# Compilation Techniques for Actively Secure Mixed Mode Two Party Computation

by

Alexander Norton

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

Waterloo, Ontario, Canada, 2019

## Abstract

Secure multiparty computation allows two or more parties to compute a function without leaking unnecessary information about their inputs to other parties. In traditional secure multiparty computation protocols, the function is represented as a circuit and each gate of the circuit is securely computed. The mixed mode model is a generalization where only some gates are computed securely, and other gates are computed in a local, unsecured manner. There are computations where mixed mode protocols are known to be just as secure and much more efficient, and so it is natural to ask if it is possible to automatically construct optimized mixed mode secure protocols for a given function.

Previous results describe powerful compilation techniques to transform circuits into efficient mixed mode protocols, but the results are only secure against very restricted (passive) adversaries. These passively secure protocols can be secured against active adversaries using extensions of classic secure multiparty computation compilation techniques. However, this comes with a significant loss of concrete efficiency, which negates the mixed mode efficiency advantages.

In this thesis, we describe novel techniques that can efficiently compile mixed mode two party protocols from passive to active security. The techniques exploit structural properties of the underlying circuits to reduce the overhead of compilation without compromising the security. The gain in efficiency varies based on the circuit that is being compiled, and although for some circuits the techniques will yield no gains, for others the resulting secure protocols have exponentially lower computation and communication cost.

## Acknowledgements

First and foremost, I thank and acknowledge my supervisor Florian Kerschbaum for the inspiration and guidance. I would also like to thank the thesis readers Urs Hengartner and Werner Dietl for their time and valuable feedback. This work was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

## Dedication

Dedicated to my family, old and new.

# Table of Contents

# List of Figures

# List of Programs

# Chapter 1

# Introduction

Secure multiparty computation (SMC) protocols allow multiple parties to securely compute a function over their combined inputs. Traditional SMC constructions [21, 7] take a circuit representation of a function and guarantee security by securely computing *each gate*. Conceptually this can be viewed as computing a circuit in a single secure block, and we call these *single mode* protocols. Mixed mode SMC is a generalization where a protocol is split over many circuits, some of which are computed securely and some of which are computed locally by each party without any secure overhead. For certain functions [12, 1, 3, 17], mixed mode implementations exist that are equally as secure and significantly more efficient than single mode implementations.

For passively secure protocols, there are known methods [12, 16] to automatically compile a functionality into an optimized mixed mode implementation. These optimized passively secure protocols can be made actively secure using standard passive-to-active compilation techniques [7, 8], but the efficiency of the original optimizations is lost [12]. Our contributions are optimizations that exploit properties of the compiled actively secure protocols to improve their efficiency without losing security. In conjunction with prior results [12, 16] this provides the first known method to automatically transform a functionality circuit into an *optimized* actively secure mixed mode protocol.

For specific functionalities, there are previously known extremely efficient actively secure mixed mode protocols, like Aggarwal et al.'s two party median protocol [1]. However, their optimizations and associated security proofs were constructed manually. Our results can be applied to generic protocols and are robust enough to *automatically recreate* Aggarwal et al.'s protocol, with the security of the resulting optimized protocol following directly from the security of each optimization.

The remainder of this thesis is structured as follows: Chapter 2 covers the background and prior work that our results build on. We begin in Section 2.1 with a description of SMT formulas, which are the building blocks of our optimizations. In Section 2.2 – 2.4 we define secure multiparty computations and the precise structure of mixed mode protocols. In Section 2.5 we describe the prior results on compiling programs to optimized passively secure mixed mode protocols. In Section 2.6 we describe how standard passive-to-active compilation techniques [7, 8] can add active security at the cost of efficiency. In Section 2.7 we note other related work. In Chapter 3 we describe an example protocol that is particularly inefficient when compiled from passive to active security, and show how our optimizations can remove this inefficiency. In Chapter 4 we describe each of our optimizations in depth, and walk through an application of them to the example protocol. In Chapter 5 we demonstrate how our optimizations recreate Aggarwal et al.'s optimized median protocol. We conclude with Chapter 6.

# Chapter 2

# Background

## 2.1 SMT Formulas

A *satisfiability formula* is a logic formula consisting of boolean variables joined by boolean gates, and a *satisfiability solver* returns whether or not there is any possible assignment to the variables in the formula that would cause the formula to evaluate to true. A *satisfiability modulo theories* (SMT) formula is a satisfiability formula where the variables are allowed to be non-boolean. The worst case computational complexity of an SMT solver depends on what types of formulas are allowed — the satisfiability of unqualified formulas over boolean variables is famously NP-Complete, whereas the satisfiability of integer polynomials (Diophantine equations) is famously undecidable. Nevertheless, solvers exist for a wide range of input types, such as Microsoft's Z3 [6]. As such throughout this work we will assume the existence of a SMT solver with the following properties, all of which Z3 satisfies:

1. Valid inputs are formulas over reals, integers or boolean typed variables, joined by standard boolean and arithmetic gates including inequalities, and with arbitrary universal and existential qualifiers.

2. For certain inputs, the solver may fail to return anything.

3. When the solver returns, it returns a correct proof of either satisfiability or unsatisfiability.

## 2.2 Secure Multiparty Computation

A secure multiparty computation protocol computes a function while securing any information about a party's input that cannot be inferred from the other parties' inputs and the output of the function, even when some other parties are controlled by an adversary. We limit our attention to the *two party* case where at most one party is adversarially controlled. We call the parties $A$ and $B$, and when referring to a generic party $p$ let $\neg p$ denote the other party.

In the *semi-honest* security model, adversarially controlled parties must still correctly follow the protocol, but may perform additional computations on the messages received in an attempt to extract additional information. In the *malicious* model, adversarially controlled parties may deviate arbitrarily from the protocol. A protocol that is secure in the semi-honest model is said to be *passively* secure, whereas one that is secure in the malicious model is said to be *actively* secure.

Security is defined in reference to an *ideal world* where both parties send their inputs to a trusted third party, who calculates the function and sends back the outputs. A protocol is secure if no information is leaked beyond what would leak in an ideal world execution. We say a protocol *securely computes* a function if for every non-uniform probabilistic polytime adversary $\mathcal{A}$, there exists a non-uniform probabilistic polytime "simulator" $\mathcal{S}$ such that regardless of which party is corrupted, the joint output of protocol execution between the honest party and $\mathcal{A}$ is computationally indistinguishable from the joint output of ideal world execution between the honest party and $\mathcal{S}$.

For both models there are many different provably secure constructions based on a variety of cryptographic assumptions [21, 7, 10, 5, 20]. Some constructions also allow for the computation of *reactive* functionalities [8, 4]. Reactive functionalities are modeled as trusted parties who are interacted with over many rounds, where some global state is stored between rounds and at each round the function computed can depend on the state and that round's inputs.

## 2.3 Computation Model

As is standard for SMC, we focus on circuit representations of programs. Our results are not limited to solely arithmetic or boolean circuits, so here we allow "circuit" to refer to *mixed arithmetic boolean circuits*.

Each circuit contains a series of wires (also referred to as variables) connected by gates. Each wire carries either a boolean or numerical value, and gates compute standard boolean and arithmetic operations, including functional blocks like comparisons. Each wire has a unique label, and wires are partitioned into *input*, *intermediate*, and *output* sets. In the SMC context, each input wire is *owned* by one party, and all other wires are owned by one or both parties.

We specify circuits as programs written in a simple pseudocode, as in Program 2.1. Programs are in single assignment form without loops or branching, exactly like a circuit. For readability we allow conditional assignment, which corresponds to an assignment that combines values masked by the conditional expression. For example, the conditional assignment of $z$ in Program 2.1 corresponds to the assignment $z = s * (x + y) + \neg s * (x - y)$, with boolean wires implicitly being converted to 0 and 1 in arithmetic gates.

**Program 2.1:** Example circuit

```
/* Inputs:  Boolean $s$, integers $w, x, y$                */
/* Outputs:  Integer $r$                                    */
$z = \begin{cases} x + y & \text{if } s \\ x - y & \text{if } \neg s \end{cases}$
$r = w * z$
```

For any circuit $C$ there is an integrity function $\mathcal{I}(C)$. Every wire in $C$ corresponds to an input wire of $\mathcal{I}(C)$, and $\mathcal{I}(C)$ outputs a single boolean checking if every wire in $C$ was calculated correctly. In order for a party to show that they computed $C$ correctly without leaking their inputs, they can use a reactive SMC protocol (or any zero knowledge proof implementation) to show that their inputs cause $\mathcal{I}(C)$ to output true.

It is straightforward to compile any circuit to its integrity function. Each assignment corresponds to an equality expression, all of which are joined together by conjunction. Conditional assignments become disjunctions with a clause for each case representing the conjunction of the assignment equality and the conditional expression. Hence for Program 2.1, the corresponding integrity function is

$$\big((s \wedge (z == x + y)) \vee (\neg s \wedge (z == x - y))\big) \wedge \big(r == w * z\big)$$

## 2.4 Mixed Mode Model

In a *mixed mode secure two party computation*, a single protocol is split into alternating rounds of secured and local (unsecured) computation. Local rounds represent computations performed locally, and parties may compute different functions in a given local round. Secure rounds represent a joint computation between parties that must be performed securely, and can be thought of as an execution of a normal SMC protocol.

We define a two party $n$-round mixed mode protocol $\Pi$ as $\{\mathcal{L}_1^A, \mathcal{L}_1^B\}$, $\mathcal{S}_1$, $\{\mathcal{L}_2^A, \mathcal{L}_2^B\}$, $\mathcal{S}_2$, $\ldots$, $\{\mathcal{L}_n^A, \mathcal{L}_n^B\}$, $\mathcal{S}_n$ , where $\mathcal{S}_i$ are circuits to be computed by SMC and $\mathcal{L}_i^p$ are circuits to be computed locally. Each input wire for all $\mathcal{L}_i^p$ must be owned by party $p$. Wire labels can be shared between rounds, in that inputs and outputs from any prior round can be used as input wires for subsequent rounds. The execution of $\Pi$ by party $p$ is the consecutive execution $\mathcal{L}_1^p$, $\mathcal{S}_1$, $\mathcal{L}_2^p$, $\mathcal{S}_2$, $\ldots$, $\mathcal{L}_n^p$, $\mathcal{S}_n$, where $\mathcal{L}_i^p$ rounds are computed solely by party $p$, and $\mathcal{S}_i$ rounds are computed jointly through SMC.

Optionally, we allow protocols to have *input requirements*, represented by preconditions that the inputs must satisfy. This is a circuit over the input set of the protocol, with a single boolean output that is true if and only if all preconditions are met. For a protocol $\Pi$ this circuit is denoted $\mathcal{P}(\Pi)$.

In the mixed mode model, we desire the overall protocol execution to be secure if *only* the secure rounds $\mathcal{S}_i$ are performed as SMC. For passively secure mixed mode protocols, one can achieve identical security to that of the single round case, as the adversary is not allowed to deviate from the protocol in any local round. However for actively secure mixed mode protocols we allow a slight weakening of the security definition. The standard security definitions [8] only allow an adversary to abort before input is sent or after output is received, whereas here an adversary can abort between rounds of the protocol. We consider selective aborts to be unavoidable in any practical definition of mixed mode active security for two parties, and instead require only that no information is leaked about the honest party's inputs beyond what would leak in a complete protocol execution.

We use an extension of the above pseudocode to specific mixed mode protocols. Inputs are specified for each party, including preconditions if present, and owners are specified for each output. Local rounds are labelled by which party runs them, and secure rounds are labelled by whether they should be passively or actively secure, and whether reactive functionality is required. Empty rounds are omitted. Our pseudocode assumes that each secure round output is owned by both parties, but our optimizations generalize to protocols where this is not the case.

Note that in Program 2.2 $\mathcal{L}_1^A$, $\mathcal{L}_1^B$, $\mathcal{L}_2^A$ and $\mathcal{S}_2$ are implicitly empty.

**Program 2.2:** Example mixed mode protocol

```
/* A's input is boolean $s$                                              */
/* B's inputs are integers $w, x, y$ s.t.  $y > x$                       */
/* Output $r$ owned by B                                                 */
passively secure round $\mathcal{S}_1$
    $z = \begin{cases} x + y & \text{if } s \\ x - y & \text{if } \neg s \end{cases}$
local round $\mathcal{L}_2^B$
    $r = w * z$
```

## 2.5 Mixed Mode Passive Security via Knowledge Inference

Suppose we have access to some passively secure SMC implementation that allows two parties to securely compute any circuit. Then the trivial "mixed mode" passively secure protocol is one with an empty local round and a single secured round that contains the entire protocol. Kerschbaum showed that it is possible to automatically split a single round protocol into a many round mixed mode protocol that is still passively secure [12]. As local rounds can be computed with no security overhead, this resulted in dramatic concrete efficiency improvements for the protocols he considered.

In essence, this optimization is based on the observation that in the semi-honest model, by definition, any information that can be inferred from computation on a party's input and output does not need to be secured by the other party. As such, it is possible to detect when intermediate wire values can always be deduced by the input and output wires a party receives, and then split the computation of these deducible values into local rounds. Program 3.3 below provides an example of this compilation technique.

Rastogi et al. formalized this optimization in terms of *knowledge inference* [16]. They define the knowledge inference problem as detecting when an intermediate variable $y$ can always be determined from input and output sets $I$ and $O$ by some function $f(I, O)$, and the *constructive* knowledge inference problem as constructing $f$ should it exist. Further, they showed that one can find explicit solutions for both problems given an SMT solver.

Using the above optimizations, one can automatically transform a two party functionality into an optimized passively secure mixed mode protocol.

## 2.6 Mixed Mode Active Security

We now describe an inefficient transformation from passive mixed mode security to active mixed mode security. The construction mimics seminal SMC techniques [7, 8] to compile a passively secure protocol into an actively secure protocol, where upfront commitments are implemented as inputs passed to a reactive functionality, and zero knowledge proofs of correctness are implemented as secure computations of the appropriate integrity functions. Our optimizations use this construction as a baseline, and identify situations where the concrete cost can be reduced without losing security.

Following other active security definitions [8], we begin by noting what an active adversary can do that a passive adversary cannot, which is:

- Abort the protocol between rounds
- Substitute inputs between rounds
- Deviate from protocol in secure round computation
- Deviate from protocol in local round computation

As noted above, we define active security in the mixed mode model to allow between-round abortion (note that abortion before or after execution, as well as input substitution before execution, are already allowed by active security definitions [8]). Hence to convert a passively secure mixed mode protocol into an actively secure one, we must secure the protocol against the remaining three types of adversarial behaviour.

We assume we have access to an actively secure SMC implementation that supports reactive functionalities. This ensures that deviations in secure rounds are detected and lead to abortion without any insecure information leakage. To handle input substitution, we require that both parties send all inputs during an additional secure round $\mathcal{C}$, at the start of the protocol. The reactive property allows each input to be sent only once and reused between rounds, and as inputs are sent before any computation happens, this ensures that no adaptive input substitution can occur. As such, $\mathcal{C}$ functions as a commitment to every party's inputs.

Finally, we require a mechanism to ensure that local rounds are always performed correctly. To do this we add additional secure *integrity* rounds $\mathcal{I}_i$ between $\mathcal{L}_i$ and $\mathcal{S}_i$. Each $\mathcal{I}_i$ checks that the previous local rounds were performed correctly by securely computing $\mathcal{I}(\mathcal{L}_i^A) \wedge \mathcal{I}(\mathcal{L}_i^B)$. If either integrity function outputs false then computation is aborted. For

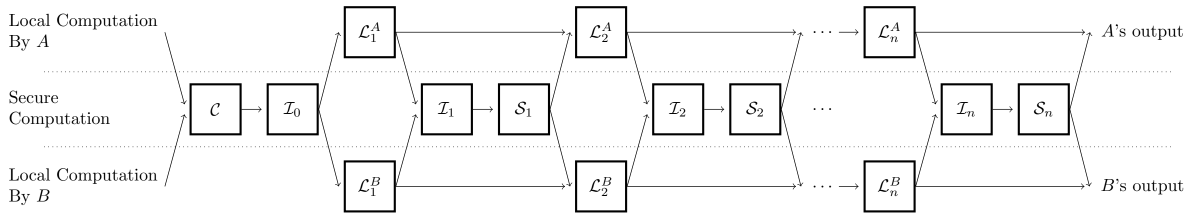

Figure 2.1: Actively Secure Mixed Mode Construction

protocols with input requirements, we add a round $\mathcal{I}_0$ after $\mathcal{C}$ that computes $\mathcal{P}(\Pi)$ and aborts on failure.

As this construction captures all types of adversarial behaviour in the malicious model, these augmentations transform any passively secure mixed mode protocol into an actively secure mixed mode protocol. [1] See Figure 2.1 for a diagram of the construction. $\mathcal{C}$, $\mathcal{L}$, $\mathcal{I}$ and $\mathcal{S}$ are commitment, local, integrity, and secure rounds respectively.

With respect to concrete efficiency we pay a heavy price when going from passive to active security. The potential of mixed mode protocols comes from "unsecuring" the local rounds, and yet each local round has a corresponding secure integrity function computation that in general requires as many gates as the local computation itself, negating mixed mode efficiency gains. However, this is a useful starting point for further optimization. Our results show that for protocols with certain properties, we can reduce the overhead of this construction without compromising security.

## 2.7 Related Work

Our work is related to mixed mode secure computations and passive-to-active security protocol transformations. Efficient *passively* secure mixed mode protocols for specific functionalities are known in a variety of domains, such as numerical problems [1], graph problems [3] and matroid problems [17]. There exist optimizing compilers to convert data queries [18] and general programs [12, 16] to passively secure mixed mode protocols (see Section 2.5), as well as many libraries to conveniently implement them [15, 9]. For *active* security, there exist relatively fewer efficient specific mixed mode protocols [1, 17]. The unoptimized passive-to-active Goldreich compiler [8] for single mode protocols can be adapted to the

[1] We focus only on deterministic functionalities here. For randomized functionalities, we augment $\mathcal{C}$ to output sufficient random bits for each party, and check that they are used correctly in the $\mathcal{I}_i$ rounds.

mixed mode case (see Section 2.6), but we know of no optimized passive-to-active mixed mode compilation results before this work.

# Chapter 3

# Motivating Example: Binary Classification Tree

To build intuition about our optimizations we will show how they apply to a classification tree protocol. In the variant we consider, $A$ has a vector of inputs of length $l$ and wants to learn the classification label for her input. $B$ has a binary tree of depth $l+1$, where internal nodes are thresholds and leaves are distinct classification labels that are public information known in advance to both parties. $B$ traverses the tree from root to leaf through up to $l$ rounds of comparison. At each round, $B$ traverses to the left or right child based on the result of comparing $A$'s current input to $B$'s current node's threshold. When a leaf is reached $B$ learns the classification result and shares it with $A$. This variant differs from other protocols in that both parties learn the result of the classification, and it is publicly known which label corresponds to which leaf in $B$'s tree. These properties are critical in enabling the following optimizations.

Although our optimizations apply to arbitrary tree shapes, for simplicity we will build an optimized secure protocol for the full depth 2 tree in Figure 3.1, beginning with the two party functionality of Program 3.1.

A secure "single mode" implementation can be constructed by transforming Program 3.1 into a circuit, as in Program 3.2.

When a binary classification circuit like Program 3.2 is securely evaluated using an SMC implementation, the number of secure gate evaluations grows *exponentially* with tree depth as circuits do not support conditional evaluation. That is, there are secure gate evaluations for each node in the tree, regardless of the path taken from root to leaf.

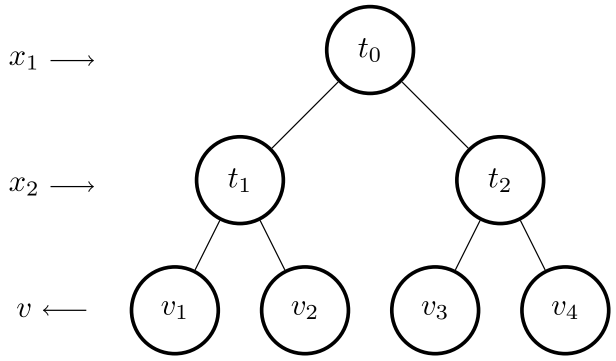

Figure 3.1: Depth 2 Binary Classification Tree

However, the previous results described in Section 2.5 can compile Program 3.1 into the secure *mixed mode* protocol Program 3.3. This compilation reveals the comparison results at each depth to both parties, which is secure because both can deduce the path taken through the tree from the public classification label they receive as output.

Program 3.3 has secure gate evaluations *linear* in the depth of tree, which is exponentially better than a single mode secure version of Program 3.2, but is only passively secure. To obtain active security, we apply the construction in Section 2.6 to create Program 3.4.

**Program 3.1:** Binary Classification Tree Functionality

```
/* A's inputs are reals $x_1, x_2$ */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$ */
/* Trusted third party $T$ computes the following */
$C_1 = x_1 \leq t_0$
if $C_1$ then
    $N = t_1$
else
    $N = t_2$
$C_2 = x_2 \leq N$
if $C_1 \wedge C_2$ then
    $v = v_1$
else if $C_1 \wedge \neg C_2$ then
    $v = v_2$
else if $\neg C_1 \wedge C_2$ then
    $v = v_3$
else
    $v = v_4$
/* T sends $v$ to $A$ and $B$ */
```

**Program 3.2:** Binary Classification Tree Secure Circuit

```
/* A's inputs are reals $x_1, x_2$ */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$ */
/* Output $v$ owned by both parties */
secure round
    $C_1 = x_1 \leq t_0$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
    $C_2 = x_2 \leq N$
    $v = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
```

**Program 3.3:** Passively Secure Binary Classification Tree Protocol

```
/* A's inputs are reals $x_1, x_2$                                  */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$          */
/* Output $v$ owned by both parties                                 */
passively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
passively secure round $\mathcal{S}_2$
    $C_2 = x_2 \leq N$
local round $\mathcal{L}_3^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
passively secure round $\mathcal{S}_3$
    /* B shares $R$ with A                                          */
    $v = R$
```

**Program 3.4:** Actively Secure Binary Classification Tree Protocol (Unoptimized)

```
/* A's inputs are reals $x_1, x_2$                                    */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$          */
/* Output $v$ owned by both parties                                   */
reactive actively secure round $\mathcal{C}$
    $A$ passes inputs $x_1, x_2$
    $B$ passes inputs $t_0, t_1, t_2, v_1, v_2, v_3, v_4$
reactive actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \le t_0$
local round $\mathcal{L}_2^B$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    abort if not $(C_1 \wedge (N == t_1)) \vee (\neg C_1 \wedge (N == t_2))$
reactive actively secure round $\mathcal{S}_2$
    $C_2 = x_2 \le N$
local round $\mathcal{L}_3^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{I}_3$
    abort if not $\begin{array}{ll} (C_1 \wedge C_2 \wedge (R == v_1)) & \vee \\ (C_1 \wedge \neg C_2 \wedge (R == v_2)) & \vee \\ (\neg C_1 \wedge C_2 \wedge (R == v_3)) & \vee \\ (\neg C_1 \wedge \neg C_2 \wedge (R == v_4)) & \end{array}$
reactive actively secure round $\mathcal{S}_3$
    /* B shares $R$ with A                                            */
    $v = R$
```

Although Program 3.4 is actively secure, each node of the tree corresponds to a variable sent to $\mathcal{C}$ and a secure gate evaluation in an $\mathcal{I}_i$ round. The number of nodes grows exponentially as tree depth increases, and so the corresponding secure computation cost is exponential in the tree depth, negating the savings of Program 3.3. As a motivating example of our optimizations, we will demonstrate how they can remove this overhead without losing active security. For each optimization, we examine the actively secure protocol and explain where its inefficiencies are. We then formally define how a compiler could perform the optimization, and compile the actively secure protocol into a more efficient version.

While single mode actively secure implementations like Program 3.2 require the total number of secure gate operations to grow *exponentially* as the depth of the input tree increases, our fully optimized protocols have secure gate evaluations *linear* in tree depth, which corresponds to an exponential decrease in computation and communication complexity. We implemented the single mode protocols and our optimized mixed mode protocols using the actively secure authenticated garbled circuit scheme [20] provided in the EMP-toolkit [19] library.[1] The concrete runtime improvements are shown in Figure 3.2.

The mixed mode protocols have a linear increase in the number of rounds of communication, and in higher latency environments this has an associated runtime cost. Despite this, the asymptotic improvement results in concretely faster protocols for higher tree depths, as Figure 3.3 demonstrates.

We stress that these improvements can be deduced automatically using our compilation techniques, and that no additional security proof is required for the resulting protocols.

[1]Experiments were performed on a single machine running 64-bit Ubuntu 16.04.5 LTS with 16 GB of RAM and an Intel Core i7-3517U processor. Networking was simulated, and each data point is the mean of 15 trials.

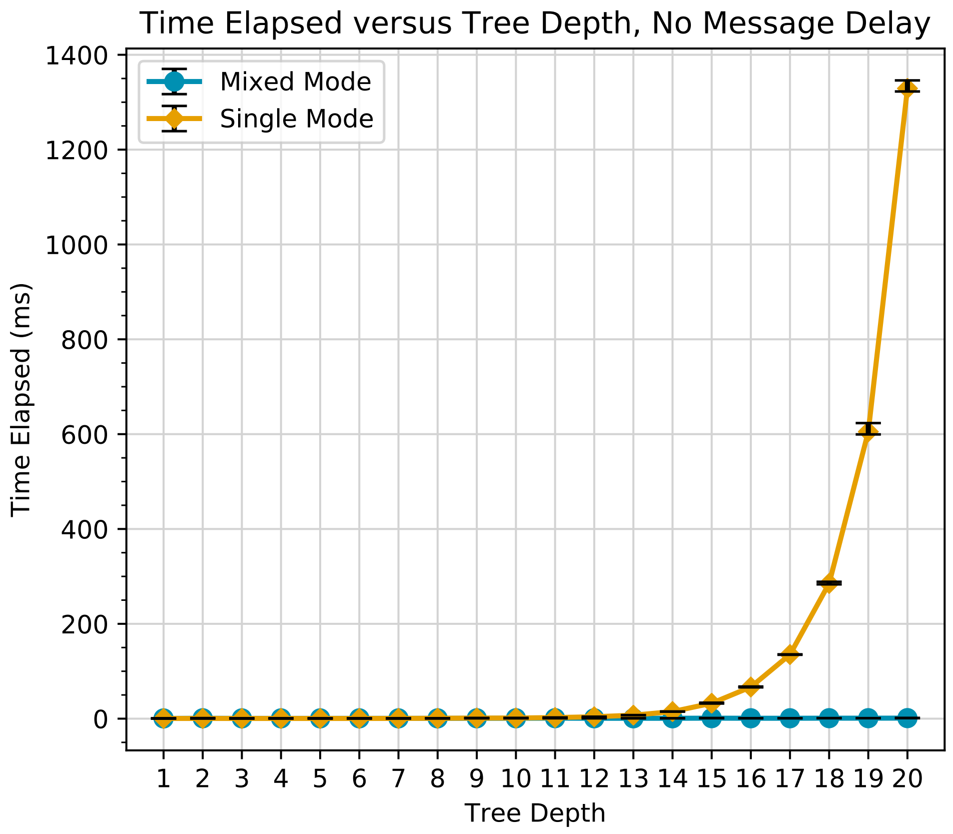

Figure 3.2: Secure Binary Classification Online Phase Time (No Message Delay)

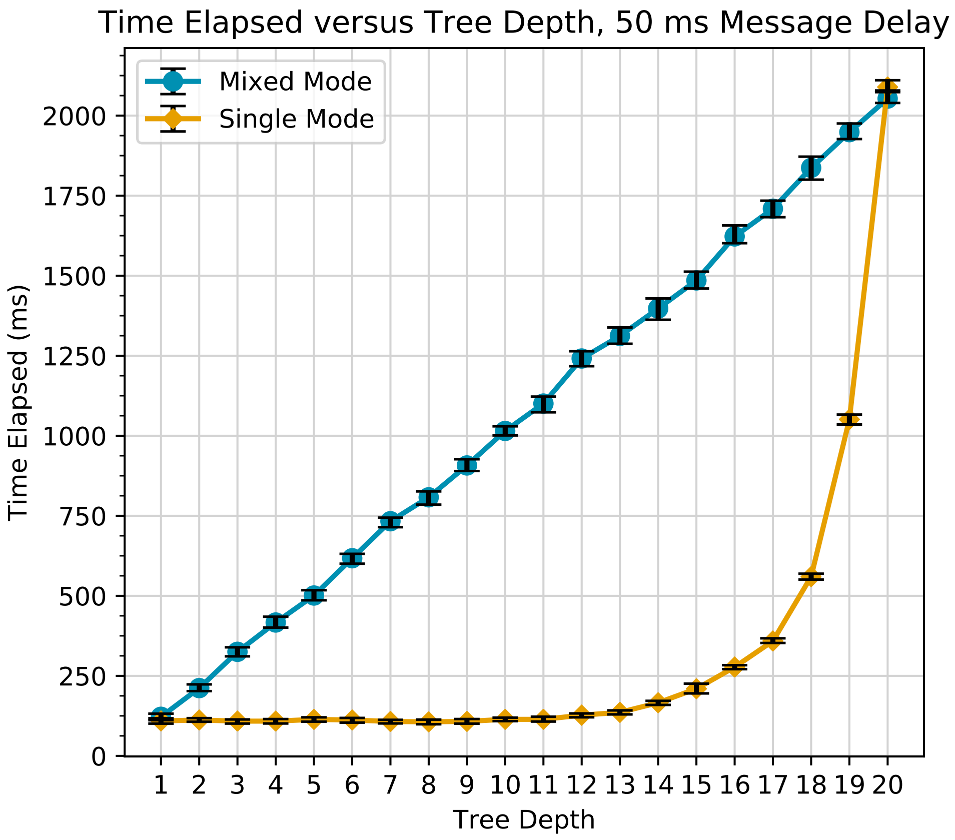

Figure 3.3: Secure Binary Classification Online Phase Time (With Message Delay)

# Chapter 4

# Optimization Techniques

In general, assuming we choose the most efficient reactive actively secure SMC implementation for the secure rounds of a given protocol, the optimizable overhead of the active construction in Section 2.6 comes from the secure round $\mathcal{C}$ and the complexity of the integrity rounds $\mathcal{I}_i$. For $\mathcal{C}$, although the exact overhead will depend on the implementation, we suppose that each input passed into $\mathcal{C}$ has some cost as secure state must be established. For protocols where not all inputs are relevant to every execution, we will show circumstances where this can be avoided. Although standard circuit optimization methods are applicable to the $\mathcal{I}_i$ circuits, we will also identify situations where inequivalent simpler circuits provide the same level of security.

We formalize three specific optimizations based on SMT formulas, such that any compiler with an SMT solver can automatically optimize an actively secure implementation of a protocol. We precisely describe an algorithm for each optimization, though we leave implementing the algorithms in a compiler to future work. In Section 4.1 we show how the information flow from secure rounds can simplify the integrity functions in later rounds. In Section 4.2, we show how inputs can be removed from $\mathcal{C}$, reducing the cost of establishing secure state. In Section 4.3 we show how these removed inputs can enable integrity functions to be further simplified.

## 4.1 Known Boolean Splitting

In this optimization, we will show how the flow of information about boolean variables in a protocol can simplify integrity round circuits. For example, consider Program 3.4. $B$'s integrity function for the first non-empty local round is

$$\mathcal{I}(\mathcal{L}_2^B) = (C_1 \wedge (N == t_1)) \vee (\neg C_1 \wedge (N == t_2))$$

This captures two different computations of the next threshold $N$ that occur depending on the value of $C_1$, and can be viewed as a proof that $B$ either performed the "$C_1$ is true" computation correctly or the "$C_1$ is false" computation correctly, without leaking which computation occurred. However, $A$ also receives the value of $C_1$ during $\mathcal{S}_1$, and hence already knows which of the computations should have been performed. Therefore intuitively it is sufficient to check that $N == t_1$ if $C_1$ is true, and that $N == t_2$ if $C_2$ is false. This implies that rather than securely computing $\mathcal{I}(\mathcal{L}_2^B)$ it is secure to compute something simpler, given that the value of $C_1$ is known to $A$.

We generalize and formalize this intuition as follows: Suppose for party $p$ at some round $i$, $\mathcal{I}(\mathcal{L}_i^p)$ contains a boolean variable $x$. Let $\mathcal{I}(\mathcal{L}_i^p)\langle x := \top\rangle$ and $\mathcal{I}(\mathcal{L}_i^p)\langle x := \bot\rangle$ be $\mathcal{I}(\mathcal{L}_i^p)$ with $x$ substituted with true and false respectively. If we can guarantee that $\neg p$ *knows* $x$ before execution of $\mathcal{S}_i$, then we can secure $\mathcal{L}_i^p$ in $\mathcal{I}_i$ by computing one of the substituted circuits, rather than $\mathcal{I}(\mathcal{L}_i^p)$.

Let the *known boolean split* for a local round $\mathcal{L}_i^p$ be the tuple $(x, f, \mathcal{I}_\top, \mathcal{I}_\bot)$, where $x$ is a boolean variable in $\mathcal{I}(\mathcal{L}_i^p)$, $f$ is a circuit computing $x$ using variables contained in $\neg p$'s subprotocol $\mathcal{C}, \ldots, \mathcal{L}_i^{\neg p}$, and $\mathcal{I}_\top$ and $\mathcal{I}_\bot$ are (potentially optimized) circuits that compute $\mathcal{I}(\mathcal{L}_i^p)\langle x := \top\rangle$ and $\mathcal{I}(\mathcal{L}_i^p)\langle x := \bot\rangle$ respectively.

Given a known boolean split, we optimize our actively secure construction as follows: Instead of calculating $\mathcal{I}(\mathcal{L}_i^p)$ in $\mathcal{I}_i$, $A$ and $B$ prepare two versions of $\mathcal{I}_i$, one that computes $\mathcal{I}_\top$ for when $x$ is true and another that computes $\mathcal{I}_\bot$ for when $x$ is false. During protocol execution, both parties calculate $x$ ($\neg p$ uses $f$ and $x$ is always known to $p$) and continue to the appropriate secure round. If parties disagree on $x$ then the protocol aborts.

### 4.1.1 Compilation Algorithm

There are three parts required to implement this optimization: determining if a variable is known by both parties, simplifying the substituted circuits, and ensuring both parties know which circuit to use at runtime.

The simplest heuristic for known booleans is to consider a boolean wire known if both parties own it. However, consider the case where $A$ owns a boolean $x$ and $B$ owns some boolean $y = \neg x$. B doesn't own $x$, but B clearly knows its value as $B$ can calculate $x = \neg y$. Hence to be complete, we should consider a boolean variable known if parties can calculate it as a function of the variables that they own. This problem was studied under the name *constructive knowledge inference* [16], where it was shown that given a two party protocol, one can use SMT solvers to determine if a party can construct a formula for a variable they do not own out of variables they do own.

To construct $\mathcal{I}_\top$ and $\mathcal{I}_\bot$, we substitute $x$ with the values true and false respectively and use circuit simplification techniques like those built into Z3 [6]. For programs where the resulting $\mathcal{I}$ circuits are purely boolean, standard boolean algebra techniques apply [11, 14].

To store each circuit for runtime lookup, we create two tables. $X$ is indexed by *names* of variables and stores how each party can compute that variable's value at runtime. $T$ is indexed by *values* of variables and stores which integrity circuit should be used for the given runtime values.

Specifically, we describe the following procedure for each local round $\mathcal{L}_i$:

1. Create tables $X$ and $T$. $X$ is initially empty and has columns labelled $A$ and $B$ that for each known boolean will track how that party can calculate it at runtime. $T$ tracks the appropriate integrity function for each party given known boolean values, and is initialized with $\mathcal{I}(\mathcal{L}_i^A)$ in a column labelled $A$ and $\mathcal{I}(\mathcal{L}_i^B)$ in a column labelled $B$.

2. For each party $p$ and each boolean variable $x$ in $\mathcal{I}(\mathcal{L}_i^p)$ that is not already in $X$, we must determine if $x$ is known by $\neg p$.

   If $x$ is owned by $\neg p$, set $f = x$. Otherwise, we use a constructive knowledge inference procedure [16]. The protocol we use as input is the subprotocol for $\neg p$ up to $\mathcal{I}_i$, where each wire owned by $\neg p$ is considered an output. Specifically, this is $\mathcal{C}, \mathcal{I}_0, \mathcal{L}_1^{\neg p}, \mathcal{I}_1, \mathcal{S}_1, \ldots, \mathcal{L}_i^{\neg p}$. Set $f$ to whatever the constructive knowledge inference procedure returns. If $f$ is empty (because $x$ is not knowable by $\neg p$, or because of failures in the underlying SMT solver calls), proceed to the next variable.

   If $f$ is not empty, add a row for $x$ to $X$, with the trivial circuit $x$ for $p$ and circuit $f$ for $\neg p$, and continue to the next step.

3. Add a new column to $T$ labelled $x$. Each row $R$ in $T$ has a circuit $\mathcal{I}^A$ and $\mathcal{I}^B$. Compute $\mathcal{I}_\top^A$ and $\mathcal{I}_\bot^A$ by simplifying $\mathcal{I}^A\langle x := \top\rangle$ and $\mathcal{I}^A\langle x := \bot\rangle$ respectively, and compute $\mathcal{I}_\top^B$ and $\mathcal{I}_\bot^B$ similarly. Replace $R$ with two rows, one with entries $\top$ in column $x$, $\mathcal{I}_\top^A$ in column $A$ and $\mathcal{I}_\top^B$ in column $B$, and one with entries $\bot$ in column $x$, $\mathcal{I}_\bot^A$ in column $A$ and $\mathcal{I}_\bot^B$ in column $B$.

4. Using $X$ and $T$, make the following modifications to the protocol. For each party $p$ and variable $x$ with a non-trivial $f$ in $X$, add an assignment to $\mathcal{L}_i$ where $x$ is computed using $f$. For each row in $T$, replace the normal $\mathcal{I}_i = \mathcal{I}(\mathcal{L}_i^A) \wedge \mathcal{I}(\mathcal{L}_i^B)$ with a new $\mathcal{I}_i = \mathcal{I}^A \wedge \mathcal{I}^B$. At runtime, both parties use their values of the variables in $X$ to branch the appropriate $\mathcal{I}_i$.

### 4.1.2 Example Compilation

We demonstrate this optimization on Program 3.4, beginning with $\mathcal{L}_2$ as $\mathcal{L}_1$ is empty. $T$ is initialized to

| $A$ | $B$ |
|---|---|
| $\top$ | $(C_1 \wedge (N == t_1)) \vee (\neg C_1 \wedge (N == t_2))$ |

As $A$ has no local rounds in this protocol $A$'s integrity function is empty, i.e. it always evaluates to true. $\mathcal{I}(\mathcal{L}_2^B)$ contains a single boolean variable $C_1$. It is trivially known to $A$ as it is also owned by $A$, so $X$ is updated with the trivial functions

| Variable | $A$ | $B$ |
|---|---|---|
| $C_1$ | $C_1$ | $C_1$ |

$T$ currently has one row. The first function simplification is

$$\begin{aligned}
\mathcal{I}(\mathcal{L}_2^B) &= (C_1 \wedge (N == t_1)) \vee (\neg C_1 \wedge (N == t_2)) \\
\mathcal{I}(\mathcal{L}_2^B)\langle C_1 \coloneqq \top\rangle &= (\top \wedge (N == t_1)) \vee (\neg \top \wedge (N == t_2)) \\
&= (\top \wedge (N == t_1)) \vee (\bot \wedge (N == t_2)) \\
&= (\top \wedge (N == t_1)) \vee \bot \\
&= \top \wedge (N == t_1) \\
\mathcal{I}_\top^B &= N == t_1
\end{aligned}$$

which matches our intuition for what should be checked when $C_1$ is true. The calculation of $\mathcal{I}_\bot^B = N == t_2$ follows symmetrically. Hence $T$ is updated to

| $C_1$ | $A$ | $B$ |
|---|---|---|
| $\top$ | $\top$ | $N == t_1$ |
| $\bot$ | $\top$ | $N == t_2$ |

$C_1$ is the only boolean variable and so this is the final value of $T$ for $\mathcal{L}_2$.

For $\mathcal{L}_3$, $\mathcal{I}(\mathcal{L}_3^B)$ contains the boolean variables $C_1$ and $C_2$. The optimization proceeds symmetrically for $C_1$, resulting in the following table for $T$:

| $C_1$ | $A$ | $B$ |
|---|---|---|
| $\top$ | $\top$ | $(C_2 \wedge (R == v_1)) \vee (\neg C_2 \wedge (R == v_2))$ |
| $\bot$ | $\top$ | $(C_2 \wedge (R == v_3)) \vee (\neg C_2 \wedge (R == v_4))$ |

The remaining boolean variable $C_2$ is also trivially known to both parties, and so $X$ is updated to

| Variable | $A$ | $B$ |
|---|---|---|
| $C_1$ | $C_1$ | $C_1$ |
| $C_2$ | $C_2$ | $C_2$ |

and after routine simplification the final value of $T$ is

| $C_2$ | $C_1$ | $A$ | $B$ |
|---|---|---|---|
| $\top$ | $\top$ | $\top$ | $R == v_1$ |
| $\top$ | $\bot$ | $\top$ | $R == v_2$ |
| $\bot$ | $\top$ | $\top$ | $R == v_3$ |
| $\bot$ | $\bot$ | $\top$ | $R == v_4$ |

Using the above tables, Program 3.4 is optimized into Program 4.1. The number of secure gate operations in the integrity rounds has been significantly reduced, but the amount of secure state established by $\mathcal{C}$ is still exponential in the depth of the tree. Next we will show how this too can be optimized.

## 4.2 Removing Initial Input Commitments

In this optimization we show how to detect circumstances where providing inputs to $\mathcal{C}$ adds no extra security. When a party passes an input to $\mathcal{C}$, it functions as a *commitment* to that input, which has a concrete cost as some secure state must be established by the reactive SMC implementation. If an input will be used in a later secure round then there

**Program 4.1:** Actively Secure Binary Classification Tree Protocol
(After known boolean splitting)

```
/* A's inputs are reals $x_1, x_2$                                */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$        */
/* Output $v$ owned by both parties                                */
reactive actively secure round $\mathcal{C}$
    $A$ passes inputs $x_1, x_2$
    $B$ passes inputs $t_0, t_1, t_2, v_1, v_2, v_3, v_4$
reactive actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    $\begin{cases} \text{abort if not } (N == t_1) & \text{if } C_1 \\ \text{abort if not } (N == t_2) & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{S}_2$
    $C_2 = x_2 \leq N$
local round $\mathcal{L}_3^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{I}_3$
    $\begin{cases} \text{abort if not } (R == v_1) & \text{if } C_1 \wedge C_2 \\ \text{abort if not } (R == v_2) & \text{if } C_1 \wedge \neg C_2 \\ \text{abort if not } (R == v_3) & \text{if } \neg C_1 \wedge C_2 \\ \text{abort if not } (R == v_4) & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{S}_3$
    /* B shares $R$ with A                                         */
    $v = R$
```

is no additional cost to providing it earlier in $\mathcal{C}$, but in certain protocols not every input is used in every execution.

For example, consider the depth 2 binary classification tree in Figure 3.1. If $x_1 \leq t_0$, then execution traverses to the left, and so the entire right subtree of $t_2$, $v_3$, and $v_4$ will never be used in the execution. Furthermore, in Program 4.1, these values are passed to $\mathcal{C}$ without ever being used in a subsequent $\mathcal{S}$ or $\mathcal{I}$ round. Ideally, there would be a way to remove these commitments without compromising security.

Generally speaking, commitments are required for the security of a protocol. In our example, if $B$ does not provide $t_1$ and $t_2$ in advance, then a corrupted $B$ could choose $t_1$ and $t_2$ after it already knows $C_1$, which manipulates the next threshold value $N$. That is, a corrupted $B$ could choose $t_1$ and $t_2$ such that $N$ equals *any function* of $C_1$, which is adversarial behaviour. However, our binary classification protocol has an interesting property. Before any computation occurs, $B$ can *already* manipulate $N$ to be any function of $C_1$. To see this, consider that any function $f$ from boolean to real is defined by two values $f(\top)$ and $f(\bot)$. As $N$ is set to $t_1$ when $C_1$ is true and $t_2$ when $C_2$ is false, by setting $t_1 = f(\top)$ and $t_2 = f(\bot)$, $B$ can ensure $N = f(C_1)$ *before* learning $C_1$'s value. Hence no behaviour is prevented by requiring $B$ to commit to $t_1$ and $t_2$ in $S_0$, and so intuitively these commitments are not required for security. Similarly, in $\mathcal{L}_3$, $B$ can set $R = f(C_1, C_2)$ for any $f$ by setting $v_1 = f(\top, \top)$, $v_2 = f(\top, \bot)$, $v_3 = f(\bot, \top)$, and $v_4 = f(\bot, \bot)$, and so no behaviour is prevented by requiring commitments to $v_1$, $v_2$, $v_3$, and $v_4$.

We call variables *free* when they are not passed to $\mathcal{C}$. To prove that a protocol with free variables is secure, we can construct a simulator between the “ideal world” where all variables are sent in $\mathcal{C}$ and the “real world” where free variables are sent only when needed. Standard hybrid arguments [8] allow us to model each reactive SMC round as interactions with a trusted third party $T$. As is standard in simulation arguments [13] it is sufficient to suppose one party is controlled by some deterministic adversary in the real world, and consider how to simulate that adversary's behaviour in the ideal world.

We can automatically determine if all adversaries are simulatable by building an SMT formula that represents simulatability. In our example, the resulting formula captures the idea that there is a single setting of $t_1$ such that for all possible branches of computation, the computation of all secure rounds will have the same value in both worlds. The satisfiability of the formula implies that a simulator exists, which proves that it is secure to remove the commitment to $t_1$. We can iteratively construct these formulas to test whether it is secure to free each of the input variables in Program 4.1. As all are satisfiable, it is secure to remove all commitments from $\mathcal{C}$ for this protocol.

### 4.2.1 Compilation Algorithm

In our simulator arguments, we treat each secure round in the protocol as an interaction between $A$, $B$ and a trusted third party $T$. $T$ receives inputs from $A$ and $B$ and sends them the output of the secure round. $T$ is reactive, that is, they store the variables they receive between rounds. Hence, each variable is passed to $T$ only once in a given protocol execution. $T$ only receives variables that are owned by a single party (either that party's inputs, or variables that are computed in local rounds) as the variables that are owned by both parties must have been computed by $T$ in a secure round.

Note that the interactions between $T$ and the other parties may depend on the runtime values of specific variables. This can happen as a result of known boolean splitting, which can affect which variables are sent to $T$ in a given secure round, or due to an adversarially controlled party choosing an input variable as a function of values received at runtime. Hence different runtime values of certain variables will cause different *branches* of computation to occur. Each branch of computation is uniquely determined by a *context* $C$, which is a set of assignments to boolean variables.

Note that $C$ contains only boolean variables. If an adversary receives an integer or a real valued $x$, and is then allowed to choose the value of an input variable $y$, then this cannot be secure. Intuitively, there are infinitely many possible values of $x$ and so an adversary can choose infinitely many values for $y$, and no simulator can simulate all of these cases. Hence $y$ must be committed to before $x$ is received. We will capture this in our procedure by failing if an input variable is provided to the adversarial party after a non-boolean output has been received from $T$.

We now discuss the mechanics of simulating an adversarially controlled party $p$. Suppose we have two protocols $P$ and $Q$, where $P$ is known to be secure and $Q$ is of questionable security. We require that $P$ and $Q$ differ only in which variables are passed to $\mathcal{C}$, that is, which variables require initial input commitments. In essence, we will consider $P$ and $Q$ to be run in two "worlds". Following standard nomenclature, we refer to $P$'s world as the *ideal* world as $P$ is known to be secure, and $Q$'s world as the *real* world. We will use simulatability to investigate whether or not $Q$ is also secure.

A simulator $S$ for a party $p$ performs the role of $p$ in an execution of $P$. That is, $S$ sends inputs to a trusted third party $T$ and receives outputs as if $S$ were $p$ itself. There is an honest $\neg p$ also interacting with $T$, but those interactions are invisible to $S$. As part of its definition, $S$ can interact with $p$ in an execution of $Q$. That is, for any secure round, $S$ can receive inputs from $p$ as if it were $T$, and send outputs to $p$ to see how it behaves. By interacting with $p$ in fake executions of $Q$, $S$ aims to ensure that any execution of $P$ with

$S$ and $\neg p$ will have *identical output in each secure round* to any execution of $Q$ with $p$ and $\neg p$. If this is possible, then $Q$ is secure against an adversarially controlled $p$, as there is nothing $p$ can do in $Q$ that cannot be done in the guaranteed secure protocol $P$.

The overall structure of a simulatability capturing SMT formula comes from the following intuition:

- $p$'s interactions in $Q$ can be simulated in $P$ if $p$ is simulatable in all possible branches of $P$ and $Q$.
- A single branch is simulatable if there is identical output between all secure rounds.
- Two $\mathcal{S}_i$ rounds are equivalent if all outputs are equal in both protocols and all outputs are computed following the protocol, that is, the inputs and outputs of the round satisfy the round's integrity function.
- Two $\mathcal{I}_i$ rounds are equivalent if both output true.

Hence we can build an SMT formula representing the simulatability of a protocol by building a formula for each branch consisting of formulas for each round, and taking the conjunction over all branch formulas. The variables in the formula have the following structure:

- The honest party $\neg p$ has consistent inputs that it uses between both protocols. There is one formula variable for each input.
- The adversarial party $p$'s inputs are only consistent given the context $C$, as it can choose different values for inputs based on the assignments in the context. Hence for each input there is a formula variable corresponding to each context $C$.
- As the secure round outputs can depend on the adversarially chosen inputs, each secure round output has a formula variable for each context $C$.
- All of the above variables are universally quantified as they are provided by parties besides $S$.
- There is one formula variable for each input $S$ passes to $T$. These are existentially quantified, as $S$ must choose one assignment to each that satisfies any value provided by the other parties.

The above does not discuss how to simulate an adversary that aborts or passes inputs that cause integrity rounds to fail. Although we omit these cases in our procedure below, both behaviours can be covered with simple modifications. Aborts are captured by a boolean variable for each round that specifies whether the adversary aborts. Failures can be handled by extending the definition of round equivalence to capture that two executions must fail at the same round. That is, two branches are equivalent not only if they have identical output at every round, but also if both fail at the first round, or are identical in the first round but both fail at the second round, etc. Each of these cases can be captured by a formula and each formula can be joined by disjunction, but this complicates the formulas significantly and detracts from the intuition behind the process. As such, we limit our procedure to capture the simulatability of adversaries that do not cause a protocol to abort early. A procedure that handles aborting adversaries can be found in Appendix A.

We provide the following procedure to create an SMT formula. The formula has three parts. $U$ contains universally quantified variables, $E$ contains existentially quantified variables, and $F$ is a boolean valued circuit over these variables. This represents a formula with the form "for all assignments to variables in $U$, does there exist an assignment to variables in $E$ such that $F$ evaluates to true"? The procedure has a recursive inner part, which can branch into many different calls that return different values for $U$, $E$, and $F$. These will be merged into one formula by the outer procedure, which has inputs $p$, $P$, and $Q$. The inner procedure has the following inputs:

- $U$, the set of universally quantified variables.
- $E$, the set of existentially quantified variables.
- $F^Q$, the formula representing the successful completion of $p$'s execution of $Q$
- $F^P$, the formula representing the successful completion of $S$'s execution of $P$
- $C$, a set of assignments to boolean variables that occur throughout a branch of execution.
- $n$, a boolean flag which tracks whether a non-boolean output has been received.
- $S$, which tracks the current secure round in both protocols.

**create_formula**$(p, P, Q)$ is defined as:

1. Call **create_formula'**($\{\}, \{\}, \top, \top, \{\}, \bot, \mathcal{C}$), and aggregate each returned $(U', E', F')$ into sets $(U^*, E^*, F^*)$. (If any call returns $\bot$, abort procedure.)

2. Let $U = \bigcup_{U' \in U^*} U'$.

3. Let $E = \bigcup_{E' \in E^*} E'$.

4. Let $F = \bigwedge_{F' \in F^*} F'$.

5. Return $(U, E, F)$.

where **create_formula'**$(U, E, F^Q, F^P, C, n, S)$ is:

1. If $S = \emptyset$, return $(U, E, (F^Q \Rightarrow F^P))$.

2. Let $I_p^P$ be the set of variables used in round $S$ in $P$ that are owned solely by $p$ and are not already present in $E$. If $n$ is true and $I_p^P$ is non-empty, return $\bot$. Otherwise, for each $x$ in $I_p^P$, add a new variable $x'^C$ in $E$.

3. Let $I_p^Q$ be the set of variables used in round $S$ in $Q$ that are owned solely by $p$ and are not already present in $U$. For each $x$ in $I_p^Q$, add a new variable $x^C$ in $U$.

4. Let $I_{\neg p}$ be the set of variables used in round $S$ in either $P$ or $Q$ that are owned solely by $\neg p$ and are not already present in $U$. For each $x$ in $I_{\neg p}$, add a new variable $x$ to $U$.

5. If $S$ is a secure round $\mathcal{S}_i$:

   (a) By our restrictions on the forms of $P$ and $Q$, $S$ has the same outputs in both protocols. Let $\mathcal{S}$ be the circuit computed by $S$, and let $O$ be the set of its outputs.

   (b) For every variable $x$ in $O$, add a variable $x^C$ to $U$.

   (c) Let $O_p$ be the subset of $O$ that is owned by $p$. If $O_p$ contains any non-boolean variables, set $n$ to $\top$.

   (d) Let $\mathcal{I}^P$ and $\mathcal{I}^Q$ be the integrity formula $\mathcal{I}(\mathcal{S})$.

6. If $S$ is a secure round $\mathcal{I}_i$:

   (a) Let $\mathcal{I}^P$ be the circuit computed by $\mathcal{I}$ in $P$ and $\mathcal{I}^Q$ be the circuit computed by $\mathcal{I}$ in $Q$.

7. In $\mathcal{I}^P$, every variable $x$ that is solely owned by $p$ exists in $E$ with the form $x'^C$. Replace all such $x$ with the matching $x'^C$ form in $\mathcal{I}^P$.

8. In $\mathcal{I}^Q$, every variable $x$ that is solely owned by $p$ exists in $U$ with the form $x^C$. Replace all such $x$ with the matching $x^C$ form in $\mathcal{I}^Q$.

9. Update $F^P$ to $F^P \wedge \mathcal{I}^P$.

10. Update $F^Q$ to $F^Q \wedge \mathcal{I}^Q$.

11. Let $S'$ be the secure round that follows $S$. If $S$ is the final secure round, then set $S' = \emptyset$.

12. If $O_p$ contains boolean variables, then let $X$ be the set of assignments to all boolean variables in $O_p$. For each $X'$ in $X$
    (a) Let $C' = C \cup X'$.
    (b) Call **create_formula'**$(U, E, F^P, F^Q, C', n, S')$

13. Otherwise, call **create_formula'**$(U, E, F^P, F^Q, C, n, S')$

With the above method to create SMT formulas, we describe the following method for securely removing commitments from a protocol $P$. It considers every input as a candidate for freeing, which is a simple approach that results in many potentially inefficient calls to the SMT solver. We conjecture that there are heuristics that improve the efficiency of this process.

1. Let $P' = P$.

2. For each variable $x$ passed in $\mathcal{C}$ in $P$:
    (a) Create protocol $Q'$ by removing $x$ from $\mathcal{C}$ in $P'$.
    (b) Let $F_A$ be **create_formula**$(A, P', Q')$ and $F_B$ be **create_formula**$(B, P', Q')$.
    (c) If either call fails, continue to the next variable.
    (d) Otherwise, use the SMT solver to check $F_A$ and $F_B$.
    (e) If both are satisfiable, $Q'$ is secure, so let $P' = Q'$.
    (f) Otherwise, continue to the next variable.

3. Output $P'$ as the optimized secure $P$.

**Program 4.2:** $P$: Depth 1 Binary Classification Tree Protocol, With Commits

```
/* A's input is real $x_1$, B's inputs are reals $t_0, v_1, v_2$                    */
/* Output $v$ owned by both parties                                                   */
reactive actively secure round $\mathcal{C}$
    $A$ passes input $x_1$
    $B$ passes inputs $t_0, v_1, v_2$
reactive actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \\ v_2 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    $\begin{cases} \text{abort if not } (R == v_1) & \text{if } C_1 \\ \text{abort if not } (R == v_2) & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{S}_2$
    /* B shares $R$ with A                                                            */
    $v = R$
```

**Program 4.3:** $Q$: Depth 1 Binary Classification Tree Protocol, Without Commits

```
/* A's input is real $x_1$, B's inputs are reals $t_0, v_1, v_2$                    */
/* Output $v$ owned by both parties                                                   */
/* $\mathcal{C}$ is empty                                                             */
reactive actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \\ v_2 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    $\begin{cases} \text{abort if not } (R == v_1) & \text{if } C_1 \\ \text{abort if not } (R == v_2) & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{S}_2$
    /* B shares $R$ with A                                                            */
    $v = R$
```

### 4.2.2 Example Compilation

The above procedure is sufficient to free each variable in Program 4.1 one by one. Showing the details of each step of this process is considerably tedious and the intuition is best demonstrated by a simpler example. We will walk through the procedure that compares a shorter binary classification tree protocol $P$ where all variables are committed to (Program 4.2) directly to a protocol $Q$ where no variables are committed to (Program 4.3). We begin by examining the simulatability of party $B$.

- **create_formula**$(B, P, Q)$ calls **create_formula'**$(\{\}, \{\}, \top, \top, \{\}, \bot, \mathcal{C})$.
- The variables that $B$ passes into $\mathcal{C}$ in $P$ are $t_0, v_1, v_2$. Hence $E = \{t'_0, v'_1, v'_2\}$ without superscript as $C$ is empty.
- $B$ passes in no inputs to $\mathcal{C}$ in $Q$, so nothing is added to $U$.
- $A$ passes in $x_1$ in $P$, so it is added to $U$.
- $S$ is $\mathcal{C}$ which is neither a $\mathcal{S}$ or $\mathcal{I}$ round, hence $F^P$ and $F^Q$ are not updated.
- **create_formula'** is called with $S' = \mathcal{S}_1$. $C$ is still empty and so all $C$ superscripts will be empty in this call.
- In $P$, $B$ passes no new inputs in $\mathcal{S}_1$, so nothing is added to $E$.
- In $Q$, $B$ passes in $t_0$, so $t_0$ is added to $U$.
- In $P$, $A$ passes no new inputs. In $Q$, $A$ passes in $x_1$ but it is already in $U$ so nothing is added.
- $\mathcal{S}_1$ computes the comparison $C_1$ between $A$'s value $x_1$ and $B$'s threshold $t_0$. Hence $O$ is $\{C_1\}$ which is added to $U$, and $\mathcal{I}(\mathcal{S}) = C_1 == x_1 \leq t_0$.
- By variable substitution, $\mathcal{I}^P = C_1 == x_1 \leq t'_0$.
- By variable substitution, $\mathcal{I}^Q = C_1 == x_1 \leq t_0$ ($t_0$ is replaced with $t_0^C$, but $C$ is empty.)
- $F^P$ is updated to $\top \wedge (C_1 == x_1 \leq t'_0)$ and $F^Q$ is updated to $\top \wedge (C_1 == x_1 \leq t_0)$, which trivially simplify to $(C_1 == x_1 \leq t'_0)$ and $(C_1 == x_1 \leq t_0)$. Individually, $F^P$ and $F^Q$ capture that $C_1$ was computed successfully in both protocols. $F^Q \Rightarrow F^P$ captures that whenever $C_1$ is computed successfully in $Q$ it is also computed identically in $P$, and hence that $\mathcal{S}_1$ is equivalent in both protocols.

- The next secure round $S'$ is $\mathcal{I}_2$.
- $O_B$ contains $C_1$, and the set of assignments is $\{C_1 = \top, C_1 = \bot\}$. Hence **create_formula'** is called twice, once with $C = \{C_1 = \top\}$ and once with $C = \{C_1 = \bot\}$. We will show the $\{C_1 = \top\}$ call and generate the $\{C_1 = \bot\}$ output symmetrically.
- In $P$, the classification label $R$ is passed as input to $\mathcal{I}_2$ by $B$, so $R'^{\{C_1=\top\}}$ is added to $E$.
- In $Q$, when $C_1 = \top$, the label $v_1$ is passed as input along with $R$. Hence $R^{\{C_1=\top\}}$ and $v_1^{\{C_1=\top\}}$ are added to $U$.
- $A$ passes no inputs to $\mathcal{I}_2$ and so $U$ is not updated.
- As $C_1 = \top$, $\mathcal{I} = R == v_1$.
- By variable substitution, $\mathcal{I}^P = R'^{\{C_1=\top\}} == v_1'$.
- By variable substitution, $\mathcal{I}^Q = R^{\{C_1=\top\}} == v_1^{\{C_1=\top\}}$.
- $F^P$ is updated to $(C_1 == x_1 \leq t_0') \wedge (R'^{\{C_1=\top\}} == v_1')$.
- $F^Q$ is updated to $(C_1 == x_1 \leq t_0) \wedge (R^{\{C_1=\top\}} == v_1^{\{C_1=\top\}})$
- **create_formula'** is called with $S'$ is set to $\mathcal{S}_2$, which adds $v^{\{C_1=\top\}}$ to $U$, and updates $F^P$ and $F^Q$ with $(v^{\{C_1=\top\}} == R'^{\{C_1=\top\}})$ and $(v^{\{C_1=\top\}} == R^{\{C_1=\top\}})$ respectively.

  As $\mathcal{S}_2$ is the last round, **create_formula'** returns on next call.

Hence the $\{C_1 = \top\}$ branch outputs

$$\begin{aligned}
U =& \{x_1, t_0, C_1, R^{\{C_1=\top\}}, v_1^{\{C_1=\top\}}, v^{\{C_1=\top\}}\} \\
E =& \{t_0', v_1', v_2', R'^{\{C_1=\top\}}\} \\
F =& ((C_1 == x_1 \leq t_0) \wedge (R^{\{C_1=\top\}} == v_1^{\{C_1=\top\}}) \wedge (v^{\{C_1=\top\}} == R^{\{C_1=\top\}})) \Rightarrow \\
& ((C_1 == x_1 \leq t_0') \wedge (R'^{\{C_1=\top\}} == v_1') \wedge (v^{\{C_1=\top\}} == R'^{\{C_1=\top\}}))
\end{aligned}$$

Symmetrically, the $\{C_1 = \bot\}$ branch outputs

$$\begin{aligned}
U =& \{x_1, t_0, C_1, R^{\{C_1=\bot\}}, v_2^{\{C_1=\bot\}}, v^{\{C_1=\bot\}}\} \\
E =& \{t'_0, v'_1, v'_2, R'^{\{C_1=\bot\}}\} \\
F =& ((C_1 == x_1 \leq t_0) \wedge (R^{\{C_1=\bot\}} == v_2^{\{C_1=\bot\}}) \wedge (v^{\{C_1=\bot\}} == R^{\{C_1=\bot\}})) \Rightarrow \\
& \quad ((C_1 == x_1 \leq t'_0) \wedge (R'^{\{C_1=\bot\}} == v'_2) \wedge (v^{\{C_1=\bot\}} == R'^{\{C_1=\bot\}}))
\end{aligned}$$

After aggregation, the final output of **create_formula**$(B, P, Q)$ is

$$\begin{aligned}
&\forall x_1, t_0, C_1, R^{\{C_1=\top\}}, R^{\{C_1=\bot\}}, v_1^{\{C_1=\top\}}, v_2^{\{C_1=\bot\}}, v^{\{C_1=\top\}}, v^{\{C_1=\bot\}} \\
&\exists t'_0, v'_1, v'_2, R'^{\{C_1=\top\}}, R'^{\{C_1=\bot\}} \\
&\quad (((C_1 == x_1 \leq t_0) \wedge (R^{\{C_1=\top\}} == v_1^{\{C_1=\top\}}) \wedge (v^{\{C_1=\top\}} == R^{\{C_1=\top\}})) \Rightarrow \\
&\quad ((C_1 == x_1 \leq t'_0) \wedge (R'^{\{C_1=\top\}} == v'_1) \wedge (v^{\{C_1=\top\}} == R'^{\{C_1=\top\}}))) \\
&\quad \wedge \\
&\quad (((C_1 == x_1 \leq t_0) \wedge (R^{\{C_1=\bot\}} == v_2^{\{C_1=\bot\}}) \wedge (v^{\{C_1=\bot\}} == R^{\{C_1=\bot\}})) \Rightarrow \\
&\quad ((C_1 == x_1 \leq t'_0) \wedge (R'^{\{C_1=\bot\}} == v'_2) \wedge (v^{\{C_1=\bot\}} == R'^{\{C_1=\bot\}})))
\end{aligned}$$

This formula is indeed satisfiable. With the exception of $v'_1$ and $v'_2$, every primed variable in $E$ has an equivalent non-primed variable in $U$ with the same label $C$. This implies that a simulator $S$ can directly copy the value of these variables from the simulated execution of $Q$ to their execution of $P$ with $T$. This corresponds to a partial assignment of the primed variables in $E$ to their non-primed versions in $U$, e.g. setting $t'_0 = t_0$. After partial assignment and routine boolean formula simplification, the remaining formula is

$$\begin{aligned}
&\forall v_1^{\{C_1=\top\}}, v_2^{\{C_1=\bot\}}, R^{\{C_1=\top\}}, R^{\{C_1=\bot\}} \\
&\exists v'_1, v'_2 \\
&\quad ((R^{\{C_1=\top\}} == v_1^{\{C_1=\top\}}) \Rightarrow (R^{\{C_1=\top\}} == v'_1)) \\
&\quad \wedge \\
&\quad ((R^{\{C_1=\bot\}} == v_2^{\{C_1=\bot\}}) \Rightarrow (R^{\{C_1=\bot\}} == v'_2))
\end{aligned}$$

This formula contains the essential difference between $P$ and $Q$, which is that in $Q$, $v_1$ and $v_2$ can be chosen as a function of $C_1$. Clearly, it is satisfiable with $v'_1 = v_1^{\{C_1=\top\}}$ and

$v'_2 = v_2^{\{C_1=\bot\}}$. This captures that $S$ can force equivalency by simulating a $C_1 = \top$ execution and a $C_1 = \bot$ execution of $P$ to extract $v_1^{\{C_1=\top\}}$ and $v_2^{\{C_1=\bot\}}$, and then providing those to $T$ as $v'_1$ and $v'_2$ respectively. Indeed, given any satisfying assignment to a returned formula, the corresponding simulator proceeds by simulating each execution branch that exists in any variable label in the assignment. This suggests that any SMT solver that returns a satisfying assignment as proof of satisfiability could be used to programmatically create the appropriate simulator, although we do not describe the details of such a procedure here.

The equivalent **create_formula** call for party $A$ returns a formula that is trivially satisfiable, as each variable in $E$ has a corresponding variable in $U$. As both formulas are satisfiable, $Q$ is secure.

Program 4.4 shows the result of this optimization on Program 4.1. One variable from Program 4.1 is freed at each step of the compilation, and the security of the resulting protocols follows very similarly to the example above. As all variables have been freed, $\mathcal{C}$ is removed entirely. This brings the overall asymptotic complexity of the actively secure classification protocols down to match that of the passively secure protocols, i.e. they are both linear in tree depth. The remaining structural difference between passively secure Program 4.4 and actively secure Program 3.3 is in the integrity rounds, which can be further optimized by our next technique.

## 4.3 Integrity Function Reduction

In this optimization, we exploit free variables to enable simpler integrity functions to be used without affecting security. As a motivation, consider an execution of Program 4.4 and suppose the first comparison resulted in $C_1 == \top$. An honest $B$ would set the next threshold $N$ to $t_1$, which is checked by the corresponding integrity function $\mathcal{I}_2 = N == t_1$. However as $t_1$ is free, an adversarial $B$ can simultaneously choose both $N$ and $t_1$. As such, an adversary has only two options — they can choose to send an *arbitrary* but equal value for $N$ and $t_1$ and continue the protocol, or they can send different values, causing $\mathcal{I}_2$ to evaluate false and the protocol to abort. As an unrestricted adversary can also abort arbitrarily and choose $N$ arbitrarily, intuitively $\mathcal{I}_2$ is not preventing any adversarial behaviour and can be removed entirely.

More generally, the purpose of integrity rounds is to validate that newly provided inputs have been generated following the protocol. As such, integrity rounds check that certain restrictions over the possible values of a party's variables are being met, e.g. that

**Program 4.4:** Actively Secure Binary Classification Tree Protocol
(After known boolean splitting, commitments removed)

```
/* A's inputs are numbers $x_1, x_2$                                    */
/* B's inputs are numbers $t_0, t_1, t_2, v_1, v_2, v_3, v_4$           */
/* Output $v$ owned by both parties                                     */
reactive actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    $\begin{cases} \text{abort if not } (N == t_1) & \text{if } C_1 \\ \text{abort if not } (N == t_2) & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{S}_2$
    $C_2 = x_2 \leq N$
local round $\mathcal{L}_3^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{I}_3$
    $\begin{cases} \text{abort if not } (R == v_1) & \text{if } C_1 \wedge C_2 \\ \text{abort if not } (R == v_2) & \text{if } C_1 \wedge \neg C_2 \\ \text{abort if not } (R == v_3) & \text{if } \neg C_1 \wedge C_2 \\ \text{abort if not } (R == v_4) & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{S}_3$
    /* B shares $R$ with A                                              */
    $v = R$
```

$N == t_1$. However, we have no guarantee that the integrity functions generated by our compilation are the simplest functions that check these restrictions. Given an integrity function $\mathcal{I}$, if an adversary can use inputs satisfying a simpler integrity function $\mathcal{I}'$ to ensure that $\mathcal{I}$ is always satisfied, then $\mathcal{I}'$ is a secure replacement for $\mathcal{I}$. In the above example, $N == t_1$ can be trivially satisfied by the adversary and so $N == t_1$ can be replaced with an empty function, but in general more nuanced simplifications may exist. Given a candidate simplified integrity function, we can check if it is secure to use this function using the same simulatability SMT formula construction described in Section 4.2. We describe some formula simplification methods below.

### 4.3.1 Compilation Algorithm

Our procedure for simplifying integrity functions is the following:

1. Let $P' = P$.
2. For each integrity circuit $\mathcal{I}$ in any integrity round in $P$:
   (a) Create candidate $\mathcal{I}'$ from $\mathcal{I}$.
   (b) Let $Q'$ be a copy of $P'$ that uses $\mathcal{I}'$ instead of $\mathcal{I}$.
   (c) Let $F_A$ be **create_formula**$(A, P', Q')$ and $F_B$ be **create_formula**$(B, P', Q')$.
   (d) If either call fails, continue to the next round.
   (e) Otherwise, use the SMT solver to check $F_A$ and $F_B$.
   (f) If both are satisfiable, $Q'$ is secure, so let $P' = Q'$.
3. Output $P'$ as the optimized secure $P$.

This procedure does not specify how a candidate $\mathcal{I}'$ should be created from $\mathcal{I}$. In general, any process can be used, as any insecure candidate $\mathcal{I}'$ will be caught by the unsatisfiability of the resulting **create_formula** output. Hence a simple heuristic is to always try the empty formula $\mathcal{I}' = \top$. Of course, this is likely to fail for most integrity rounds, as in general there is a meaningful restriction on inputs that needs to be checked. How to optimally simplify an integrity function is an open problem, but we present some partial results here.

For any integrity round $\mathcal{I}$, we call a variable $x$ *loose* if it is possible for $\mathcal{I}$ to be the only secure round in the protocol execution that uses $x$ as input. If we can simplify $\mathcal{I}$ such that it does not use $x$ then it is possible that no secure state is established for $x$, which reduces the concrete cost of execution. Our simplification procedures remove all loose variables from all $\mathcal{I}$ rounds, thereby reducing the cost of establishing secure state.

Our simplification procedures do not reduce integrity rounds in isolation. Suppose we have two integrity rounds $\mathcal{I}_1$ and $\mathcal{I}_2$ that both contain a loose variable $x$ owned by party $A$, and let $\mathcal{I}_1^A$ and $\mathcal{I}_2^A$ be the circuits checking the integrity of $A$'s variables in $\mathcal{I}_1$ and $\mathcal{I}_2$ respectively. If there exists ${\mathcal{I}'}_1^A$ that does not depend on $x$ such that when it is satisfied an adversary can always generate a value for $x$ that satisfies $\mathcal{I}_1^A$, then ${\mathcal{I}'}_1^A$ can be used. Similarly, if such an ${\mathcal{I}'}_2^A$ can be found then it can be used as well, removing $x$ entirely from $\mathcal{I}_1$ and $\mathcal{I}_2$. However, there is no guarantee that the values for $x$ produced by the adversary in each simulated round will agree with each other, and if they can be inconsistent then the simplified integrity rounds will result in unsatisfiable formulas. Hence to simplify $\mathcal{I}_2$, we simplify the circuit $\mathcal{I}_1^A \wedge \mathcal{I}_2^A$, rather than just $\mathcal{I}_2^A$. This guarantees that the simplified circuit will be consistent with the simplified $\mathcal{I}_1^A$ used in the reduced $\mathcal{I}_1$ round.

We provide two simplification procedures for restricted classes of protocols. The first is for protocols where all integrity functions are *pure* boolean circuits, that is, all variables are boolean valued. The second is for protocols where all integrity functions check inequalities over real numbers. Hence our $\mathcal{I}'$ candidate creation process is as follows:

1. On input $\mathcal{I}_j$, define $C_A$ and $C_B$ as

$$C_A = \bigwedge_{i=0}^{j} \mathcal{I}_i^A$$

$$C_B = \bigwedge_{i=0}^{j} \mathcal{I}_i^B$$

   where $\mathcal{I}_i^p$ is the circuit checking party $p$'s input integrity in round $\mathcal{I}_i$ in the branch of execution corresponding to $\mathcal{I}_j$.

2. For $p$ in $\{A, B\}$, if $C_p$ is a pure boolean circuit, or consists only of conjunctions of real number inequalities, simplify it using the appropriate procedure below to create $C'_p$. Otherwise, set $C'_p = \top$.

3. Output $\mathcal{I}'_j$ as an integrity round that checks ${C'}_A \wedge {C'}_B$.

We now discuss our pure boolean circuit simplification procedure. Suppose we have some pure boolean circuit $C$ with loose variable $x$. Consider the modified circuit $C' =$

$C\langle x \coloneqq \top\rangle \vee C\langle x \coloneqq \bot\rangle$. $C'$ is satisfiable if and only if $C$ is satisfiable, and $C'$ does not contain $x$, so this makes $C'$ a suitable candidate circuit. Hence we can reduce $C$ as follows:

1. For all loose variables $x$ in $C$:
    (a) Let $C' = C\langle x \coloneqq \top\rangle \vee C\langle x \coloneqq \bot\rangle$.
    (b) Optionally, use boolean formula simplification techniques to simplify $C'$.
    (c) Let $C = C'$.
2. Return $C$.

In the worst case $C'$ will have twice as many gates as $C$, but in practice boolean algebra techniques [11, 14] may simplify $C'$ significantly. Hence this simplification allows a protocol designer to balance the concrete cost of a commitment with the concrete cost of secure gate evaluations.

Finally, we present our simplification procedure for real inequalities. When $C$ consists of *conjunctions of inequalities over real numbers*, we have an efficient procedure that removes loose variables without increasing the number of gates in the circuit. The intuition is simple. Given some loose variable $x$, if $x == y$ then an adversary can trivially pass $y$ as the value for $x$. If we require $w < x < y$ then as long as $w < y$, there is always some real value between $w$ and $y$ that an adversary can pass for $x$. We can iteratively remove loose variables from inequalities based on this intuition.

Any conjunction of inequalities like

$$(a == b) \wedge (c \geq a) \wedge (d < e)$$

has an associated graph representation like

$$\begin{tikzcd} a \arrow[r, leftrightarrow, "=="] \arrow[dr, "\leq"] & b & & d \arrow[r, "<"] & e \\ & c & & & \end{tikzcd}$$

Specifically, inequalities can be transformed into graphs with the following structure:

- There is one node for every variable in the inequalities.
- There is an equals edge $x \overset{==}{\longleftrightarrow} y$ if $x == y$ is present in the inequalities.
- There is a less-than-or-equals edge $x \xrightarrow{\leq} y$ if $x \leq y$ or $y \geq x$ is present in the inequalities.
- There is a less-than edge $x \xrightarrow{<} y$ if $x < y$ or $y > x$ is present in the inequalities.

Suppose we have a graph $G = (V, E)$ formed by the above process, and let $L$ be the subset of nodes corresponding to loose variables. Then we can remove all free nodes from $G$ through the following process.

1. While there exist equals edges $x \overset{==}{\longleftrightarrow} y$ where $x \in L$, *merge* $x$ into $y$, that is:
   (a) For each outgoing edge $e = (x, z)$ add edge $e' = (y, z)$ with same type as $e$.
   (b) For each incoming edge $e = (z, x)$ add edge $e' = (z, y)$ with same type as $e$.
   (c) Delete $x$ and all its edges.
2. While there exist nodes $x \in L$:
   (a) Let $O$ be the nodes outgoing from $x$, that is $\{y \mid (x, y) \in E\}$.
   (b) For each $z \in O$ and each incoming edge $e = (y, x)$, create a new edge $e' = (y, z)$. If $(y, x)$ and $(x, z)$ are both are less-than-or-equals edges, then $e'$ is as well. Otherwise, $e'$ is a less-than edge.
   (c) If this creates an edge of the form $x \xrightarrow{<} x$, fail. (This implies the original inequalities were unsatisfiable, which does not occur for integrity functions created by our compilation process.)
   (d) Delete $x$ and all its edges.
3. Delete any self edges of the form $x \xrightarrow{\leq} x$.
4. Delete any duplicate edges from $G$.
5. Delete any connected components that only contain nodes from $L$.
6. Delete any nodes that have no edges.

The reduced inequality graph is converted back to a formula by creating a clause for each edge and taking the conjunction over them.

As an example, consider the following integrity function

$$(f_1 == c_1) \wedge (f_1 < c_2) \wedge (c_2 < f_2) \wedge (f_2 < c_3) \wedge (f_2 < c_4)$$

where $f_1$ and $f_2$ are loose variables and $c_1$, $c_2$, $c_3$ and $c_4$ are not.

This is transformed into the following graph.

$$\begin{tikzcd}
c_1 \arrow[r, leftrightarrow, "=="] & f_1 \arrow[r, "<"] & c_2 \arrow[r, "<"] & f_2 \arrow[r, "<"] \arrow[dr, "<"] & c_3 \\
 & & & & c_4
\end{tikzcd}$$

First, equals edges are removed by merging $f_1$ with $c_1$.

$$\begin{tikzcd}
c_1 \arrow[rr, "<"] & & c_2 \arrow[r, "<"] & f_2 \arrow[r, "<"] \arrow[dr, "<"] & c_3 \\
 & & & & c_4
\end{tikzcd}$$

Next, $f_2$ is removed and its less-than edges are transferred to $c_2$.

$$\begin{tikzcd}
c_1 \arrow[rr, "<"] & & c_2 \arrow[rr, "<"] \arrow[drr, "<"] & & c_3 \\
 & & & & c_4
\end{tikzcd}$$

The resulting graph is transformed back into the inequality function

$$(c_1 < c_2) \wedge (c_2 < c_3) \wedge (c_2 < c_4)$$

This process ensures that as long as the reduced function is satisfied, there is a satisfiable assignment to $f_1$ and $f_2$ in the original function, which makes the reduced function a good candidate for $\mathcal{I}'$. As we will show below, the inequality reduction procedure handles the integrity functions of median computation and is sufficient to automatically recreate Aggarwal et al.'s actively secure two party median protocol [1].

We conjecture that our boolean reduction procedure and inequality reduction procedure are guaranteed to produce $\mathcal{I}'$ formulas that can be securely used instead of $\mathcal{I}$. If this is true, our procedure would not need to check the satisfiability of the simulatability formulas when these procedures are used. We leave proving this to future work, along with finding secure simplification procedures for other classes of integrity formulas.

**Program 4.5:** Actively Secure Binary Classification Tree Protocol
(With all optimizations applied)

```
/* A's inputs are reals $x_1, x_2$                                   */
/* B's inputs are reals $t_0, t_1, t_2, v_1, v_2, v_3, v_4$          */
/* Output $v$ owned by both parties                                  */
actively secure round $\mathcal{S}_1$
    $C_1 = x_1 \leq t_0$
local round $\mathcal{L}_2^B$
    $N = \begin{cases} t_1 & \text{if } C_1 \\ t_2 & \text{if } \neg C_1 \end{cases}$
actively secure round $\mathcal{S}_2$
    $C_2 = x_2 \leq N$
local round $\mathcal{L}_3^B$
    $R = \begin{cases} v_1 & \text{if } C_1 \wedge C_2 \\ v_2 & \text{if } C_1 \wedge \neg C_2 \\ v_3 & \text{if } \neg C_1 \wedge C_2 \\ v_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
actively secure round $\mathcal{S}_3$
    /* B shares $R$ with A                                           */
    $v = R$
```

### 4.3.2 Example Compilation

Consider running this procedure on Program 4.4. At $\mathcal{I}_2$, the integrity function is $N == t_i$ where $i \in \{1, 2\}$. $t_i$ is loose as neither are used in any other secure round, and $N$ is not as it is definitely used in $\mathcal{S}_2$. Hence the inequality reduction procedure merges $t_i$ to $N$ leaving a single node without edges. This node is removed, leaving the empty function. $\mathcal{I}_3$ is removed symmetrically and hence all reduced integrity functions are empty. The resulting protocol without integrity functions is Program 4.5. Note that as no inputs are reused between secure rounds in Program 4.5, the reactive functionality required in Program 4.4 is no longer required.

Although our procedure would generate Program 4.4 by removing one integrity function at a time, for simplicity we will show the generated SMT formula when all are removed simultaneously. That is, we consider an execution of **create_formula** where $P$ is Program 4.4 and $Q$ is Program 4.5, beginning with party $B$.

Consider the **create_formula'** call corresponding to the branch $\{C_1 == \top, C_2 == \top\}$. Omitting the context superscripts for clarity, the generated formula is

$$\begin{aligned}
&\forall t_0, N, R, x_1, x_2, C_1, C_2, v \\
&\exists t'_0, t'_1, N', v'_1, R' \\
&\quad ((C_1 == x_1 <= t_0) \wedge (C_2 == x_2 <= N) \wedge (v == R)) \\
&\quad \Rightarrow \\
&\quad ((C_1 == x_1 <= t'_0) \wedge (N' == t'_1) \wedge (C_2 == x_2 <= N') \wedge (R' == v'_1) \wedge (v == R'))
\end{aligned}$$

This is trivially satisfiable with $t'_0 = t_0$, $t'_1 = N' = N$, and $v'_1 = R' = R$, which corresponds to a simulator that extracts the thresholds that the adversary passes in $Q$ and uses them to trivially satisfy the integrity functions in $P$. The adversary can do this for each branch of the computation, and so the full formula representing the conjunction over all branches is satisfiable symmetrically. This shows the simulatability of party $B$, and as there is no difference between $P$ and $Q$ for party $A$, Program 4.5 securely computes the binary classification tree protocol.

Program 4.5 is our fully optimized protocol. It differs from the passively secure Program 3.3 only in that the secure rounds must be computed using actively secure SMC. In essence, this demonstrates that for this specific protocol our optimizations remove all the inefficiencies generated by Goldreich compilation. If we apply our optimizations to binary classification tree protocols of increasing tree depth, the resulting protocols have exponentially fewer secure gate evaluations than the single mode actively secure protocol for the same tree depth. We do note that each additional tree depth adds one round of secure computation, so as constant round complexity SMC protocols exist [2, 20] the mixed mode protocols have a linear increase in the rounds of communication compared to any single mode protocol for the same tree depth.

# Chapter 5

# Compilation Example: Median Computation

Aggarwal et al. contributed a mixed mode protocol [1] for calculating the median element over two parties' input sets. This protocol is a well-known example of an optimized actively secure mixed mode protocol. At each round, both parties calculate the medians over their own input sets, securely compare them, and discard half of their inputs based on the comparison result. To achieve active security, Aggarwal et al. showed that it is sufficient to have a reactive functionality track the maximum and minimum values of each party's comparison inputs and check only that new inputs respect these bounds. We show here that our optimizations can automatically recreate their optimized protocol.

Program 5.1 is the instantiation of their protocol with four elements per party. It is passively secure, as proved manually by Aggarwal et al. and automatically deduced by the passively secure compilation techniques described in 2.5. We begin by applying the unoptimized actively secure construction of Section 2.6 to create Program 5.2.

We apply our optimizations in order to create Program 5.2. Firstly, as $C_1$ and $C_2$ are known to both parties, known boolean splitting simplifies all integrity functions. Next, all inputs are trivially freed from $\mathcal{C}$. The simulatability formula is trivially satisfiable as all inputs are immediately passed into $\mathcal{I}_0$, and hence they cannot be maliciously chosen as a function of runtime information. Although the integrity functions cannot be fully simplified away as in Program 4.5, they can be simplified significantly by our inequality reduction procedure.

The reduction proceeds very similarly for both parties in each branch, so we demonstrate it only for party $A$ when $C_1$ and $C_2$ are both true. The circuit corresponding to $\mathcal{I}_0$

**Program 5.1:** Passively Secure Median Computation

```
/* A's inputs are reals s.t.  $a_1 < a_2 < a_3 < a_4$                */
/* B's inputs are reals s.t.  $b_1 < b_2 < b_3 < b_4$                */
/* Output $r$ owned by both parties                                  */
passively secure round $\mathcal{S}_1$
    $C_1 = a_2 \leq b_2$
local round $\mathcal{L}_2^A$
    $M_2^A = \begin{cases} a_3 & \text{if } C_1 \\ a_1 & \text{if } \neg C_1 \end{cases}$
local round $\mathcal{L}_2^B$
    $M_2^B = \begin{cases} b_1 & \text{if } C_1 \\ b_3 & \text{if } \neg C_1 \end{cases}$
passively secure round $\mathcal{S}_2$
    $C_2 = M_2^A \leq M_2^B$
local round $\mathcal{L}_3^A$
    $M_3^A = \begin{cases} a_4 & \text{if } C_1 \wedge C_2 \\ a_3 & \text{if } C_1 \wedge \neg C_2 \\ a_2 & \text{if } \neg C_1 \wedge C_2 \\ a_1 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
local round $\mathcal{L}_3^B$
    $M_3^B = \begin{cases} b_1 & \text{if } C_1 \wedge C_2 \\ b_2 & \text{if } C_1 \wedge \neg C_2 \\ b_3 & \text{if } \neg C_1 \wedge C_2 \\ b_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
passively secure round $\mathcal{S}_3$
    $v = \min(M_3^A, M_3^B)$
```

**Program 5.2:** Actively Secure Median Computation (Unoptimized)

```
reactive actively secure round $\mathcal{C}$
    $A$ passes inputs $a_1, a_2, a_3, a_4$
    $B$ passes inputs $b_1, b_2, b_3, b_4$
reactive actively secure round $\mathcal{I}_0$
    abort if not $(a_1 < a_2) \wedge (a_2 < a_3) \wedge (a_3 < a_4) \wedge (b_1 < b_2) \wedge (b_2 < b_3) \wedge (b_3 < b_4)$
reactive actively secure round $\mathcal{S}_1$
    $C_1 = a_2 \leq b_2$
local round $\mathcal{L}_2^A$
    $M_2^A = \begin{cases} a_3 & \text{if } C_1 \\ a_1 & \text{if } \neg C_1 \end{cases}$
local round $\mathcal{L}_2^B$
    $M_2^B = \begin{cases} b_1 & \text{if } C_1 \\ b_3 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    abort if not
    $\big((C_1 \wedge (M_2^A == a_3)) \vee (\neg C_1 \wedge (M_2^A == a_1))\big) \wedge \big((C_1 \wedge (M_2^B == b_1)) \vee (\neg C_1 \wedge (M_2^B == b_3))\big)$
reactive actively secure round $\mathcal{S}_2$
    $C_2 = M_2^A \leq M_2^B$
local round $\mathcal{L}_3^A$
    $M_3^A = \begin{cases} a_4 & \text{if } C_1 \wedge C_2 \\ a_3 & \text{if } C_1 \wedge \neg C_2 \\ a_2 & \text{if } \neg C_1 \wedge C_2 \\ a_1 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
local round $\mathcal{L}_3^B$
    $M_3^B = \begin{cases} b_1 & \text{if } C_1 \wedge C_2 \\ b_2 & \text{if } C_1 \wedge \neg C_2 \\ b_3 & \text{if } \neg C_1 \wedge C_2 \\ b_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{I}_3$
    abort if not
    $\big((C_1 \wedge C_2 \wedge (M_3^A == a_4)) \vee (C_1 \wedge \neg C_2 \wedge (M_3^A == a_3)) \vee$
     $(\neg C_1 \wedge C_2 \wedge (M_3^A == a_2)) \vee (\neg C_1 \wedge \neg C_2 \wedge (M_3^A == a_1))\big) \wedge$
     $\big((C_1 \wedge C_2 \wedge (M_3^B == b_1)) \vee (C_1 \wedge \neg C_2 \wedge (M_3^B == b_2)) \vee$
     $(\neg C_1 \wedge C_2 \wedge (M_3^B == b_3)) \vee (\neg C_1 \wedge \neg C_2 \wedge (M_3^B == b_4))\big)$
reactive actively secure round $\mathcal{S}_3$
    $v = \min(M_3^A, M_3^B)$
```

is $\mathcal{I}_0^A = (a_1 < a_2) \wedge (a_2 < a_3) \wedge (a_3 < a_4)$. All variables are loose except $a_2$ which will be definitely used in $\mathcal{S}_1$. Hence all other variables are removed leaving $a_2$ edges. Hence $a_2$ is removed as well, which results in an empty formula $\top$. $\mathcal{I}_1$ is empty, and the circuit corresponding to $\mathcal{I}_2$ is $\mathcal{I}_0^A \wedge \mathcal{I}_2^A = (a_1 < a_2) \wedge (a_2 < a_3) \wedge (a_3 < a_4) \wedge (M_2^A == a_3)$. All variables are loose except $a_2$ which has already been used in $\mathcal{S}_1$ and $M_2^A$ which will definitely be used in $\mathcal{S}_2$. The reduction procedure merges $a_3$ into $M_2^A$ and discards everything except $a_2 < M_2^A$. Similarly, the circuit corresponding to $\mathcal{I}_3$ is reduced to $M_2^A < M_3^A$.

The satisfiability of the corresponding simulatability formulas captures that a simulator $S$ for $A$ can do the following. By passing $C_1 = \top$ and $C_2 = \top$ to party $A$, $S$ receives $a_2$, $M_2^A$, and $M_3^A$ such that $a_2 < M_2^A < M_3^A$. $S$ can choose $a'_1$ less than $a_2$ and set $a'_2 = a_2$, $a'_3 = M_2^A$, $a'_4 = M_3^A$, and of course $M_2'^A = M_2^A$ and $M_3'^A = M_3^A$. By providing these to $T$, $S$ is guaranteed to satisfy the ideal world integrity formulas $a'_1 < a'_2 < a'_3 < a'_4$, $M_2'^A == a'_3$ and $M_3'^A == a'_4$ that arise during this branch of execution. The simulatability of each other branch, and of the whole protocol execution for $A$ and $B$, follows symmetrically.

In summary, unlike the fully optimized binary classification protocol Program 4.5, the optimized median protocol contains some reduced integrity functions that must be checked before $\mathcal{S}_2$ and $\mathcal{S}_3$. They have been optimized significantly to just the conjunction of two comparisons. The optimized integrity functions check that at each round, the next partial median $M_i^p$ is correctly bounded by partial medians from previous rounds. This can be viewed as a reactive functionality that tracks an updating upper and lower bound for each party. Each subsequent $M_i^p$ updates the bounds for party $p$, and an honest party will always send partial medians that respect these bounds. Hence the integrity functions detect whether a party has deviated from the protocol by sending an $M_i^p$ that does not correspond with their respective bounds. This is exactly the strategy for achieving active security that is proposed by Aggarwal et al. in [1]. As such, our optimizations automatically recreate their optimized actively secure protocol.

**Program 5.3:** Actively Secure Median Computation (Optimized)

```
/* A's inputs are reals s.t.  $a_1 < a_2 < a_3 < a_4$                          */
/* B's inputs are reals s.t.  $b_1 < b_2 < b_3 < b_4$                          */
/* Output $r$ owned by both parties                                            */
reactive actively secure round $\mathcal{S}_1$
    $C_1 = a_2 \leq b_2$
local round $\mathcal{L}_2^A$
    $M_2^A = \begin{cases} a_3 & \text{if } C_1 \\ a_1 & \text{if } \neg C_1 \end{cases}$
local round $\mathcal{L}_2^B$
    $M_2^B = \begin{cases} b_1 & \text{if } C_1 \\ b_3 & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{I}_2$
    $\begin{cases} \text{abort if not } (a_2 < M_2^A) \wedge (M_2^B < b_2) & \text{if } C_1 \\ \text{abort if not } (M_2^A < a_2) \wedge (b_2 < M_2^B) & \text{if } \neg C_1 \end{cases}$
reactive actively secure round $\mathcal{S}_2$
    $C_2 = M_2^A \leq M_2^B$
local round $\mathcal{L}_3^A$
    $M_3^A = \begin{cases} a_4 & \text{if } C_1 \wedge C_2 \\ a_3 & \text{if } C_1 \wedge \neg C_2 \\ a_2 & \text{if } \neg C_1 \wedge C_2 \\ a_1 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
local round $\mathcal{L}_3^B$
    $M_3^B = \begin{cases} b_1 & \text{if } C_1 \wedge C_2 \\ b_2 & \text{if } C_1 \wedge \neg C_2 \\ b_3 & \text{if } \neg C_1 \wedge C_2 \\ b_4 & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{I}_3$
    $\begin{cases} \text{abort if not } (M_2^A < M_3^A) \wedge (M_3^B == M_2^B) & \text{if } C_1 \wedge C_2 \\ \text{abort if not } (M_3^A == M_2^A) \wedge (M_3^B == b_2) & \text{if } C_1 \wedge \neg C_2 \\ \text{abort if not } (M_3^A == a_2) \wedge (M_3^B == M_2^B) & \text{if } \neg C_1 \wedge C_2 \\ \text{abort if not } (M_3^A == M_2^A) \wedge (M_2^B < M_3^B) & \text{if } \neg C_1 \wedge \neg C_2 \end{cases}$
reactive actively secure round $\mathcal{S}_3$
    $v = \min(M_3^A, M_3^B)$
```

# Chapter 6

# Conclusion

In this thesis, we have shown that it is feasible to compile circuits for two party functionalities into efficient mixed mode two party protocols that are actively secure. Our results build on the efficient passively secure mixed mode protocol compilation techniques of Kerschbaum [12] and Rastogi et al. [16] by applying novel optimization techniques to an inefficient passive-to-active Goldreich compilation [8]. Following the prior work on passively secure compilation techniques [16], we formalize our optimizations as computational problems that can be solved with SMT solvers like Z3 [6]. Our compilation process is robust enough to generate Aggarwal et al.'s extremely efficient actively secure protocol for calculation of the median element [1]. Furthermore, our compilation process generates extremely asymptotically efficient protocols for a variant of classification using binary trees. Compared to single mode binary classification tree protocols, the optimized mixed mode protocols have an exponential decrease in computation and communication cost with a linear increase in rounds of communication. We measured a corresponding concrete decrease in protocol runtime, which confirms the practical effectiveness of our optimizations. We believe this represents an important step in the study of automatic optimizations for actively secure computation protocols, and that implementing these optimizations in a compiler would be valuable future work.

# References

[1] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the kth-ranked element. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 40–55, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[2] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[3] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *Proceedings of the 11th International Conference on Theory and Application of Cryptology and Information Security*, ASIACRYPT'05, pages 236–252, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 494–503, New York, NY, USA, 2002. ACM.

[5] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, pages 643–662, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[8] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[9] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: general-purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[10] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer — efficiently. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, pages 572–591, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.

[12] Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 703–714, New York, NY, USA, 2011. ACM.

[13] Yehuda Lindell. *How to Simulate It – A Tutorial on the Simulation Proof Technique*, pages 277–346. Springer International Publishing, Cham, 2017.

[14] Edward J McCluskey Jr. Minimization of boolean functions. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[15] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 655–670, Washington, DC, USA, 2014. IEEE Computer Society.

[16] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A. Hammer. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 3–14, New York, NY, USA, 2013. ACM.

[17] Abhi Shelat and Muthuramakrishnan Venkitasubramaniam. Secure computation from millionaire. In *Proceedings, Part I, of the 21st International Conference on Advances in Cryptology – ASIACRYPT 2015 - Volume 9452*, pages 736–757, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[18] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 3:1–3:18, New York, NY, USA, 2019. ACM.

[19] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient Multi-Party computation toolkit. https://github.com/emp-toolkit, 2016.

[20] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 21–37, New York, NY, USA, 2017. ACM.

[21] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

# APPENDICES

# Appendix A

# Simulatability Formulas For Aborting Adversaries

Here we describe how to capture the simulatability of arbitrary adversaries, i.e. adversaries that can cause execution to abort early. The overall structure of the resulting SMT formulas comes from the following intuition:

- Secure rounds can be equivalent in success, or equivalent in failure.
- Two $\mathcal{S}_i$ rounds are equivalent in failure if the adversary aborts in both.
- Two $\mathcal{S}_i$ rounds are equivalent in success if all outputs are equal in both protocols, all outputs match the integrity function, and neither protocol is aborted.
- Two $\mathcal{I}_i$ rounds are equivalent in failure if both output false, or if the adversary aborts in both. (The honest party will only see $T$ abort in either case.)
- Two $\mathcal{I}_i$ rounds are equivalent in success if both output true and neither protocol is aborted.
- A single branch of protocol execution is simulatable if both protocols are equivalent in success at every round, or if they are equivalent in failure the first round, or if they are equivalent in success in the first round and identical in success at the second round, etc. That is, they are equivalent in success up to some specific round, and then both fail.

We augment **create_formula'** from Section 4.2 to handle the potential for failure. In addition to the standard procedure there is as additional boolean variable for each secure round representing whether the adversary aborts and another representing whether the simulator aborts. The procedure has the following inputs:

- $U$, the set of universally quantified variables.
- $E$, the set of existentially quantified variables.
- $F$, the formula representing equivalence up the current round.
- $B$, the formula representing equivalent success up to the current round.
- $n$, a boolean flag which tracks whether a non-boolean output has been received.
- $S$, which tracks the current secure round in both protocols.

**create_formula"** is called from **create_formula**$(p, P, Q)$ defined in Section 4.2, and is defined as **create_formula"**$(U, E, F, B, C, n, S)$:

1. If $S = \emptyset$, set $F = F \vee B$ and return $(U, E, F)$.
2. Let $I_p^P$ be the set of variables used in round $S$ in $P$ that are owned solely by $p$ and are not already present in $E$. If $n$ is true and $I_p^P$ is non-empty, return $\perp$. Otherwise, for each $x$ in $I_p^P$, add a new variable $x'^C$ in $E$.
3. Add a boolean variable $a'^C_S$ to $E$, representing whether $S$ aborts this round in $P$.
4. Let $I_p^Q$ be the set of variables used in round $S$ in $Q$ that are owned solely by $p$ and are not already present in $U$. For each $x$ in $I_p^Q$, add a new variable $x^C$ in $U$.
5. Add a boolean variable $a^C_S$ to $U$, representing whether $p$ aborts this round in $Q$.
6. Let $I_{\neg p}$ be the set of variables used in round $S$ in either $P$ or $Q$ that are owned solely by $\neg p$ and are not already present in $U$. For each $x$ in $I_{\neg p}$, add a new variable $x$ to $U$.
7. If $S$ is a secure round $\mathcal{S}_i$:
   (a) By our restrictions on the forms of $P$ and $Q$, $S$ has the same outputs in both protocols. Let $\mathcal{S}$ be the circuit computed by $S$, and let $O$ be the set of its outputs.

(b) For every variable $x$ in $O$, add a variable $x^C$ to $U$.

(c) Let $O_p$ be the subset of $O$ that is owned by $p$. If $O_p$ contains any non-boolean variables, set $n$ to $\top$.

(d) Let $\mathcal{I}^P$ be the integrity formula $\mathcal{I}(\mathcal{S})$. Every variable $x$ in $\mathcal{I}^P$ that is an input owned by $p$ exists in $E$ in the form $x'^C$. Replace all such $x$ with the matching $x'^C$ form in $\mathcal{I}^P$.

(e) Let $\mathcal{I}^Q$ be the integrity formula $\mathcal{I}(\mathcal{S})$. Every variable $x$ in $\mathcal{I}^Q$ that is an input owned by $p$ exists in $E$ in the form $x^C$. Replace all such $x$ with the matching $x^C$ form in $\mathcal{I}^Q$.

(f) Update $F^F$ to $F \vee (B \wedge a'^C_S \wedge a^C_S)$.

(g) Update $B$ to $B \wedge \neg a'^C_S \wedge \mathcal{I}^P \wedge \neg a^C_S \wedge \mathcal{I}^Q$.

8. If $S$ is a secure round $\mathcal{I}_i$:

   (a) Let $\mathcal{I}^P$ be the circuit computed by $\mathcal{I}$ in $P$ and $\mathcal{I}^Q$ be the circuit computed by $\mathcal{I}$ in $Q$.

   (b) Every variable $x$ in $\mathcal{I}^P$ that is an input owned by $p$ exists in $E$ in the form $x'^C$. Replace all such $x$ with the matching $x'^C$ form in $\mathcal{I}^P$.

   (c) Every variable $x$ in $\mathcal{I}^Q$ that is an input owned by $p$ exists in $U$ in the form $x^C$. Replace all such $x$ with the matching $x^C$ form in $\mathcal{I}^Q$.

   (d) Update $F$ to $F \vee (B \wedge (a'^C_S \vee \neg\mathcal{I}^P) \wedge (a^C_S \vee \neg\mathcal{I}^Q))$.

   (e) Update $B$ to $B \wedge \neg a'^C_S \wedge \mathcal{I}^P \wedge \neg a^C_S \wedge \mathcal{I}^Q$.

9. Let $S'$ be the secure round that follows $S$. If $S$ is the final secure round, then set $S' = \emptyset$.

10. If $O_p$ contains boolean variables, then let $X$ be the set of assignments to all boolean variables in $O_p$. For each $X'$ in $X$

    (a) Let $C' = C \cup X'$.

    (b) Call **create_formula"**$(U, E, F, B, C', n, S')$

11. Otherwise, call **create_formula"**$(U, E, F, B, C, n, S')$