Using SMT Solvers to Automate Design Tasks for Encryption and Signature Schemes

Joseph A. Akinyele[*]
Johns Hopkins University
Baltimore, MD, USA
akinyelj@cs.jhu.edu

Matthew Green[†]
Johns Hopkins University
Baltimore, MD, USA
mgreen@cs.jhu.edu

Susan Hohenberger[‡]
Johns Hopkins University
Baltimore, MD, USA
susan@cs.jhu.edu

ABSTRACT

Cryptographic design tasks are primarily performed by hand today. Shifting more of this burden to computers could make the design process faster, more accurate and less expensive. In this work, we investigate tools for programmatically altering existing cryptographic constructions to reflect particular design goals. Our techniques enhance both security and efficiency with the assistance of advanced tools including Satisfiability Modulo Theories (SMT) solvers.

Specifically, we propose two complementary tools, AutoGroup and AutoStrong. AutoGroup converts a pairing-based encryption or signature scheme written in (simple) symmetric group notation into a specific instantiation in the more efficient, asymmetric setting. Some existing symmetric schemes have hundreds of possible asymmetric translations, and this tool allows the user to optimize the construction according to a variety of metrics, such as ciphertext size, key size or computation time. The AutoStrong tool focuses on the security of digital signature schemes by automatically converting an existentially unforgeable signature scheme into a *strongly* unforgeable one. The main technical challenge here is to automate the "partitioned" check, which allows a highly-efficient transformation.

These tools integrate with and complement the AutoBatch tool (ACM CCS 2012), but also push forward on the complexity of the automation tasks by harnessing the power of SMT solvers. Our experiments demonstrate that the two design tasks studied can be performed automatically in a matter of seconds.

[*]Joseph A. Akinyele's work was supported in part by National Science Foundation (NSF) grant CNS-1154035.

[†]Matthew Green's work was supported in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211 and the Office of Naval Research (ONR) contract N00014-11-1-0470.

[‡]Susan Hohenberger's work was supported in part by NSF CNS-1154035 and CNS-1228443; the DARPA and the AFRL under contract FA8750-11-2-0211, DARPA N11AP20006, the ONR under contract N00014-11-1-0470, and a Microsoft Faculty Fellowship. Applying to all authors, the views expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defense, the NSF, or the U.S. Government.

CCS'13, November 4–8, 2013, Berlin, Gernany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516718.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Cryptographic controls, Authentication, Access controls, Verification

Keywords

Digital Signatures, Public-Key Encryption, Pairing-Based Cryptography, Automation, Cryptographic Compilers

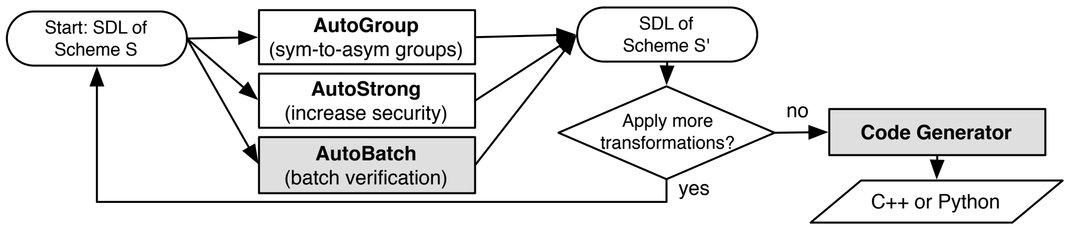
1. INTRODUCTION

Cryptographic design is challenging, time consuming and mostly performed by hand. A natural question to ask is: to what extent can computers ease this burden? Which common design tasks can computers execute faster, more accurately or less expensively?

In particular, this work investigates tools for programmatically altering existing cryptographic constructions in order to enhance efficiency or security design goals. For instance, digital signatures, which are critical for authenticating data in a variety of settings, ranging from sensor networks to software updates, come in many possible variations based on efficiency, functionality or security. Unfortunately, it is often infeasible or tedious for humans to document each possible optimal variation for each application. It would be enormously valuable if there could be a small number of simple ways to present a scheme – as simple as possible to avoid human-error in the design and/or verification process – and then computers could securely provide any variation that may be required by practitioners.

A simple, motivating example (which we explore in this work) is the design of pairing-based signature schemes, which are often presented in a simple "symmetric" group setting that aids in exposition, but does not map to the specific pairing-based groups that maximize efficiency. Addressing this disconnect is ripe for an automated tool.

Summary of Our Contributions In this work, we explore two novel types of design problems for pairing-based cryptographic schemes. The first tool (AutoGroup) deals with efficiency, while the second (AutoStrong) deals with security. We illustrate how they interact in Figure 1. The tools take a Scheme Description Language (SDL) representation of a scheme (and optionally some user optimization



Start: SDL of Scheme S
AutoGroup (sym-to-asym groups)
AutoStrong (increase security)
AutoBatch (batch verification)
SDL of Scheme S'
Apply more transformations?
no
yes
Code Generator
C++ or Python

Figure 1: A high-level presentation of the new automated tools, AutoGroup and AutoStrong. They take as input a Scheme Description Language (SDL) representation of a cryptographic scheme and output an SDL representation of a transformation of the scheme, which can possibly be further transformed by another tool. These tools are compatible with the existing AutoBatch tool and Code Generator (shaded). An SDL input to the Code Generator produces a software implementation of the scheme in either C++ or Python.

constraints) and output an SDL representation of the altered scheme. This SDL output can be run through another tool or a Code Generator to produce C++ or Python software.

A contribution of this work is that we integrated our tools with the publicly-available source code for AutoBatch [3,4] of Akinyele et al. (ACM CCS 2012), a tool that automatically identifies a batch verification algorithm for a given signature scheme, therein weaving together a larger automation system. For instance, a practitioner could take any symmetric-pairing signature scheme from the literature, use AutoGroup to reduce its bandwidth in the asymmetric setting, use AutoBatch to reduce its verification time, and then automatically obtain a C++ implementation of the optimized construction. Our work appears unique in that we apply advanced tools, such as SMT solvers and Mathematica, to perform complex design tasks related to pairing-based schemes.

Automated Task 1: Optimize Efficiency of an Encryption or Signature Scheme via User Constraints. Pairings are often studied because they can realize new functionalities, e.g., [17, 19], or offer low-bandwidth solutions, e.g., [17, 21]. Pairing (a.k.a., bilinear) groups consist of three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with an efficient bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Many protocols are presented in a *symmetric* setting where $\mathbb{G}_1 = \mathbb{G}_2$ (or equivalently, there exists an efficient isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ or vice versa).

While symmetric groups simplify the description of new cryptographic schemes, the corresponding groups are rarely the most efficient setting for implementation [32]. The state of the art is to use *asymmetric* groups where $\mathbb{G}_1 \neq \mathbb{G}_2$ and no efficient isomorphism exists between the two. See for instance the work of Ramanna, Chatterjee and Sarkar [50] (PKC 2012) which translates the dual system encryption scheme of Waters [57] from the symmetric to a handful of asymmetric settings.

Such conversions currently require manual analysis (of all steps) – made difficult by the fact that certain operations such as group hash functions only operate in a single group. Moreover, in some cases, there are hundreds of possible symmetric to asymmetric translations, making it tedious to identify the optimal translation for a particular application.

We propose a tool called AutoGroup that automatically provides a "basic" translation from symmetric to asymmetric groups.[1] It employs an SMT solver to identify valid group assignments for all group elements and also accepts user constraints to optimize the efficiency of the scheme according to a variety of metrics, including signature/ciphertext size, signing/encryption time, and public parameter size. The tool is able to enumerate the full set of possible solutions (which may run to the hundreds), and can rapidly identify the most efficient solution.

Automated Task 2: Strengthen the Security of a Digital Signature Scheme. Most signature schemes are presented under the classic, existential unforgeability definition [35], wherein an adversary cannot produce a signature on a "new" message. However, *strong* unforgeability guarantees more – that the adversary cannot produce a "new" signature even on a previously signed message. Strongly-unforgeable signatures are often used as a building block in signcryption [6], chosen-ciphertext secure encryption [25,28] and group signatures [7,18].

There are a number of general transformations from classic to strong security [14, 15, 37, 53–55], but also a highly-efficient transformation due to Boneh, Shen and Waters [22] that only applies to "partitioned" schemes. We propose a tool called AutoStrong that automatically decides whether a scheme is "partitioned" and then applies BSW if it is and a general transformation otherwise. The partitioned test is non-trivial, and our tool harnesses the power of both an SMT solver and Mathematica to make this determination. We are careful to err only on false negatives (which impact efficiency), but not false positives (which could compromise security.) Earlier works [14,15] claimed that there were "very few" examples of partitioned schemes; however, our tool proved this was not the case by identifying valid partitions for most schemes we tested.

1.1 Related Work

Many exciting works have studied how to automate various cryptographic tasks. Automation has been introduced into the design process for various security protocols [38, 40, 49, 52], optimizations to software implementations involving elliptic-curves [10] and bilinear-map functions [48], the batch verification of digital signature schemes [4], secure two-party computation [36,41,42], and zero-knowledge proofs [5,8,9,23,43].

[1]By "basic", we mean that it translates the scheme as written into the asymmetric setting, with minor optimizations performed, but does not attempt a re-imagining of the construction based on a stronger asymmetric complexity assumption. While the latter is sometimes possible, e.g., [50], it may not be required in some applications and the novel security analysis required places it beyond the current ability of our automation tools. See Section 3.3 for more.

Our current work is most closely related to the AutoBatch tool of Akinyele et al. [4]. We borrow our tool-naming system from their paper and designed our tools so that they can integrate with the publicly-available source code of AutoBatch [3] to form a larger, more comprehensive solution. This work is different from AutoBatch in that it attacks new, more complicated design tasks and integrates external SMT solvers and Mathematica to find its solutions.

Prior work on automating the writing and verification of cryptographic proofs, such as the EasyCrypt work of Barthe et al. [13], are complimentary to but distinct from our effort. Their goal was automating the construction and verification of (game-based) cryptographic *proofs*. Our goal is automating the construction of cryptographic *schemes*. A system that combines both to automate the design of a scheme and then automate its security analysis would be optimal.

2. TOOLS USED

Our automations make use of three external tools. First, Z3 [26, 46] is a freely-available, state-of-the-art and highly efficient Satisfiability Modulo Theories (SMT) solver produced by Microsoft Research. SMT is a generalization of boolean satisfiability (SAT) solving, which determines whether assignments exist for boolean variables in a given logical formula that evaluates the formula to *true*. SMT solvers builds on SAT to support many rich first-order theories such as equality reasoning, arithmetic, and arrays. In practice, SMT solvers have been used to solve a number of constraint-satisfaction problems and are receiving increased attention in applications such as software verification, program analysis, and testing. Z3 in particular has been used as a core building block in API design tools such as Spec#/Boogie [11, 27] and in verifying C compilers such as VCC.

We leverage Z3 v4.3.1 to perform reasoning over statements involving arithmetic, quantifiers, and uninterpreted functions. We use Z3's theories for equality reasoning combined with the decision procedures for linear arithmetic expressions and elimination of universal quantifiers (*e.g.,* $\forall x$) over linear arithmetic. Z3 includes support for uninterpreted (or *free*) functions which allow any interpretation consistent with the constraints over free functions and variables.

Second, we utilize the development platform provided by Wolfram Research's Mathematica [59] (version 9), which allows us to simplify equations for several of our analytical techniques. We leverage Mathematica in our automation to validate that given cryptographic algorithms have certain mathematical properties. Finally, we utilize some of the publicly-available source code of the AutoBatch tool [3], including its Scheme Description Language (SDL) parser and its Code Generator, which translates an SDL representation to C++ or Python.

3. AUTOGROUP

In this section, we present and evaluate a tool, called AutoGroup, for automatically altering a cryptographic scheme's algebraic setting to optimize for efficiency.

3.1 Background on Pairing Groups

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be algebraic groups of prime order p.[2] We say that $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a pairing (a.k.a., bilinear map) if it is: efficiently-computable, (*bilinear*) for all $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$ and $a, b \leftarrow \mathbb{Z}_p$, $e(g^a, h^b) = e(g,h)^{ab}$; and (*non-degenerate*) if g generates $\mathbb{G}_1$ and h generates $\mathbb{G}_2$, then $e(g,h) \neq 1$. This is called the *asymmetric* setting. A specialized case is the *symmetric* setting, where $\mathbb{G}_1 = \mathbb{G}_2$.[3]

In practice, all efficient candidate constructions for pairing groups are constructed such that $\mathbb{G}_1$ and $\mathbb{G}_2$ are groups of points on some elliptic curve E, and $\mathbb{G}_T$ is a subgroup of a multiplicative group over a related finite field. The group of points on E defined over $\mathbb{F}_p$ is written as $E(\mathbb{F}_p)$. Usually $\mathbb{G}_1$ is a subgroup of $E(\mathbb{F}_p)$, $\mathbb{G}_2$ is a subgroup of $E(\mathbb{F}_{p^k})$ where k is the embedding degree, and $\mathbb{G}_T$ is a subgroup of $\mathbb{F}^*_{p^k}$. In the symmetric case $\mathbb{G}_1 = \mathbb{G}_2$ is usually a subgroup of $E(\mathbb{F}_p)$.

The challenge in selecting pairing groups is to identify parameters such that the size of $\mathbb{G}_T$ provides acceptable security against the MOV attack [44]. Hence the size of p^k must be comparable to that of an RSA modulus to provide the same level of security – hence elements of $\mathbb{F}_{p^k}$ must be of size approximately 3,072 bits to provide security at the 128-bit symmetric equivalent level. The group order q must also be large enough to resist the Pollard-ρ attack on discrete logarithms, which means in this example $q \geq 256$.

Two common candidates for implementing pairing-based constructions are supersingular curves [31, 47] in which the embedding degree k is ≤ 6 and typically smaller (an example is $|p| = 1536$ for the 128-bit security level at $k = 2$), or ordinary curves such as MNT or Barreto-Naehrig (BN) [12]. In BN curves in particular, the embedding degree $k = 12$, thus $|p| = |q|$ can be as small as 256 bits at the 128-bit security level, with a corresponding speedup in field operations.

A challenge is that the recommended BN subgroups do not possess an efficiently-computable isomorphism from $\mathbb{G}_1$ to $\mathbb{G}_2$ or vice versa, which necessitates re-design of some symmetric cryptographic protocols. A related issue is that BN curves permit efficient hashing only into the group $\mathbb{G}_1$. This places restrictions on the set of valid group assignments we can use.

3.2 How AutoGroup Works

AutoGroup is a new tool for automatically translating a pairing-based encryption or signature scheme from the symmetric-pairing setting to the asymmetric-pairing setting. At a high-level, AutoGroup takes as input a representation of a cryptographic protocol (*e.g.,* signature or encryption scheme) written in a Domain-Specific Language called Scheme Description Language (SDL), along with a description of the optimizations desired by the user. These optimizations may describe a variety of factors, e.g., requests to minimize computational cost, key size, or ciphertext / signature size. The tool outputs a new SDL representation of the scheme, one that comprises the optimal assignment of groups for the given constraints. The assignment of groups is non-trivial, as many schemes are additionally constrained by features of common asymmetric bilinear groups settings, most notably, restrictions on which groups admit efficient hashing. At a high level, AutoGroup works by reducing this constrained group assignment problem to a boolean satisfiability problem, applying an SMT solver, and processing the results. We next describe the steps of AutoGroup, as illustrated in Figure 2.

[2]Pairing groups may also have composite order, but we will be focusing on the more efficient prime order setting here.

[3]An alternative instantiation of the symmetric setting has $\mathbb{G}_1 \neq \mathbb{G}_2$ but admits an efficiently-computable isomorphism between the groups.

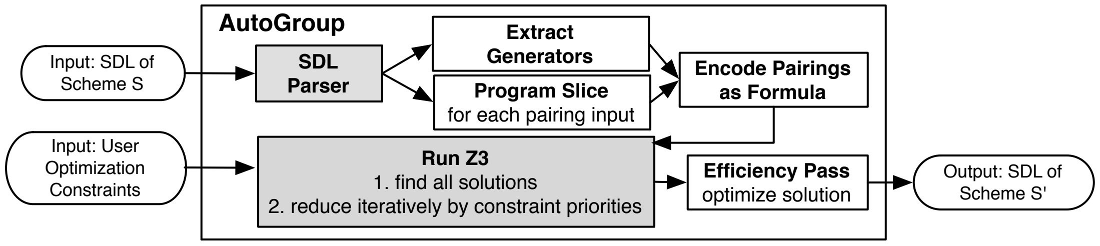

AutoGroup
Input: SDL of Scheme S
SDL Parser
Extract Generators
Program Slice for each pairing input
Encode Pairings as Formula
Input: User Optimization Constraints
Run Z3
1. find all solutions
2. reduce iteratively by constraint priorities
Efficiency Pass optimize solution
Output: SDL of Scheme S'

Figure 2: A high-level presentation of the AutoGroup tool, which uses external tools Z3 and SDL Parser.

1. Extract Generator Representation. The first stage of the AutoGroup process involves parsing SDL to identify all base generators of $\mathbb{G}$ that are used in the scheme. For each generator $g \in \mathbb{G}$, AutoGroup creates a pair of generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. This causes an increase in the parameter size of the scheme, something that we must address in later steps.

We assume the Parser knows the basic structure of the scheme, and can identify the algorithm responsible for parameter generation. This allows us to parse the algorithm to observe which generators that are created. When AutoGroup detects the first generator, it marks this as the "base" generator of $\mathbb{G}$ and splits g into a pair $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. Every subsequent group element sampled by the scheme is defined in terms of the base generators. For example, if the setup algorithm next calls for "choosing a random generator h in $\mathbb{G}$", then AutoGroup will select a random $t' \in \mathbb{Z}_p$ and compute new elements $h_1 = g_1^{t'}$ and $h_2 = g_2^{t'}$.

2. Traceback Inputs to the Pairing Function. Recall that the pairing function $e(A, B)$ takes two inputs. We extract all the pairings required in the scheme; these might come from the setup algorithm, encryption/signing, or decryption/verification. Prior to tracing the pairing inputs, we split pairings of the form $e(g, A \cdot B)$ as $e(g, A) \cdot e(g, B)$ to prepare for encoding pairings as logical formulas in the SMT solver. In the final step of AutoGroup we recombine the pairings to preserve efficiency. We reuse techniques introduced in [4, 29] to split and combine pairings in AutoGroup.

After splitting applicable pairings, we obtain a program slice for each variable input to determine which (symmetric) generators were involved in computing it. This also helps us later track which variables are affected when an assignment for a given variable is made in $\mathbb{G}_1$ or $\mathbb{G}_2$. Consider the example $A = X \cdot Y$. Clearly, the group assignment of A affects variables X and Y, and capturing the slice for each pairing input variable is crucial for AutoGroup to perform correct re-assignment for the subset of affected variables.

3. Convert Pairings to Logical Formulas. Asymmetric pairings require that one input to the function be in $\mathbb{G}_1$, and the other be in $\mathbb{G}_2$. Conversion from a symmetric to an asymmetric pairing can be reduced to a constraint satisfiability problem; we model the asymmetric pairing as an inequality operator over binary variables. This is analogous because an inequality constraint enforces that the binary variables either have a 0 or 1 value, but not both for the equation to be satisfiable. Therefore, we express symmetric pairings as a logical formula of inequality operators over binary variables separated by conjunctive connectors (*e.g.,* $A \neq B \wedge C \neq D$). We then employ an SMT solver to find a satisfiable solution and apply the solver's solution to produce an equivalent scheme in the asymmetric setting.

4. Convert Pairing Limitations into Constraints. When translating from the symmetric to the asymmetric pairing setting, we encounter several limitations that must be incorporated into our model. Chief among these are limitations on hashing: in some asymmetric groups, hashing to $\mathbb{G}_2$ is not possible. In other groups, there is no such isomorphism, but it is possible to hash into $\mathbb{G}_1$. Depending on the groups that the user selects, we must identify an asymmetric solution that respects these constraints. Fortunately these constraints can easily be expressed in our formulae, by simply assigning the output of hash functions to a specific group, e.g., $\mathbb{G}_1$.

5. Execute SMT Solver. We run the logical formula plus constraints through an SMT solver to identify a satisfying assignment of variables. The solver checks for a satisfiable solution and produces a model of 0 (or $\mathbb{G}_1$) and 1 (or $\mathbb{G}_2$) values for the pairing input variables that satisfies the specified constraints. We can go one step further and enumerate all the unique solutions (or models) found by the solver for a given formula and constraints. After obtaining all the possible models, we utilize the solver to evaluate each model and determine the solutions that satisfies the user's application-specific requirements.

6. Satisfy Application-specific Requirements. To facilitate optimizations in the asymmetric setting that suit user applications, we allow users to specify additional constraints on the chosen solution. There are two possible ways of tuning AutoGroup: one set of options focus on reducing the size of certain scheme outputs. For public key encryption, the user can choose to minimize the representation of the secret keys, ciphertext or both. Similarly, for signatures schemes, the user can optimize for minimal-sized public keys, signatures or both. The second set of options focus on reducing algorithm execution times. This is possible due to the fact that for many candidate asymmetric groups, group operations in $\mathbb{G}_1$ are dramatically more efficient than those that take place in $\mathbb{G}_2$. Users may also combine various operations, in order to find an optimal solution based on a combination of size and operation time.

We find application-specific solutions by minimizing an objective function over all the possible models obtained from the solver. Our objective function is straightforward and calculated as follows:

$$F(A, C, w_1, w_2) = \sum_{i=1}^{n} ((1 - a_i) \cdot w_1 + a_i \cdot w_2) \cdot c_i$$

where $A = a_i, \dots, a_n$ and represents the pairing input variables, w_1 and w_2 denote *weights* over groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, $C = c_i, \dots, c_n$ and each c_i corresponds to the *cost* for each a_i. Each input variable a_i can have a value of $0 = \mathbb{G}_1$ or $1 = \mathbb{G}_2$. We now describe how the above options are converted into parameters of F and discuss how the SMT solver is used to obtain a minimal solution.

For each parameter that we intend to optimize, we define a *weight function* that evaluates each candidate solution according to some metric. For each assigned variable, the weight function calculates the total "cost" of the construction as a function of some cost value for the specific variable, as well as an overall cost for an assignment of $\mathbb{G}_1$ and $\mathbb{G}_2$. In the case of ciphertext size we assign the cost value to 1 for each group element that appears in the ciphertext, and 0 for all others. For encryption time, we assign a cost that corresponds to the number of group operations applied to this variable during the encryption operation. The overall cost value then determines the cost of placing a value in one of the two groups – for size-related calculations, this roughly corresponds to the length of a group element's representation, and for operation time it corresponds to the cost of a single group operation. By assigning these costs correctly, we are able to create a series of different weight functions that represent all of the different values that we would like to minimize (e.g., ciphertext size, parameter size, time).

If the user chooses to optimize for multiple criteria simultaneously, we must find a model that balances between all of these at the same time. This is not always possible. For example, some schemes admit solutions that favor a minimized secret key size or ciphertext size, but not both. In this case, we allow the user to determine which constraint to relax and thereby select the next best solution that satisfies their requirements.

7. Evaluate and Process the Solution. Once the application-specific solution is obtained from the solver, the next step is to apply the solution to produce an asymmetric scheme. As indicated earlier, we interpret the solution for each variable as $0 = \mathbb{G}_1$ and $1 = \mathbb{G}_2$. To apply the solution, we first pre-process each algorithm in SDL to determine how the pairing inputs are affected by each assignment. Consider a simplistic example: $e(A, B)$ where $A = g^a$ and $B = h^b$. Let us assume that the satisfying solution is that $A \in \mathbb{G}_1$ and $B \in \mathbb{G}_2$. Therefore, we would rewrite these two variables as $A = g_1^a$ and $B = h_2^b$ where $g_1 \in \mathbb{G}_1$ and $h_2 \in \mathbb{G}_2$. The program slice recorded for each pairing input in step (2) provides the necessary information to correctly rewrite the scheme in the asymmetric setting.

In addition to rewriting the scheme, AutoGroup performs several final optimizations. First, it removes any unused parameter values in the public and secret keys. For signature schemes, we try to optimize further by reducing the public parameters used per algorithm. In particular, we trace which variables in the public key are actually used during signing and verification. For elements that appear only in the signing (resp. decryption) algorithms, we split the public key into two: one is kept just for computing signatures (resp. decryption), and the other is given out for use in encryption/verification. Second, AutoGroup performs an additional efficiency check and attempts to optimize pairing product equations to use as few pairings as possible. This is due to the decoupling of pairings in earlier phases of translating the scheme to the asymmetric setting or perhaps, just a loose design by the original SDL designer. In either case, we apply pairing optimization techniques from previous work [4, 29] to provide this automatic efficiency check. Finally, AutoGroup outputs a new SDL of the modified scheme.

We do not offer the efficiency check of AutoGroup as a standalone tool for symmetric groups at present, because our experience inclines us to believe that most practitioners concerned with efficiency will want to work in asymmetric groups. However, our results herein also demonstrate that a simple tool of this sort is efficient and feasible.

3.3 Security Analysis of AutoGroup

Whether a scheme is translated by hand (as is done today [50]) or automatically (as in this work), a completely separate question applying to both is: is the resulting asymmetric scheme secure? The answer is not immediately clear. Unlike the signature transformation that we automate in Section 4 that already has an established security proofs showing that the transformations preserve security, the theoretical underpinnings of symmetric-to-asymmetric translations are less explored. Here are some things we can say.

First, the original proof of security is under a symmetric pairing assumption, and thus can no longer immediately apply since the construction and assumption are changing their algebraic settings. This would seem to require the identification of a new complexity assumption together with a new proof of security. In many examples, e.g., [21], the new assumption and proof are only minor deviations from the original ones, e.g., where the CDH assumption in $\mathbb{G}$ (given $[g, g^a, g^b]$, compute g^{ab}) is converted in a straight-forward manner to the co-CDH assumption in $(\mathbb{G}_1, \mathbb{G}_2)$ (given $[g_1, g_2, g_2^a]$, compute g_1^a). However, there could be cases where a major change is required to the proof of security. For instance, in some asymmetric groups it is not possible to hash into $\mathbb{G}_2$, but in these groups there exists an isomorphism from $\mathbb{G}_2$ to $\mathbb{G}_1$. In other groups there is no such isomorphism, but it is possible to hash into $\mathbb{G}_2$. So if a scheme requires both for the security proof, that scheme may not be realizable in the asymmetric setting (see [32] for more).

In best practices today, a human first devises the new construction (based on their desired optimizations) and then the human works to identify the new assumption and proof. Our current work automates the first step in this process, and hopefully gives the human more time to spend on the second step. In this sense, our automation is arguably faster, and no less secure than what is done by hand today.

However, a more satisfactory solution requires a deeper theoretical study of symmetric-to-asymmetric pairing translations, which we feel is an important open problem, but which falls outside the scope of the current work. What can one prove about the preservation of security in symmetric-to-asymmetric translations? Is it necessary to dig into the proof of security? Or could one prove security of the asymmetric scheme solely on the assumption of security of the symmetric one? Will this work the same for encryption, signatures and other protocols? Do the rules by which translations are done (by hand or AutoGroup) need to change based on these findings? These questions remain open.

3.4 Experimental Evaluation of AutoGroup

To determine the effectiveness of our automation, we evaluate several encryption and signature schemes on a variety of optimization combinations supported by our tool. We sum-

Encryption	Time			Approx. Size		Num.
	Keygen•	Encrypt•	Decrypt•	Secret Key	Ciphertext	Solutions
ID-Based Enc.						
BB04 [16, §4] Symmetric (SS1536)	59.9 ms	64.8 ms	125.4 ms	3072 bits	6144 bits	
Asymmetric (BN256) [Min. CT]	4.8 ms	7.8 ms	27.6 ms	2048 bits	3584 bits	4
Gentry06 [33, §3.1] Symmetric (SS1536)	39.9 ms	176.2 ms	67.8 ms	3072 bits	7680 bits	
Asymmetric (BN256) [Min. SK]	1.4 ms	41.0 ms	19.1 ms	512 bits	7168 bits	4
WATERS09 [57, §3.1] Symmetric (SS1536)	294.6 ms	286.8 ms	612.8 ms	13824 bits	18432 bits	
Asymmetric (BN256) [Min. SK/CT/Exp]	12.6 ms	19.2 ms	128.0 ms	5376 bits	8704 bits	256
Broadcast Encryption						
BGW05 [20, §3.1] Symmetric (SS1536) ($n = 100$)	1992.2 ms	119.6 ms	136.9 ms	19200 bytes	6144 bits	
Asymmetric (BN256) [Min. SK]	70.4 ms	25.7 ms	28.5 ms	3200 bytes	5120 bits	4

•Average time measured over 100 test runs and the standard deviation in all test runs were within ±1% of the average.

Figure 3: AutoGroup on encryption schemes under various optimization options. We show running times and sizes for several schemes generated in C++ and compare symmetric to automatically generated asymmetric implementations at the same security levels (roughly equivalent with 3072 bit RSA). For IBE schemes, we measured with the identity string length at 100 bytes. For BGW, n denotes the number of users in the system.

marize the results of our experiments on encryption schemes in Figure 3 and signature schemes in Figure 5.

System Configuration. All of our benchmarks were executed on a 2.66GHz 6-core Intel Xeon Mac Pro with 10GB RAM running Mac OS X 10.8.3 using only a single core of the Intel processor. Our implementation utilizes the MIRACL library (v5.5.4), Charm v0.43 [2] in C++ due to the efficiency gains over Python code, and Z3 SMT solver (v4.3.1). We based our implementations on the MIRACL library to fully compare each scheme's performance using symmetric and asymmetric curves at equivalent security levels.

Results. To demonstrate the soundness of AutoGroup on encryption and signature schemes, we compare algorithm running times, key and ciphertext/signature sizes between symmetric and asymmetric solutions. We tested AutoGroup on a variety of optimization combinations to extract different asymmetric solutions. In each test case, AutoGroup reports all the unique solutions, obtains the best solution for given user-specified constraints, and generates the executable code of the solution in a reasonable amount of time. AutoGroup execution time on each test case is reported in Figure 6, but does not include time for generating the C++ of the SDL output.

4. AUTOSTRONG

In this section, we present and evaluate a tool, called AutoStrong, for automatically generating a strongly-unforgeable signature from an unforgeable signature scheme.

4.1 Background on Digital Signatures

A digital signature scheme is comprised of three algorithms: key generation, signing and verification. The classic (or "regular") security definition for signatures, as formulated by Goldwasser, Micali and Rivest [35], is called *existential unforgeability with respect to chosen message attacks*, wherein any p.p.t. adversary, given a public key and the ability to adaptively ask for a signature on any message of its choosing, should not be able to output a signature/message pair that passes the verification equation and yet where the message is "new" (was not queried for a signature), with non-negligible probability.

An, Dodis and Rabin [6] formulated *strong unforgeability* where the adversary should not only be unable to generate a signature on a "new" message, but also be unable to generate a different signature for an already signed message. Strongly-unforgeable signatures have many applications including building signcryption [6], chosen-ciphertext secure encryption systems [25, 28] and group signatures [7, 18].

Partitioned Signatures In 2006, Boneh, Shen and Waters [22] connected these two security notions, by providing a general transformation that converts any *partitioned* (defined below) existentially unforgeable signature into a strongly unforgeable one.

DEFINITION 4.1 (PARTITIONED SIGNATURE [22]). *A signature scheme is* partitioned *if it satisfies two properties for all key pairs (pk, sk):*

- **Property 1:** *The signing algorithm can be broken into two deterministic algorithms F_1 and F_2 so that a signature on a message m using secret key sk is computed as follows:*
  1. *Select a random r from a suitable randomness space.*
  2. *Set $\sigma_1 = F_1(m, r, sk)$ and $\sigma_2 = F_2(r, sk)$.*
  3. *Output the signature (σ_1, σ_2).*
- **Property 2:** *Given m and σ_2, there is at most one σ_1 such that (σ_1, σ_2) verifies as a valid signature on m under pk.*

As one example of a partitioned scheme, Boneh et al. partition DSS [45] as follows, where x is the secret key:

$$F_1(m, r, x) = r^{-1}(m + xF_2(r, x)) \mod q$$
$$F_2(r, x) = (g^r \mod p) \mod q$$

Our empirical evidence shows that many discrete-log and pairing-based signatures in the literature are partitioned. Interestingly, some prominent prior works [14, 15] claimed that there were "few" examples of partitioned schemes "beyond Waters [56]", even though our automation discovered several examples existing prior to the publication of these works. We conjecture that it is not always easy for a human to detect a partition.

Chameleon Hashes The BSW transform uses a *chameleon hash* [39] function, which is characterized by the nonstandard property of being collision-resistant for the signer but collision tractable for the recipient. The chameleon hash

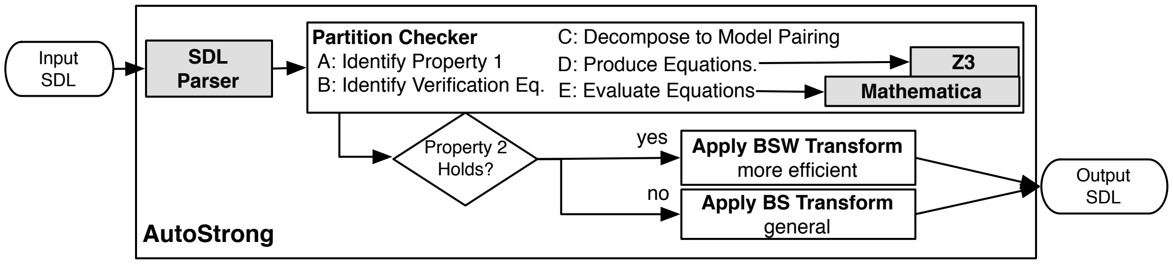


Figure 4: A high-level presentation of the AutoStrong tool, which uses external tools Z3, Mathematica and SDL Parser.

is created by establishing public parameters and a secret trapdoor. The hash itself takes as input a message m and an auxiliary value s. There is an efficient algorithm that on input the trapdoor, any pair (m_1, s_1) and any additional message m_2, finds a value s_2 such that $\text{ChamHash}(m_1, s_1) = \text{ChamHash}(m_2, s_2)$.

Boneh et al. [22] employ a specific hash function based on the hardness of finding discrete logarithms.[4] Since pairing groups also require the DL problem to be hard, this chameleon hash does not add any new complexity assumptions. It works as follows in $\mathbb{G}$, where g generates $\mathbb{G}$ of order p. To setup, choose a random trapdoor $t \in \mathbb{Z}_p{}^*$ and compute $h = g^t$. The public parameters include the description of $\mathbb{G}$ together with g and h. The trapdoor t is kept secret. To hash on input $(m, s) \in \mathbb{Z}_p{}^2$, compute

$$\text{ChamHash}(m, s) = g^m h^s.$$

Later, given any pair m, s and any message m', anyone with the trapdoor can compute a consistent value $s' \in \mathbb{Z}_p$ as

$$s' = (m - m')/t + s$$

such that $\text{ChamHash}(m, s) = \text{ChamHash}(m', s')$.

The BSW Transformation The transformation [22] is efficient and works as follows. Let $\Pi_p = (\mathsf{Gen}_p, \mathsf{Sign}_p, \mathsf{Verify}_p)$ be a partitioned signature, where the signing algorithm is partitioned using functions F_1 and F_2. Suppose the randomness for Sign_p is picked from some set R. Let $||$ denote concatenation. BSW constructs a new scheme Π as:

$\mathsf{Gen}(1^\lambda)$**:** Select a group $\mathbb{G}$ with generator g of prime order p (with λ bits). Select a random $t \in \mathbb{Z}_p$ and compute $h = g^t$. Select a collision-resistant hash function $H_{cr} : \{0,1\}^* \to \mathbb{Z}_p$. Run $\mathsf{Gen}_p(1^\lambda)$ to obtain a key pair (pk_p, sk_p). Set the keys for the new system as $pk = (pk_p, H_{cr}, \mathbb{G}, g, h, p)$ and $sk = (pk, sk_p, t)$.

$\mathsf{Sign}(sk, m)$**:** A signature on m is generated as follows:

1. Select a random $s \in \mathbb{Z}_p$ and a random $r \in R$.
2. Set $\sigma_2 = F_2(r, sk_p)$.
3. Compute $v = H_{cr}(m||\sigma_2)$.
4. Compute the chameleon hash $m' = g^v h^s$.
5. Compute $\sigma_1 = F_1(m', r, sk_p)$ and output the signature $\sigma = (\sigma_1, \sigma_2, s)$.

[4]Indeed, we observe that substituting an arbitrary chameleon hash could break the transformation. Suppose $H(m, s)$ ignores the last bit of s (it is easy to construct such a hash assuming chameleon hashes exist.) Then the BSW transformation using this hash would result in a signature of the form (σ_1, σ_2, s), which is clearly not strongly unforgeable, since the last bit can be flipped.

$\mathsf{Verify}(pk, m, \sigma)$**:** A signature $\sigma = (\sigma_1, \sigma_2, s)$ on a message m is verified as follows:

1. Compute $v = H_{cr}(m||\sigma_2)$.
2. Compute the chameleon hash $m' = g^v h^s$.
3. Output the result of $\mathsf{Verify}_p(pk_p, m', (\sigma_1, \sigma_2))$.

Theorem 4.2 (Security of BSW Transform [22]). *The signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *is strongly existentially unforgeable assuming the underlying scheme* $\Pi_p = (\mathsf{Gen}_p, \mathsf{Sign}_p, \mathsf{Verify}_p)$ *is existentially unforgeable,* H_{cr} *is a collision-resistant hash function and the discrete logarithm assumption holds in* $\mathbb{G}$.

The Bellare-Shoup Transformation The BSW transformation [22], which only works for partitioned signatures, sparked significant research interest into finding a general transformation for *any* existentially unforgeable signature scheme. Various solutions were presented in [14, 15, 37, 53–55], as well as an observation in [14] that an *inefficient* transformation was implicit in [34].

We follow the work of Bellare and Shoup [14,15], which is less efficient than BSW and, for our case, requires a stronger complexity assumption, but works on any signature. Their approach uses *two-tier* signatures, which are "weaker" than regular signatures as hybrids of regular and one-time schemes. In a two-tier scheme, a signer has a primary key pair and, each time it wants to sign, it generates a fresh secondary key pair and produces a signature as a function of the both secret keys and the message. Both public keys are required to verify the signature. Bellare and Shoup transform any regular signature scheme by signing the signature from this scheme with a strongly unforgeable two-tier scheme. They also show how to realize a strongly unforgeable two-tier signature scheme by applying the Fiat-Shamir [30] transformation to the Schnorr identification protocol [51], which requires a one-more discrete logarithm-type assumption.

The BS transformation works as follows. Let $\Pi_r = (\mathsf{Gen}_r, \mathsf{Sign}_r, \mathsf{Verify}_r)$ be a regular signature scheme and let $\Pi_t = (\mathsf{PGen}_t, \mathsf{SGen}_t, \mathsf{Sign}_t, \mathsf{Verify}_t)$ be a two-tiered strongly unforgeable scheme. A new signature scheme Π is constructed as:

$\mathsf{Gen}(1^\lambda)$**:** Run $\mathsf{Gen}_r(1^\lambda) \to (pk_r, sk_r)$ and $\mathsf{PGen}_t(1^\lambda) \to (ppk, psk)$. Output the pair $\text{PK} = (pk_r, ppk)$ and $\text{SK} = (sk_r, psk)$.

$\mathsf{Sign}(\text{SK}, m)$**:** A signature on m is generated as follows:

1. Parse SK as (sk_r, psk).
2. Run $\mathsf{SGen}_t(1^\lambda) \to (spk, ssk)$.
3. Sign the message and secondary key as $\sigma_1 \leftarrow \mathsf{Sign}_r(sk_r, (spk||m))$.

4. Sign the first signature as $\sigma_2 \leftarrow \mathsf{Sign}_t(psk, ssk, \sigma_1)$.
5. Output the signature $\sigma = (\sigma_1, \sigma_2, spk)$.

$\mathsf{Verify}(\mathrm{PK}, m, \sigma)$**:** A signature $\sigma = (\sigma_1, \sigma_2, spk)$ on a message m is verified as follows:

1. Parse PK as (pk_r, ppk).
2. If $\mathsf{Verify}_r(pk_r, (spk\|m), \sigma_1) = 0$, then return 0.
3. If $\mathsf{Verify}_t(ppk, spk, \sigma_1, \sigma_2)$, then return 0.
4. Otherwise, return 1.

THEOREM 4.3 (SECURITY OF BS TRANSFORMATION [15]). *If the input scheme is existentially unforgeable, then the output signature is strongly existentially unforgeable assuming the strong unforgeability of the two-tier scheme.*

The Transformation used in AutoStrong For our purposes, we employ the following hybrid transformation combining BSW and Bellare-Shoup. On input a signature scheme, we automate the following procedure:

1. Identify a natural partition satisfying property 1 and test if it has property 2. (We allow false negatives, but not false positives. See Section 4.3.)
2. If a valid partition is found, apply the BSW transformation [22] (using SHA-256 and the DL-based chameleon hash above).
3. If a valid partition is not found, apply the Bellare-Shoup transformation [14,15] (using the Schnorr Fiat-Shamir based two-tier scheme suggested in [15].)
4. Output the result.

The security of this transformation follows directly from the results of [15,22] as stated in Theorems 4.2 and 4.3. *The most challenging technical part is step one: determining if a scheme is partitioned.*

4.2 How AutoStrong Works

AutoStrong takes as input the SDL description of a digital signature scheme along with some metadata.[5] At a high-level, it runs the transformation described at the end of the last section, where the most challenging step is testing whether a scheme is *partitioned* according to Definition 4.1.

We now describe each step involved in testing that Properties 1 and 2 are satisfied and how we utilize Z3 and Mathematica to prove such properties, as illustrated in Figure 4.

Identify Property 1. The first goal is to identify the variables in the signature that should be mapped to σ_1 or σ_2 according to Definition 4.1. We assume that the input signature scheme is existentially unforgeable.[6] Given this assumption, our objective is to identify the portions of the signature that are computed based on the message and designate that component as σ_1. All other variables in the signature that do not meet this criteria are designated as σ_2. We determine that we have designated the correct variables for property 1 if and only if the variable mapping satisfy property 2. We test only the most "natural" division for property 1, which could result in a false negative, but this won't impact the security, so our system allows it.

To illustrate each step, we will show how our tool identifies the partition in the CL signature scheme [24].

CL signatures [24]: Key generation consists of selecting a generator, $g \in \mathbb{G}$, then randomly sampling $x \in \mathbb{Z}_q$ and $y \in \mathbb{Z}_q$. It sets $sk = (x, y)$ and $pk = (g, X = g^x, Y = g^y)$. To sign a message $m \in \mathbb{Z}_q$, the signer samples a uniformly from $\mathbb{G}$ and computes the signature as:

$$\sigma = (a, b = a^y, c = a^{x+m\cdot x\cdot y}).$$

The verifier can check σ by ensuring that $e(a, Y) = e(g, b)$ and $e(X, a) \cdot e(X, b)^m = e(g, c)$.

Intuitively, our logic would identify that c is dependent on the message, therefore, identifying that $\sigma_1 = c$ and $\sigma_2 = (a, b)$ which satisfies the definition of property 1. The next challenge is to determine whether property 2 holds given our identified mapping for σ_1 and σ_2.

Prove Property 2. Proving that a scheme satisfies this property requires the ability to abstractly evaluate the verification equations on the input variables. We require this ability to automatically prove that there exists at most one σ_1 which verifies under a fixed σ_2, m and pk for all possible inputs. To this end, the partition checker determines whether a σ_1' exists such that $\sigma_1' \neq \sigma_1$ and is a valid signature over the fixed variables. Finding such a σ_1' means the signature is not partitioned. The checker determines whether it can find a solution or if it can determine that *no* such solution exists. If no solutions exist, then the signature is indeed partitioned. Stated more precisely, does there exist a $\sigma_1' \neq \sigma_1$ such that the following condition holds:

$$\mathsf{Verify}(pk, m, (\sigma_1, \sigma_2)) = 1 \wedge \mathsf{Verify}(pk, m, (\sigma_1', \sigma_2)) = 1$$

At a high-level, our goal is to evaluate the pairing-based verification algorithms in a way that allows us to find a contradiction to the aforementioned condition. Recall that the bilinearity property of pairings states that $e(g^a, g^b) = e(g, g)^{ab}$ holds for all $a, b \in \mathbb{Z}_q$ where $g \in \mathbb{G}$. We observe that pairings can be modeled as an abstract function that performs multiplication in the exponent. Because the rules of multiplication and addition hold in the exponent, we can abstractly reduce pairings to basic integer arithmetic.

To accomplish this, we leverage Z3 to model the bilinearity of pairings so that it is possible to automatically evaluate them. Our partition checker relies on Z3's uninterpreted functions and universal quantifiers to reduce pairing product equations to simpler equations over the exponents. However, this reduction alone is not sufficient to completely evaluate the verification equations as required for detecting a partitioned signature. To satisfy the property 2 condition, we also need a way to evaluate these equations on all possible inputs. Z3 was less suited for this task and instead, we employ the Mathematica scripting framework to evaluate such equations. Our solution consists of five steps:

Step 1: Decompose Verification Equations. To model pairings using an SMT solver, we encode the verification equations into a form that the solver can interpret. The first phase extracts the verification equations in SDL, then decomposes the equations in terms of the generators and exponents used. We leverage recent term rewriting extensions introduced in the SDL Parser by Akinyele et al. [4].

[5]The user must specify the variables that denote message, signature, key material in a configuration file.

[6]We remark that we tested the partition checker for AutoStrong on schemes that are not existentially unforgeable to fully vet the checker (see Figure 5), but the resulting output in these cases may not be strongly unforgeable.

Signature	Security	Time		Approx. Size		Num.
		Sign[•]	Verify[•]	Public Key[*]	Signature	Solutions
CL04 [24, §3.1] Symmetric (SS1536)	EU-CMA	169.8 ms	316.6 ms	3072 bits	4608 bits	
Symmetric (SS1536)	SU-CMA	192.0 ms	387.8 ms	4608 bits	6144 bits	
Asymmetric (BN256) [Min. SIG]	SU-CMA	3.4 ms	56.8 ms	2048 bits	1024 bits	2
BB Short [17, §3] Symmetric (SS1536)	EU-CMA	21.5 ms	102.1 ms	7680 bits	3072 bits	
Symmetric (SS1536)	SU-CMA	62.8 ms	142.8 ms	9216 bits	4608 bits	
Asymmetric (BN256) [Min. PK]	SU-CMA	5.0 ms	18.3 ms	3840 bits	1536 bits	2
WATERS05 [56, §4] Symmetric (SS1536)	EU-CMA	47.9 ms	195.2 ms	4608 bits[†]	3072 bits	
Symmetric (SS1536)	SU-CMA	88.7 ms	236.4 ms	6144 bits[†]	4608 bits	
Asymmetric (BN256) [Min. SIG]	SU-CMA	6.5 ms	62.9 ms	2560 bits[†]	768 bits	8
WATERS09 [58, §6.1] Symmetric (SS1536)	WU-CMA	258.5 ms	896.8 ms	23040 bits	13824 bits	
Asymmetric (BN256) [Min. PK/SIG]	WU-CMA	13.6 ms	129.2 ms	12544 bits	5376 bits	256
ACDKNO12 [1, §5.3] Symmetric (SS1536)	RMA	346.4 ms	1307 ms	23040 bits	12288 bits	
Asymmetric (BN256) [Min. PK/SIG/Exp]	RMA	23.3 ms	279.9 ms	3840 bits	8192 bits	1024

[•]Average time measured over 100 test runs and the standard deviation in all test runs were within $\pm1\%$ of the average.
[*]Refers to the approximate size of public parameters used in verification.
[†]Estimates do not include the public parameters for the Water's hash.

Figure 5: We show the result of AutoGroup and AutoStrong on signature schemes. For CL, BB, and Waters (with length of identities, $\ell = 128$), we first apply AutoStrong to determine that the signature scheme is partitioned, then apply the BSW transform to obtain a strongly unforgeable signature in the symmetric setting. We then feed this as input to AutoGroup to realize an asymmetric variant under a given optimization. We also tested AutoStrong on the DSE signature and ACDK structure-preserving signature, even though these are not known to be existentially unforgeable. A partition was found for ACDK, but not DSE.

Their techniques allow us to keep track of how variables are computed in terms of the generators and exponents. With knowledge of how each variable is computed, we are able to fully decompose each equation in an automated fashion.

Our technique for modeling pairings in Z3 requires that decomposition of verification equations be guided by a few rules. First, generators must be rewritten in terms of some base generator, g, if the scheme is specified in the symmetric setting.[7] For example, the random generator $a \in \mathbb{G}$ chosen in CL would be represented as $g^{a'}$ for $a' \in \mathbb{Z}_q$. Second, hashing statements of the form $v = H(m)$ where $v \in \mathbb{G}$ are rewritten as $g^{v'}$ for some $v' \in \mathbb{Z}_q$.[8] Third, we do not decompose any variable designated as σ_1 for the purposes of determining whether a signature is partitioned. The intuition is that since σ_1' variables are adversarially controlled we also treat σ_1 as a black box. Finally, whenever we encounter signatures that compute a product over a list of elements – as in the case of the Waters hash, for example [56] – we require the user to provide an upper bound on the number of elements in this list (if known) so that we can "unroll" the product calculation and further apply our rules. When all the above reduction rules are automatically applied to the CL signature, we obtain the following equations:

$e(a, Y) = e(g, b)$ becomes $e(g^{a'}, g^y) = e(g, (g^{a'})^y)$

$e(X, a) \cdot e(X, b)^m = e(g, c)$ becomes
$$e(g^x, g^{a'}) \cdot e(g^x, (g^{a'})^y)^m = e(g, g^{c'})$$

Note that c' denotes the σ_1 for CL and is a free variable. All other variables that comprise m, pk, and σ_2 are fixed.

Step 2: Encode Rules for Evaluating Pairings. Once we have decomposed the verification equation as shown above, the next step is to encode the equations in terms that Z3 can understand. After the pairing equations are rewritten entirely using the base generator, we can model the behavior of pairings by simply focusing on the exponents. To capture the bilinearity of pairings, we rely on two features in Z3: uninterpreted functions and universal quantifiers. As mentioned earlier, uninterpreted functions enable one to abstractly model a function's behavior. Our model of a pairing is an uninterpreted function, E, that takes two integer variables and has a few mathematical properties. First, we define the multiplication rule as $\forall s, t : E(s, t) = s \cdot t$. Second, we define the addition rule as $\forall s, t, u : E(s+t, u) = s\cdot u + t\cdot u$.[9] Third, we adhere to the multiplicative notation in SDL and convert pairing products defined in terms of multiplication to addition and division to subtraction.

These rules are straightforward and sufficient for evaluating pairings. Moreover, by defining exponents in terms of integers, Z3 can apply all the built-in simplification rules for multiplication and addition. As a result, the solver uses these rules to reduce any pairing-based verification equation into a simpler integer equation.

To automatically encode the equations, we first simplify the decomposed pairing equation as much as possible using previous techniques [4]. Then, we convert each pairing to the modeled pairing function, E and remove the base generators. Upon simplifying and encoding the decomposed CL equations, we obtain the following:

$e(g^{a'}, g^y) = e(g, (g^{a'})^y)$ becomes $E(a', y) = E(1, a' \cdot y)$

$e(g^x, g^{a'}) \cdot e(g^x, (g^{a'})^y)^m = e(g, g^{c'})$ becomes
$$E(x, a') + E(x \cdot m, a' \cdot y) = E(1, c')$$

Step 3: Execute SMT Solver. After encoding the pairing functions in terms of E, the next step is to employ the solver to evaluate it. We first specify our rules in the SMT solver then evaluate these rules on each input equation. The result is a simplified integer equation representation of the verification algorithm. For the above CL formulas, the solver determines that the first equation is *true* for all possible in-

[7]The same would apply for asymmetric pairings except that we would specify $\mathbb{G}_1$ generators in terms of a base generator g_1 and $\mathbb{G}_2$ in terms of g_2.

[8]Note that this term re-writing is used only to determine whether a solution exists. The actual variables a' and v' would not (necessarily) be known in the real protocol.

[9]Similarly, $E(s, t + u) = s \cdot t + s \cdot u$

puts because a' and y are fixed variables. For the second equation, the solver produces: $a' \cdot x + a' \cdot x \cdot m \cdot y = c'$.

Step 4: Evaluate equations. At this point, we have obtained the integer equation version of the verification equation; we can now concretely express the conditions for property 2. That is,

$$c' \neq c'' \wedge a' \cdot x + a' \cdot x \cdot m \cdot y = c' \wedge a' \cdot x + a' \cdot x \cdot m \cdot y = c''$$

We use Mathematica to prove that no such c'' exists assuming the verification condition is correct via the Mathematica Script API. In particular, we utilize the *FindInstance* function to mathematically find proof over non-zero real numbers then subsequently try finding a solution over integers. If *no* such solution exists, the *FindInstance* will return such a statement and the result is interpreted as an indicator that the signature is partitionable. Otherwise, the signature may not be partitionable.

During this step, we make an explicit assumption that the verification condition is mathematically correct. Suppose that this was not the case. In this scenario, our technique would also determine that it is not possible to find a σ_1' such that $\sigma_1' \neq \sigma_1$ and verifies over fixed variables. In reality, however, no σ_1 and σ_2 pair can produce a valid signature because the verification equation does not hold for *any* input. To limit the possibility of such scenarios, our partition checker offers a sanity check on the correctness of the input verification equations.

By relaxing the rule for decomposing the variables that are designated as σ_1 in Step 1, we can evaluate the verification equation over all inputs using Mathematica. For the CL signature, a full decomposition would produce the following equation in the exponent:

$$a' \cdot x + a' \cdot x \cdot m \cdot y = a' \cdot (x + x \cdot m \cdot y)$$

It is sufficient to leverage the *Simplify* function within Mathematica to evaluate that this holds for all possible inputs. Since Mathematica has built-in techniques for solving equations of this sort, it becomes trivial to show that the above equation is correct in all cases (due to the law of distribution). We subsequently inform the user on the output of this sanity check, which is useful for determining the correctness of SDL signature descriptions.

Step 5: Apply Transformation. Once the partition checker determines whether the signature is partitioned or not, we apply the efficient BSW transform if deemed partitioned or the less-efficient BS transform if not as described in Section 4.1. We elaborate further in the full version.

4.3 Security Analysis of AutoStrong

The theoretical security of the unforgeable-to-strongly-unforgeable transformations that we use in AutoStrong were previously established in [14, 15, 22], as discussed in Section 4.1.[10] The security of the BSW transform only holds, however, if the input scheme is partitioned. Our partition test allows false negatives, but not false positives. That is, our algorithm may fail to identify a scheme as partitioned even though it is, which results in a less efficient final scheme, but it will not falsely identify a scheme as partitioned when it is not, which would result in a security failure. To see why this claim holds, consider that the partition tester guesses a partition, Z3 interprets the verification equation as a system of equations, and then Mathematica fixes the variables on one partition side and asks how many solutions there are for the free variables on the other side. If 0 or 1 are found, then the scheme meets the partitioned definition. If more than 1 is found, then it is not partitioned. If there is no answer (program crash or times out), then we consider it not partitioned. Thus, false negatives can occur, but not false positives (in theory). Proving that there are no software or hardware errors in AutoStrong, Z3, Mathematica or the underlying software and hardware on which they run is outside the scope of this work. We did experimentally verify AutoStrong's outputs and no errors were found.

[10]Perfect correctness is assumed in these transformations. All schemes tested have perfect correctness, except the Waters DSE signatures [57]. With a negligible probability, the verification algorithm of this scheme will reject an honestly-generated signature. After applying the BS transformation to the DSE scheme, this negligible error probability is carried over in the verification of the strongly-secure scheme.

4.4 Experimental Evaluation of AutoStrong

In 2008 [15], Bellare and Shoup remarked that "unfortunately, there seem to be hardly any [partitioned signature] schemes". Interestingly, our experimental results show that there are in fact many partitioned schemes, including a substantial number invented prior to 2008. We evaluated AutoStrong by testing it on a collection of signatures, including Camenisch-Lysyanskaya [24], short Boneh-Boyen [17], Waters 2005 [56], Waters Dual-System (DSE) signature [57], and a structure-preserving scheme of Abe *et al.* [1].

Of the above signatures, all but one – the Waters DSE signature – were successfully partitioned. We do not know whether the Waters DSE signature can be partitioned, although we suspect that the "randomness freedom" in the dual-system structure may inherently be at odds with the uniqueness property of the partitioned test. Although the Abe *et al.* scheme is partitioned, applying either the BSW or BS transformations destroys its structure-preserving property. An interesting open problem would be to refine the BSW or BS transformations to preserve the structured property. Figure 6 shows the time that it took our tool to identify the partitioning and output the revised signature equations. Figure 5 illustrates the performance and size of the resulting signatures, when evaluated on two different types of curve (using AutoGroup to calculate the group assignments).

5. CONCLUSION

We explored two new tasks in cryptographic automation. First, we presented a tool, AutoGroup, for automatically translating a symmetric pairing scheme into an asymmetric pairing scheme. The tool allows the user to choose from a variety of different optimization options. Second, we presented a tool, AutoStrong, for automatically altering a digital signature scheme to achieve *strong unforgeability* [6]. The tool automatically tests whether a scheme is "partitioned" according to a notion of Boneh et al. [22] and then applies a highly-efficient transformation if it is partitioned or a more general transformation otherwise. To perform some of these complex tasks, we integrated Microsoft's Z3 SMT Solver and Mathematica into our tools. Our performance measurements indicated that these standard cryptographic design tasks can be quickly, accurately and cost-effectively performed in an automated fashion. We leave open the ques-

Process	BB-IBE	Gentry	Waters09-Enc	BGW	CL	BB Short Sig	Waters05	Waters09-Sig	ACDKNO
AutoGroup	0.33s	0.34s	4.30s	0.55s	0.34s	0.31s	0.54s	4.16s	17.65s
AutoStrong	-	-	-	-	0.28s	0.27s	0.37s	3.99s	1.23s

Figure 6: Running time required by the AutoGroup and AutoStrong routines to process the schemes discussed in this work (averaged over 10 test runs). The running time for AutoGroup includes the execution time of the Z3 SMT solver. The running time for AutoStrong also includes Z3 and Mathematica and the application of the BSW transformation. In all cases, the standard deviation in the results were within $\pm 3\%$ of the average. For AutoGroup, running times are correlated with the number of unique solutions found and the minimization of the weighted function using Z3. AutoStrong running times are highly correlated with the complexity of the verification equations.

tion of which other design tasks are well suited for SMT solvers.

Acknowledgments

The authors thank Brent Waters for technical discussions involving the BSW transformation and the Waters DSE signatures. The authors are also grateful to Avi Rubin and Matthew Pagano for helpful discussions.

6. REFERENCES

[1] Masayuki Abe, Melissa Chase, Bernardo David, Markulf Kohlweiss, Ryo Nishimaki, and Miyako Ohkubo. Constant-size structure-preserving signatures: Generic constructions and simple assumptions. Cryptology ePrint Archive, Report 2012/285, 2012. `http://eprint.iacr.org/`.

[2] Joseph A. Akinyele, Christina Garman, Ian Miers, Matthew W. Pagano, Michael Rushanan, Matthew Green, and Aviel D. Rubin. Charm: a framework for rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering*, 3(2):111–128, 2013.

[3] Joseph A. Akinyele, Matthew Green, Susan Hohenberger, and Matthew W. Pagano. AutoBatch Toolkit. `https://github.com/jhuisi/auto-tools`.

[4] Joseph A. Akinyele, Matthew Green, Susan Hohenberger, and Matthew W. Pagano. Machine-generated algorithms, proofs and software for the batch verification of digital signature schemes. In *ACM CCS*, pages 474–487, 2012.

[5] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols. In *ESORICS'10*, pages 151–167, 2010.

[6] Jee Hea An, Yevgeniy Dodis, and Tal Rabin. On the security of joint signature and encryption. In *EUROCRYPT*, volume 2332, pages 83–107, 2002.

[7] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO '00*, volume 1880, pages 255–270, 2000.

[8] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *IEEE Symposium on Security and Privacy*, pages 202–215, 2008.

[9] Endre Bangerter, Thomas Briner, Wilko Henecka, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. Automatic generation of sigma-protocols. In *EuroPKI'09*, pages 67–82, 2009.

[10] M. Barbosa, A. Moss, and D. Page. Compiler assisted elliptic curve cryptography. In *OTM Conferences (2)*, pages 1785–1802, 2007.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. pages 49–69. Springer, 2004.

[12] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *SAC*, volume 3897, pages 319–331, 2006. `http://cryptojedi.org/papers/\#pfcpo`.

[13] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO*, pages 71–90, 2011.

[14] Mihir Bellare and Sarah Shoup. Two-tier signatures, strongly unforgeable signatures, and fiat-shamir without random oracles. In *PKC*, pages 201–216, 2007.

[15] Mihir Bellare and Sarah Shoup. Two-tier signatures from the fiat-shamir transform, with applications to strongly unforgeable and one-time signatures. *IET Information Security*, 2(2):47–63, 2008.

[16] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In Christian Cachin and JanL. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin Heidelberg, 2004.

[17] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *EUROCRYPT*, volume 3027, pages 382–400, 2004.

[18] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO '04*, volume 3152 of LNCS, pages 45–55, 2004.

[19] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, pages 213–229, 2001.

[20] Dan Boneh, Craig Gentry, and Brent Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO'05*, pages 258–275, 2005.

[21] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *ASIACRYPT*, volume 2248 of LNCS, pages 514–532, 2001.

[22] Dan Boneh, Emily Shen, and Brent Waters. Strongly unforgeable signatures based on computational Diffie-Hellman. In *PKC*, pages 229–240, 2006.

[23] J. Camenisch, M. Rohe, and A.R. Sadeghi. Sokrates - a compiler framework for zero- knowledge protocols. In *the Western European Workshop on Research in Cryptology*, WEWoRC, 2005.

[24] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, volume 3152, pages 56–72, 2004.

[25] Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In *EUROCRYPT*, pages 207–222, 2004.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of Software*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[27] Robert DeLine, K. Rustan, and M. Leino. Boogie pl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70.

[28] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM J. Comput.*, 30(2):391–437, 2000.

[29] Anna Lisa Ferrara, Matthew Green, Susan Hohenberger, and Michael Østergaard Pedersen. Practical short signature batch verification. In *CT-RSA*, volume 5473 of LNCS, pages 309–324, 2009.

[30] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[31] Steven D. Galbraith. Supersingular curves in cryptography. In *ASIACRYPT*, pages 495–513, 2001.

[32] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers, 2006. Cryptology ePrint Archive: Report 2006/165.

[33] Craig Gentry. Practical identity-based encryption without random oracles. In *EUROCRYPT*, pages 445–464, 2006.

[34] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[35] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comp.*, 17(2), 1988.

[36] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *ACM CCS*, pages 451–462, 2010.

[37] Qiong Huang, Duncan S. Wong, and Yiming Zhao. Generic transformation to strongly unforgeable signatures. In *ACNS*, pages 1–17, 2007.

[38] Shinsaku Kiyomoto, Haruki Ota, and Toshiaki Tanaka. A security protocol compiler generating C source codes. In *ISA'08*, pages 20–25, 2008.

[39] Hugo Krawczyk and Tal Rabin. Chameleon signatures. In *NDSS*, 2000.

[40] Stefan Lucks, Nico Schmoigl, and Emin Islam Tatli. Issues on designing a cryptographic compiler. In *WEWoRC*, pages 109–122, 2005.

[41] Philip MacKenzie, Alina Oprea, and Michael K. Reiter. Automatic generation of two-party computations. In *ACM CCS*, pages 210–219, 2003.

[42] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay – a secure two-party computation system. In *USENIX Security'04*, pages 287–302, 2004.

[43] Sarah Meiklejohn, C. Chris Erway, Alptekin Küpçü, Theodora Hinkle, and Anna Lysyanskaya. ZKPDL: a language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security'10*, pages 193–206, 2010.

[44] A. Menezes, S. Vanstone, and T. Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field. In *STOC*, pages 80–89, 1991.

[45] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[46] Leonardo Moura and Grant Olney Passmore. The strategy challenge in SMT solving. In *Automated Reasoning and Mathematics*, volume 7788, pages 15–44. 2013.

[47] Dan Page, Nigel Smart, and Fre Vercauteren. A comparison of MNT curves and supersingular curves. *Applicable Algebra in Eng,Com and Comp*, 17(5):379–392, 2006.

[48] Luis J. Dominguez Perez and Michael Scott. Designing a code generator for pairing based cryptographic functions. In *Pairing'10*, pages 207–224, 2010.

[49] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *Advanced Information Networking and Applications*, pages 400–, 2004.

[50] Somindu C. Ramanna, Sanjit Chatterjee, and Palash Sarkar. Variants of Waters' dual system primitives using asymmetric pairings - (extended abstract). In *PKC '12*, pages 298–315, 2012.

[51] Claus-Peter Schnorr. Efficient signature generation by smart cards. *J. Cryptology*, 4(3):161–174, 1991.

[52] Dawn Xiaodong Song, Adrian Perrig, and Doantam Phan. Agvi - automatic generation, verification, and implementation of security protocols. In *Computer Aided Verification*, pages 241–245, 2001.

[53] Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. How to strengthen any weakly unforgeable signature into a strongly unforgeable signature. In *CT-RSA*, pages 357–371, 2007.

[54] Isamu Teranishi, Takuro Oyama, and Wakaha Ogata. General conversion for obtaining strongly existentially unforgeable signatures. In *INDOCRYPT*, pages 191–205, 2006.

[55] Isamu Teranishi, Takuro Oyama, and Wakaha Ogata. General conversion for obtaining strongly existentially unforgeable signatures. *IEICE Transactions*, 91-A(1):94–106, 2008.

[56] Brent Waters. Efficient identity-based encryption without random oracles. In *EUROCRYPT '05*, volume 3494 of LNCS, pages 320–329. Springer, 2005.

[57] Brent Waters. Dual system encryption: Realizing fully secure ibe and hibe under simple assumptions. In *CRYPTO*, pages 619–636, 2009.

[58] Brent Waters. Dual system encryption: Realizing fully secure ibe and hibe under simple assumptions. Cryptology ePrint Archive, Report 2009/385, 2009. `http://eprint.iacr.org/`.

[59] Wolfram. Mathematica, version 9. `http://www.wolfram.com/mathematica/`.