe set only needs to include `XORQ` and 1 (out of 4) variants of `XORB` (as the other 3 variants of `XORB` map trivially to the one with a translation). Lastly, there are certain opcodes, e.g., `CMP` and `TEST`, that SYNTHCT does not include in its safe set as we were able to synthesize translations for all variants of these instructions using other instructions in the safe set.

B. Performance Evaluation

In this section, we evaluate the performance of translations synthesized by SYNTHCT. We show the distribution of lengths of synthesized safe translations in Figure 7. Note that these are lengths of programs when unsafe instructions are written completely in terms of the safe set of instructions introduced in §VII-A. We split the graph into two for clarity, one for the majority of instructions that do not use factorization, and another for instructions that need factorization for synthesis.

As we see from the graph, most instructions have translations made up of less than 6 instructions. Each instruction may have multiple translations/solutions (§VI-D1) of different lengths. We show the minimum/maximum solution lengths to showcase the solution diversity. The second graph shows

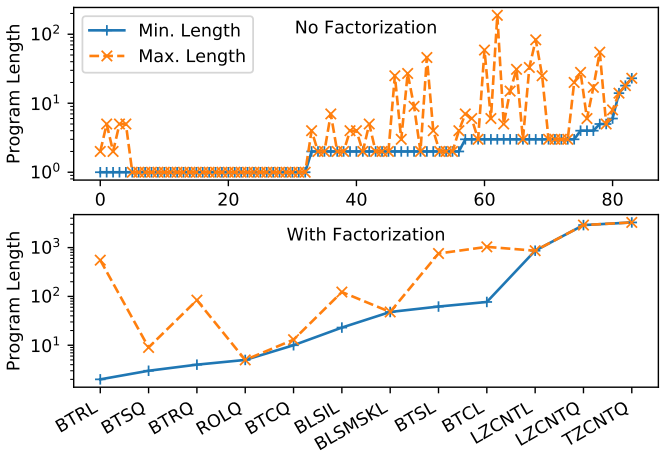

Fig. 7: Distribution of lengths of Synthesized Translations. Each x-tick denotes an index for an unsafe instruction, or a group of unsafe instructions (when they form a cycle).

the complexity of solutions that SYNTHCT can generate with factorization. The three bit-counting instructions all take 800+ instructions to implement: `LZCNTL` (864), `LZCNTQ` (2893), `TZCNTQ` (3277). This shows the strength of our factorization solution: with the help of factorization, SYNTHCT is able to scale to generate programs of length 3000+, even though the individual synthesis tasks are limited to synthesizing programs of small lengths.

C. Case-Study: Rotate Left

Now, we present a case-study on synthesis of the rotate family of instructions, specifically Rotate-Left (`ROLL`), to illustrate the effectiveness of the node-splitting strategy introduced in §VI-C. During the initial synthesis run, we were unable to get synthesis successes for rotate instructions to synthesize a translation into simpler instructions, e.g., using bitshifts and other bitwise operations. As highlighted earlier, this is primarily due to the complex and opaque K `rol` opcode used in the rotate family of instructions. Without further assistance, SYNTHCT is unable to meaningfully pick components that increase the likelihood of successful synthesis.

To assist SYNTHCT in synthesis, we introduce a new transformation in SYNTHCT to node split the `rol` intermediate opcode into simpler K opcodes. Both the original opcode and its translation is shown is Figure 8. The code snippet only shows the rotate part of `ROLL`; the instruction AST has other unrelated nodes to extract and concatenate the relevant bits of its operands (shown in the appendix, Figure 13 for reference). Note that before implementing the transform in SYNTHCT, we first manually verified the equivalence between the original opcode and the transformed AST by querying the SMT solver for equivalence through Rosette. Once node splitting is applied, the newly generated AST for `ROLL` (shown in the appendix, Figure 14) proceeds through the synthesis process as usual. As SYNTHCT now has more structural information to work with, the component selector can pick better components for synthesis. However, the new AST is larger than the original after translation. We therefore use

```
; assume %1 is already mod-ed
; %w = width of rotate rol, %0, and %1
(rol %0 %1) -> (bvor (bvlshr %0 (bvsub %w %1))
                     (bvshl %0 %1))
```

Fig. 8: Node splitting transformation rule for the ROL opcode.

```
(define (implement-DIVL-R32 S r1 r2 r3 r4 r5)
(let (
  [local-r1 (extract 31 0 (state-Rn-ref S r1))]
  [local-r2 (extract 31 0 (state-Rn-ref S r2))]
  [local-r3 (extract 31 0 (state-Rn-ref S r3))]
)
; Assume r2 is edx and r1 is eax.
; r3 is "real" argument to instruction (divisor)
; r3 outputs quotient and r4 outputs remainder
(begin
(define dividend (concat local-r2 local-r1))
(define divisor (zero-extend local-r3
                             (bitvector 64)))
; Avoid division by 0
(assume (not (bveq divisor (bv 0 64))))

(set! r3 (bv 0 64))
(set! r4 (bv 0 64))

(for ([r5 (in-range 63 -1 -1)])
  (set! r4 (bvshl r4 (bv 1 64)))
  (set! r4 (bvor r4 (bvand (bvlshr dividend
                                   (bv i 64))
                           (bv 1 64))))
  (if (bvuge r4 divisor)
    (begin
    (set! r4 (bvsub r4 divisor))
    (set! r3 (bvor r3 (bvshl (bv 1 64)
                             (bv r5 64)))))
    #f))

(state-Rn-set! S r3
               (zero-extend (extract 31 0 r3)
                            (bitvector 64)))
(state-Rn-set! S r4
               (zero-extend (extract 31 0 r4)
                            (bitvector 64)))
)))
```

Fig. 9: Long-division algorithm to node split `DIVL-R32`, implemented in racket. The algorithm is first checked for equivalence with the reference `DIVL-R32` semantics before implementing node splitting in SYNTHCT.

instruction factorization (§VI-B) to synthesize the solution for `ROLL` in parts.

To summarize, using the composition of two techniques, node splitting and factorization, along with some expert-written transformations, enables SYNTHCT to synthesize solutions for some of the more complicated instructions in the ISA. These expert-written transforms are a one time effort and scale to multiple ISAs.

D. Case-Study: Division (DIVL-R32)

In this section, we present a case study to synthesize the division instruction, specifically `DIVL-R32` — a 32-bit division from the x86_64 ISA, that is known to be *unsafe* on today's microarchitectures. Similar to the rotate instructions, we use a combination of node splitting and instruction factorization

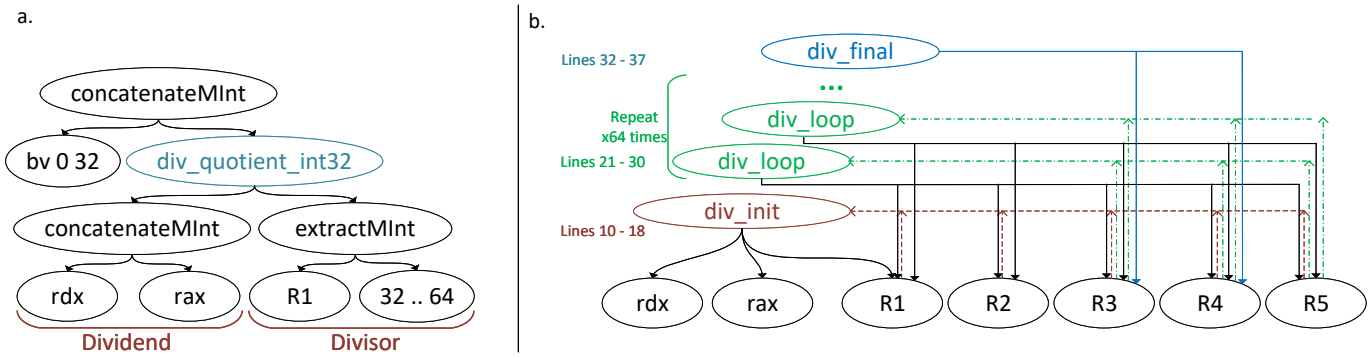

Fig. 10: `DIVL-R32` semantics AST (a) Semantics of `DIVL-R32` from the K-framework. `DIVL-R32` takes the dividend as implicit inputs in registers `rdx` and `rax` and the divisor as an explicit input in register `R1`. The complex semantics of computing the quotient (and remainder) is hidden behind the opaque K-opcode: `div_quotient_int32` (in blue). (b) Iterative node split of `DIVL-R32`. The algorithm from Figure 9 naturally splits `DIVL-R32` into three parts: (i) `div_init` (lines 10 - 18), (ii) `div_loop` (lines 21 - 30), and (iii) `div_final` (lines 32 - 37). Solid lines represent register reads by the newly split opcode and dashed lines represent register writes. The semantics of `DIVL-R32` after the split (in (b)) is equivalent to the original semantics (in (a)).

to synthesize a safe translation for division. Compared to a rotate operation, however, division has far more complex semantics that introduces new scalability challenges in our above mentioned techniques.

The semantics for the `DIVL-R32` instruction is shown in Figure 10a. As seen from the figure, the structure of `DIVL-R32` is simple. All the complexity to compute the *quotient* is hidden behind the `div_quotient_int32` K-opcode that takes two operands, the dividend and the divisor, and computes the quotient. The semantics to set the remainder is similar and omitted for brevity. In order to synthesize `DIVL-R32` we must first split the complex quotient opcode into simpler K-operations.

1) Division Algorithm: For simplicity, we use the standard long-division algorithm to compute the quotient and the remainder. The racket implementation of our division is shown in Figure 9. As with rotate, we begin by proving that our implementation of divide is semantically equivalent to `DIVL-R32` using an SMT solver. After proving equivalence, we lift the expressions for quotient and remainder directly from our racket implementation to implement node splitting in SYNTHCT.

2) Scalability Challenges: The expressions generated for the quotient (and remainder) from the long-division algorithm are very large. As shown in Figure 9 the loop body to compute the quotient and remainder is repeated `bitwidth` times (in case of `DIVL-R32`, this is 64) leading to a very large AST. While factorization can generate smaller, manageable ASTs for factors, the number of factors to synthesize makes synthesis intractable.

3) Iterative Node Splitting: Rather than splitting `div_quotient_int32` down to the simplest K-operations in one single step, we iteratively split the quotient (and remainder) K-opcode into simpler, smaller pieces in each step. Looking at the long-division algorithm, it naturally breaks into three smaller pieces: (i) An initial setup step that initializes values, (ii) a loop body that is repeated `bitwidth` (64) number of times, and (iii) a final post-loop step that extracts bits corresponding to the quotient (and remainder) and sets the output registers. This split of `div_quotient_int32` along with the relevant lines of the algorithm is shown in Figure 10b. Now, each of these splits be broken down further into simpler K-opcodes and synthesized as independent synthesis tasks, using techniques such as factorization as required. Once synthesized, the solutions to the splits are glued together to generate the translation for the original `DIVL-R32`. This process is exceedingly simple: the solutions to the three splits, i.e., `loop_init`, `loop_body`, and `loop_final`, are concatenated ensuring that the `loop_body` solution is repeated `bitwidth` (64) times. Note that rather than generating a separate split for each iteration of the loop body we can abstract away the loop iteration variable as a symbolic operand to the `loop_body` split. This allows us to synthesize `loop_body` once and instantiate with different values for `iter` (63...0) to generate loop body for the different iterations of the loop.

4) Manual Effort: During the synthesis of `DIVL-R32`, we noticed that a particular factor of `loop_body` failed to synthesize initially. On debugging, we isolated the reason for failure to be the inabilty to synthesize a factor that performed a `bvuge`, an unsigned greater-than or equal comparison, to set a register. As the x86_64 ISA does not contain an instruction to set a register based on a comparison directly (only indirectly through a `cmp` and `setcc`) our component selection failed to pick the right set of components to be able to synthesize this factor. To address the issue, we added a new pseudo-instruction to SYNTHCT, `PSETCC (r1 r2 r3 r4)`, that sets `r4` based on the result of comparison between `r2` and `r3`. The relational operator to use for comparison, e.g., equal v/s greater-than etc., is controlled by the value in register `r1`. Lastly, we manually implement `PSETCC` using a `cmp` and `setcc` from x86_64 due to limitations with the current component selection strategy.

E. Secondary Metrics

1) Effectiveness of Component Selection: In this section, we look at the effectiveness of our component selection strategy (§VI-A) and seek to verify our hypothesis: structurally

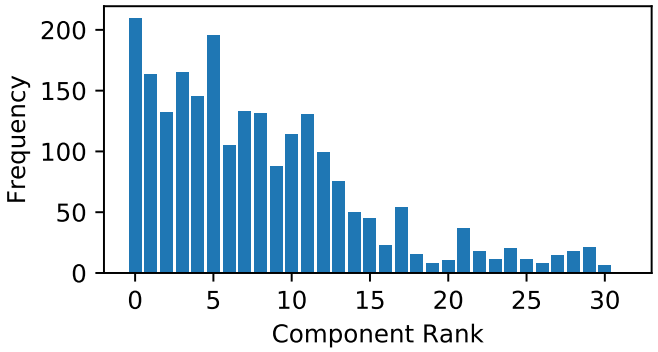

Fig. 11: Effectiveness of component selection. The histogram shows the number of synthesized translations that use a component ranked at index i (denoted on the X-axis). All synthesis tasks used a fixed set of 32 components drawn using the KNN strategy described in §VI-A. Components most similar are at index 0 while least similar component is index 31. Higher numbers for better ranked components is indicative of a good component selection strategy.

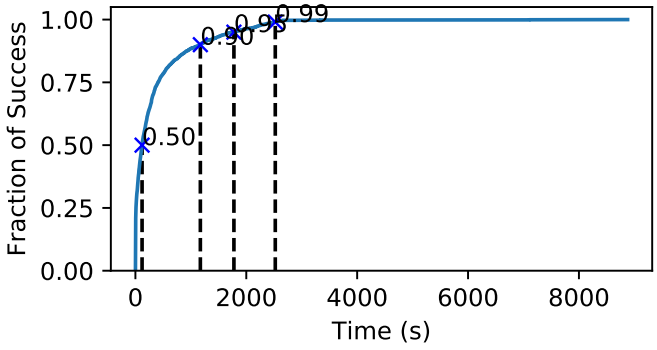

Fig. 12: Cumulative Distribution of synthesis time for successful synthesis tasks, using data from 2615 synthesis successes. The Y-axis shows the fraction of synthesis successes that take time (in seconds) shown on X-axis. 95% of synthesis successes are achieved quickly, in < 2000 seconds, showing the effectiveness of the combination of strategies developed in SYNTHCT.

similar instructions are also semantically similar, and hence more likely to be used in successful synthesis. To do so, we look at the number of times an instruction ranked as the i-th nearest neighbor by the component selection algorithm is used in the translation. The histogram plot is shown in Figure 11. Recall, for all synthesis tasks $K = 32$ components were chosen. Instructions ranked 0 are most similar to the target synthesis instruction, I_t, and instruction ranked 31 is the least similar. As we see from the histogram, instructions that are more similar to the target instruction, and hence ranked higher by the component selection strategy, are more often used in a synthesis success when compared lower ranked instructions. This plot shows the effectiveness of our component selection strategy and validates our hypothesis from earlier.

2) Synthesis Time for Successes Distribution: Lastly, we look at the time taken for synthesis successes to assess if the strategies we develop in SYNTHCT actually result in reasonable synthesis time. To do so, we look at the time taken (in seconds) for successful synthesis tasks. The cumulative distribution frequency (CDF) for synthesis time is shown in Figure 12. From this graph, we see that most successful synthesis tasks generate solutions in a short amount of time: 95% of the synthesis successes are achieved within 2000 seconds. Note that this graph shows the synthesis time for individual synthesis tasks. Instructions that need factorization run multiple smaller synthesis tasks that eventually result in successful synthesis of the original instruction. The synthesis time for such instructions is the sum of time spent in the synthesis of individual synthesis tasks. This total time is not shown in the current figure. From this, we can conclude that the combination of: (i) component selection, (ii) small program lengths, and (iii) limiting the number of registers, does indeed keep the synthesis time for individual synthesis tasks small while allowing for synthesis of translations for the majority of the ISA.

VIII. Related Work

There is a rich literature that studies how to write and run applications in constant time (sometimes called "data-oblivious programming"). For example, application-centric works propose constant-time cryptography [13], [14], machine learning [18], [41], [62], databases [19], [20], [21], memory and datastructures [23], [24], general purpose code [15], [16], [25], [63], utilities [22] and floating point functions [7]. Other works study how to write and compile (e.g., [29], [26], [27]) high-level programs to constant-time ones. ISA abstractions study how to design interfaces usable by both software designers and hardware architects to uphold constant-time security guarantees [31], [64].

SYNTHCT is complementary to these works. They all produce what we call *mostly constant-time code* (Section I) and assume a fixed set of safe instructions. SYNTHCT can be used to improve their security/portability and performance, by mapping their code to microarchitecture-specific safe sets.

The closest work to SYNTHCT conceptually is Arm's DIT [65], data-oblivious ISAs (OISAs) [31] and HW/SW contracts [66], [67]. DIT and OISAs are ISA-level specifications of what instructions are safe and therefore guarantee a consistent safe set across microarchitectures. Widespread acceptance of such abstractions across vendors would therefore decrease the need for SYNTHCT; yet, such widespread acceptance is far off because ISA changes have an extremely high barrier to entry. Case in point, there is no indication that other vendors beyond Arm will support such features. Lastly, SYNTHCT is related to HW/SW contracts for secure hardware [66], [67]: a safe set is a concrete example of such contracts (broadly construed).

There is a large body of related work that uses program synthesis to synthesize semantically equivalent alternate implementations of existing code, e.g., for better performance (superoptimizers [68]) or lower energy utilization [69]. However, none of the existing works have considered the problem of synthesizing an instruction(s) when only a subset of the ISA is allowed to be used in the synthesis. We consider

these techniques complementary. For example, it would be interesting to consider a genetic algorithm from [69] as a subroutine for synthesis instead of CEGIS as in SYNTHCT.

IX. Discussion

1) Vector and Floating-point instructions: We believe vector instructions can be supported with additional engineering effort, e.g., expanding parsing of x86 semantics and modelling vector registers in synthesis. As vector instructions are multiple instances of non-vector instructions, our techniques such as instruction factorization should reduce them to multiple simpler problems. Floating-point instructions are more challenging. In theory, using a combination of node splitting and factorization, SYNTHCT should be able generate safe translations for floating-point instructions. However, precisely modelling floating-point operations in bitvector theory in a way that performs well with SMT solvers is known to be challenging [70]. We have therefore left floating point for future work.

2) Integrating with compiler-flows: SYNTHCT is currently implemented as a post-compilation, binary tool for ease of implementation and generality. However, SYNTHCT can also be implemented at the compiler level to make it more convenient to use as a part of the compiler toolchain. Additionally, a compiler-level implementation may also generate better performing translations, e.g., by enabling optimizations and better register allocation. Implementing SYNTHCT as a part of a compiler toolchain also enables integration with security-aware DSLs designed for constant-time programming, e.g., FaCT [29], and allows us to only selectively translate *unsafe instructions* with *secret operand values* rather than all unsafe instructions.

3) Alternate deployment opportunities: SYNTHCT translations can also be deployed as a part of processor frontend or microcode in a software-transparent way. Using the translations, unsafe instructions may be selectively translated on-the-fly depending on if a security mode bit is set. This allows software to turn the protection on for only sensitive pieces of code. Yet, implementing translations in hardware is more intrusive and may introduce new tradeoffs. For example, the number of instructions making up the translation may become a first-order constraint as microcode ROM is a scarce resource.

X. Conclusion

In this work we develop SYNTHCT, a framework to facilitate writing *portable* constant-time code, i.e., that is both secure and performant across different microarchitectures. The high-order bit is that with minimal programmer intervention, we can scale to complex ISAs such as x86_64, and even tackle some of the more complex instructions in said ISAs (such as integer division).

Long term, we hope SYNTHCT (along with other spiritually-similar works [31], [67], [65]) encourages systems and hardware designers to expose abstract but explicit security-critical information in contracts such as ISAs (or their microarchitecture-specific counterparts). It is worth stating that the safe-set specification used in this work is an extremely simple, and already useful, exemplar abstraction of this kind — but is by no means the end of the story. Pushing the idea of microarchitecture-specific specifications further, one can imagine more expressive security contracts (e.g., one that specifies a partition on unsafe instruction operands, which could be used directly by SYNTHCT to improve CT performance) or even contracts geared for other metrics such as performance (e.g., more formal models of instruction interactions and their resulting pipeline throughputs).

References

[1] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*.

[2] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS '17*.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *CT-RSA'06*.

[4] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*.

[5] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*.

[6] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS'18*.

[7] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P'15*.

[8] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*.

[9] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," *IACR'18*.

[10] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security'19*.

[11] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *S&P'19*.

[12] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security'21*.

[13] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *PKC'06*.

[14] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *FSE'05*.

[15] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," *IACR'05*.

[16] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Security'15*.

[17] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional encryption using intel sgx," in *CCS'17*.

[18] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors," in *CCS'17*.

[19] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *NSDI'17*.

[20] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious Query Processing for Secure Databases," *Proceedings of the VLDB Endowment*.

[21] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An efficient oblivious search index," in *S&P'18*.

[22] S. Tople and P. Saxena, "On the trade-offs in oblivious execution techniques," in *Detection of Intrusions and Malware, and Vulnerability Assessment*.

[23] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace : Oblivious memory primitives from intel sgx," in *NDSS'18*.

[24] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx," in *NDSS'18*.

[25] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *S&P'09*.

[26] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade," *POPL'21*.

[27] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. Stefan, T. Rezk, and G. Barthe, "Towards constant-time foundations for the new spectre era," *CoRR*, vol. abs/1910.01755, 2019.

[28] J. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data," in *ISCA'21*.

[29] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Gregoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: A dsl for timing-sensitive computation," in *PLDI'19*.

[30] ARM, "Arm data indepdent timing specification," 2021. [Online; accessed 3-Jun-2021].

[31] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious isa extensions for side channel-resistant and high performance computing," in *NDSS'19*.

[32] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," *CoRR*, vol. abs/2006.03841, 2020.

[33] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *SysTEX'17*.

[34] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *MICRO'18*.

[35] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface," in *USENIX Security'20*.

[36] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled Detection of Speculative Information Flows," in *S&P'20*.

[37] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," *CoRR*, vol. abs/1807.05843, 2018.

[38] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*.

[39] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "EDDIE: EM-Based Detection of Deviations in Program Execution," in *ISCA'17*.

[40] M. Lipp, A. Kogler, D. F. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: software-based power side-channel attacks on x86," in *S&P'21*.

[41] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *USENIX Security'16*.

[42] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," S&P '15.

[43] S. E. Richardson, "Exploiting trivial and redundant computation," in *ARITH'93*.

[44] J. J. Yi and D. J. Lilja, "Improving processor performance by simplifying and bypassing trivial computations," in *ICCD'02*.

[45] E. Atoofian and A. Baniasadi, "Improving energy-efficiency by bypassing trivial computations," in *IPDPS'05*.

[46] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *ISCA'97*.

[47] A. Sodani and G. Sohi, "Understanding the differences between value prediction and instruction reuse," in *MICRO'98*.

[48] C. Molina, A. González, and J. Tubella, "Dynamic removal of redundant computations," in *ICS'99*.

[49] S. Mittal, "A survey of value prediction techniques for leveraging value locality," *CCPE'17*.

[50] C. Sakhuja, A. Subramanian, P. Joshi, A. Jain, and C. Lin, "Combining branch history and value history for improved value prediction," *CVP-Championship Value Prediction*, 2019.

[51] A. Seznec, "Exploring value prediction with the eves predictor," in *CVP-Championship Value Prediction*, 2018.

[52] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *HPCA'99*.

[53] R. Canal, A. González, and J. E. Smith, "Very low power pipelines using significance compression," in *MICRO'00*.

[54] S. Wang, J. Hu, S. G. Ziavras, and S. W. Chung, "Exploiting narrow-width values for thermal-aware register file designs," in *DATE'09*.

[55] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *PLDI'19*.

[56] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08*.

[57] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *PLDI'14*.

[58] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning distributed representations of graphs," *CoRR*, vol. abs/1707.05005, 2017.

[59] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS'13*.

[60] B. Rozemberczki, O. Kiss, and R. Sarkar, "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs," in *CIKM '20*, ACM.

[61] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, 1972.

[62] H. B. Lee, T. Jois, C. W. Fletcher, and C. A. Gunter, "DOVE: A Data-Oblivious Virtual Environment," in *NDSS'21*.

[63] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert, "Secure and private function evaluation with Intel SGX," in *SIGSAC'19*.

[64] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS'09*.

[65] "Arm architecture registers armv8, for armv8-a architecture profile." https://developer.arm.com/docs/ddi0595/latest/aarch32-system-registers/cpsr.

[66] M. Guarnieri and M. Patrignani, "Contract-aware secure compilation," *CoRR*, vol. abs/2012.14205, 2020.

[67] M. Guarnieri, B. Köpf, J. Reineke, and P. Vilar, "Hardware-Software Contracts for Secure Speculation," in *IEEE S&P 2021*.

[68] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ASPLOS'13*.

[69] E. M. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *ASPLOS'14*.

[70] E. Darulova and V. Kuncak, "Sound compilation of reals," in *POPL'14*.

APPENDIX A
EXAMPLE TRANSLATIONS

This section shows examples of synthesized translations. See Table V.

Unsafe Instruction(s)	Safe Instructions in a Translation
ADCB-RH-RH	MOVQ, SBBB-RH-RH, (SUBB-RH-RH V SUBB-RH-R8)
DECL-R32	(XORQ-R64-R64 V PXOR-R64-R64), MOVQ, (NOTL-R32 V ANDL-R32-R32)
INCL-R32	PNOT, NEGL-R32
BLSIL-R32-R32	(NOTL-R32 V ANDL-R32-R32), MOVQ, SHLQ-R64-CL, SHRQ-R64-CL, SUBQ-R64-R64, (XORQ-R64-R64 V PXOR-R64-R64), SARQ-R64-CL, NEGL-R32, PSET-FLAG
BTSL-R32-R32	MOVQ, SHLQ-R64-CL, SHRQ-R64-CL, SUBQ-R64-R64, (XORQ-R64-R64 V PXOR-R64-R64), SARQ-R64-CL, NOP, PSET-FLAG
CLTD	IMULL-R32, MOVQ
PCMOV-R64-R64-R64	CMOVEQ-R64-R64, (ANDQ-R64-R64 V PAND-R64-R64)
LZCNTQ-R64-R64	MOVQ, SHLQ-R64-CL, SHRQ-R64-CL, SUBQ-R64-R64, (XORQ-R64-R64 V PXOR-R64-R64), (ANDQ-R64-R64 V PAND-R64-R64), CMOVEQ-R64-R64, SARQ-R64-CL, PSET-FLAG
POPCNTL-R32-R32	POPCNTQ-R64-R64, (NOTL-R32 V ANDL-R32-R32)
SUBL-R32-R32	BEXTRL-R32-R32-R32, SBBL-R32-R32
XORL-R32-R32	(XORQ-R64-R64 V PXOR-R64-R64), XCHGL-R32-EAX, SARQ-R64-CL, PSPLIT
(INCW-R16 V DECW-R16 V NEGW-R16)	MOVW-R16-R16, NEGL-R32

TABLE V: Example translations for unsafe instructions. The table shows which safe instruction opcodes (§VII-A3) are used to translate several representative unsafe instructions. The "V" denotes a disjunction: These instructions form an equivalence class and we have the freedom to choose one of the instructions from the class to utilize in the final translation.

APPENDIX B
CASE STUDY: ROLL-R32-CL

This section shows the semantics AST of the original `ROLL-R32-CL` and `ROLL-R32-CL` after splitting. See Figures 13 and 14.

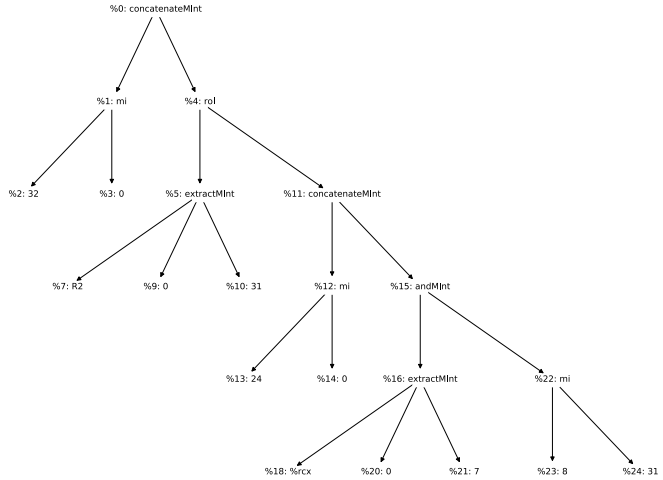

Fig. 13: The original ROLL-R32-CL instruction (used in case-study §VII-C). The AST was generated directly from $\mathbb{K}$ semantics.

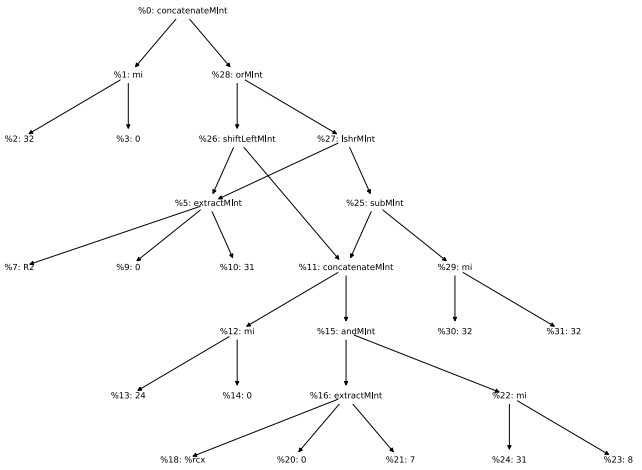

Fig. 14: The ROLL instruction after splitting the `rol` opcode. The `rol` opcode from the previous figure (Figure 13) is replaced by the equivalent translation in Figure 8, as described in the case-study (§VII-C). Notice that the node labelled as **(%4: rol)** has now been replaced by the subtree rooted by the node labelled **(%28: orMInt)**. The new AST is semantically equivalent to the old AST and is used in further synthesis steps, e.g., factorization.