DIVERSITY VIA CODE TRANSFORMATIONS: A SOLUTION FOR NGNA RENEWABLE SECURITY

Yongxin Zhou, Alec Main
Cloakware Inc

Abstract

The Next Generation Network Architecture (NGNA) has been designed to dynamically download and renew secure software components. Such renewability has reduced effectiveness without code diversity, which is the creation of semantically equivalent yet structurally different code components.

A promising solution for diversity is code transformation technology, which provides a large number of highly obfuscated, different representations of the same digital system. Such transformations provide protection of secret information and operations and effectively hide small updates, while preventing automated attacks. This forces attackers into time-consuming manual reverse engineering to break each diverse instance. When components are renewed faster than the time it takes to break an instance, then a truly secure and flexible system has been created.

We outline a mathematical foundation of generating code transformations based upon rich algebra systems, including finite modular ring, finite Galois ring, 2-adic algebra, and Boolean algebra. Then we give some functional properties of Boolean Arithmetic (BA) algebras and show that Mixed Boolean and Arithmetic (MBA) transforms provide efficient and secure methods for data and operation hiding. Lastly, we provide concrete examples of data transforms, operation transforms, and their compositions to generate diversity.

INTRODUCTION

The Need for Renewable and Diverse Software

Software – although often considered the weakest link when creating a secure content protection system – is unavoidable in content protection devices and systems of the future. Ever-more powerful processors and the supply of free, powerful operating systems means that designers will continue to leverage the flexibility of software to future-proof designs and devices while standards change and markets evolve.

The notion that software is insecure is also under debate – especially for network connected devices. IPTV networks have largely chosen lower cost, software-renewable-based security over traditional smart cards. DirecTV, the largest satellite digital broadcaster, is also fighting piracy via software updates. Of course, their system would have been more effective if it had been designed for such countermeasures from the beginning.

Most security experts now accept that a defense in depth is required. You cannot build one, super strong defense and expect it to withstand attack forever. Technology changes and defenses will fail. Similarly, the reality is that neither hardware nor software is immune to being hacked. The more important questions are:

- How do you recover from an attack?
- How do you mitigate the impact of a successful attack?

The simple answer is to renew the security. Since hardware is by nature more costly to replace, this ultimately means renewing the software. Microsoft's Windows Media Digital Rights Management (WMDRM) has remained hack-free for a number of years, largely due to Microsoft's software renewability strategy. While smart cards are also renewable, the cost has proven unacceptable and arguably their form factor has made it easier for pirates to distribute their product.

Downloadable and renewable software is a key component of the Next Generation Network Architecture (NGNA) [1] project and its new Downloadable Conditional Access System (DCAS) [2]. A secure yet generic microprocessor combined with renewable software allows for future flexibility and portability across Multiple Service Operator (MSO) networks, while improving security.

However, the software must also change when it is renewed. This is the basic premise behind software aging to prevent piracy of consumer software [Anck] and DirecTV's approach toward piracy. But the time between updates and the nature of the changes are also critical factors: i) if it takes months between software updates, then the hacker (and his millions of closest friends on the Internet) will have free content until the next update; and ii) if the change to the security is minimal, the hacker will perform a differential analysis on the new software and quickly isolate the changes and re-hack the system.

Software diversity is the creation of semantically equivalent yet structurally diverse software – the key to solving piracy. Software diversity can be deployed across an installed base (spatial diversity) to mitigate the effect of a successful attack to a subset of the users. In addition, software diversity can be deployed over time (temporal diversity). Software obfuscation and diversity is vital to hiding small changes between software releases and preventing differential analysis attacks against the software.

The resistance of a system against attack is a function of both initial resistance and diversity (Figure 1). While well-designed hardware provides higher initial resistance, it typically lacks diversity. Software has lower initial resistance but can easily be diversified and renewed at low cost [Main].

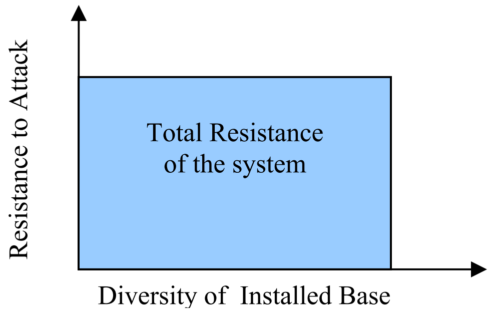

Figure 1. Resistance of a System vs. Diversity

Another effect to consider is that the incentive of a hacker to attack a system (e.g. fame, monetary gain, impact) increases as diversity decreases (Figure 2). This is the corollary of the Microsoft "monoculture" described by some security experts [Greer]. It is also the effect of mass produced hardware security (e.g. smartcards, Xbox and PSP).

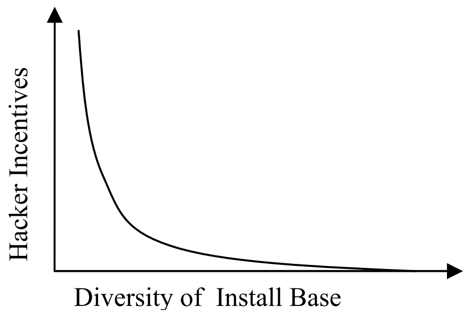

Figure 2. Hacker Incentive

Automated Diversity

Code transformation is an automated approach to generating a large number of highly diverse and obfuscated software instances that is fundamental to any renewable software security.

Based on computational complexity theory, such transformations provide protection of secret information and operations and hide small code changes between updates, while preventing automated attacks via diversity. This forces the attacker into time-consuming manual reverse engineering to break each diverse instance. When components are renewed faster than the time it takes to break the instance, then a truly secure and flexible system has been created [Schn]. Clearly, automated techniques are required to allow for fast updates when necessary.

In this paper, first we build our mathematical foundation for generation of code transformations over existing electrical architectures, which inherit rich algebra systems, including finite modular ring, finite Galois ring, 2-adic algebra, and Boolean algebra.

Then, by selecting the closed algebra systems, we define Boolean Arithmetic (BA) algebras, which applies to real machine computations. We then show that Mixed Boolean Arithmetic (MBA) functions defined over these BA algebra provide an efficient and secure method for data and operation hiding.

Thirdly, we provide concrete examples of code transformations using MBA functions applied to data transforms, operation transforms, and their compositions to obfuscate some common operations, such as addition.

Finally, we briefly discuss general methods of attacking transformed code and counterattacks.

Note that since crypto software components in Conditional Access and Digital Rights Management (DRM) systems are related to integer computations only, we will not discuss floating-point computations. Floating point does not preserve the algebraic structure discussed and more hybrid and practical approaches are required.

THE MATHEMATICAL FOUNDATION

Basic Binary Data Operations in Processors

Since binary code contains all functionality of software and is what is exposed to the hostile environment, it is natural to address software security from this level of code. Almost all processors, both real time and general purpose, have a set of common binary integer instructions. This set includes basic arithmetic operations, bitwise operations, comparison operations, branch operations, and memory access operations.

Arithmetic operations are addition +, subtraction -, multiplication *, and division /. Bitwise operations are and &, or |, exclusive or ^, not ~, bit shift right >> and left <<. Comparison operations include greater than >, greater than or equal >=, less than <, less or equal than <=, not equal !=, and equal ==. In most systems, comparisons are built together with branch operations. Since memory access operations involve no computation, they can simply be regarded as value assignments, which can impact the resulting code, but does not impact the mathematical theory.

Introduction of Boolean Aritmetic Algebra

Software is composed of those basic operations computing on binary data. The most common data format used in all operation systems is two's complement format [Dornhoff].

In this paper, we assume all data has a fixed bit size n. We use B^n to represent all n-bit binary data, where B = {0, 1}. For example, B^{32} is the set of all 32-bit binary values. To simplify the discussion, we assume the address space of the digital system is also n-bit, which is the case of real machines with a 32-bit address space.

We have defined a new term called Boolean Arithmetic (BA) algebra to encapsulate all concepts above. BA algebra is defined on set B^n with arithmetic, bitwise, and comparison operations, thus, the algebra is defined as (B^n, +, -, *, &, |, ^, ~, >>, <<, >, >=, <, <=, ==, !=). For simplicity, hereafter, we use B^n to represent this algebra system.

Digital machines work on BA algebras. That is, a binary program can be regarded as a mathematical function from domain B^n X B^n X … X B^n to co-domain B^n. A large number of diversified code formats can be generated for a given BA algebra function.

Algebra Systems over a BA Algebra

For a BA algebra, B^n, there are several known algebraic systems as its subsystems. Here we mention some of them, which are useful for our code transformations:

(B^n, &, |, ~) is a Boolean algebra structure. It is isomorphic to the one-bit Boolean algebra and has all identities of logical relationships [Dornhoff].

(B^n, ^, &) is a Boolean ring [Dornhoff]. (B^n, +, *) is isomorphic to the modular integer ring $Z/(2^n)$, where Z is the integer ring. Since this is two's complement format, then we define the Abelian group (B^n, +). The modular ring (B^n, +, *) can also be regarded as a Galois ring $GR(2^{s,},2^{st})$, where s and t are integers such that s*t=n [Wan].

(B^n, +, *, /) can be interpreted as a truncation of the 2-adic ring Z_2.

Over the binary data values B^n, we can define a finite field $GF(2^n)$ = (B^n, ^, #) algebra system, where # is the multiplication of the field [Lidl].

Over rings $Z/(2^n)$, $GF(2^n)$ and $GR(2^{s,},2^{st})$, we have polynomial rings $Z/(2^n)[x]$, $GF(2^n)[x]$, and $GR(2^{s,},2^{st})[x]$, as well as multivariate polynomial rings [Jacobson]. More algebra structures can be defined over those rings, such as group rings $Z/(2^n)G$, $GF(2^n)G$ and $GR(2^{s,},2^{st})G$, and loop rings $Z/(2^n)L$, $GF(2^n)L$ and $GR(2^{s,},2^{st})L$, where G is a group and L is a loop [Passman] [Goodaire]. Matrix rings can also be defined over all these rings as can many more algebra systems [Jacobson].

In summary, we demonstrated that over the set B^n, a large number of algebra systems is defined and can be represented by operations in BA algebra. This fact is the theoretical foundation for us to create diversified code. This is a huge area to explore further and apply to software, but in the remainder of this paper, we restrict ourselves to the algebra properties from Boolean (B^n, &, |, ^, ~) and modular ring (B^n, +, *) and use them to create examples of code transformations.

CODE TRANSFORMATIONS

Transforms for Operands and Operators

Software is defined by its operations and orders of the execution of those operations. Code transformations are transforms of operands, called data transforms; and transforms of operators, called operation transforms; and rearrangements of the execution order, most likely related to different basic blocks, called control flow transforms.

This paper focuses on the first two transforms and does not discuss control flow transforms. Specifically, we explore protection and diversity by applying bitwise and arithmetic operations of a BA algebra.

Data Transforms

To generate functionally equivalent code, the requirement of invertibility of a data transform is necessary. Any one-to-one mapping f(x) from B^n to B^n can be used as data transforms. For simplicity, we will not consider multivariate cases, such as matrix transforms, in this paper.

To illustrate data transforms, let's start with linear transforms. Suppose a is an odd number and b is any integer of $Z/(2^n)$. Define a linear transform f(x) = a*x + b. Then f(x) is an invertible data transform with inverse $g(x) = a^{-1}*x + (-a^{-1}*b)$. For instance, for 32-bit words:

f(x) = 674529845*x + 944280100

is a data transform with inverse function

g(x) = 998260765 *x + 490631660

It is easy to verify that f(g(x))=x and gf(x)=x, for any x in $Z/(2^n)$.

We also have invertible quadratic polynomial transforms over $Z/(2^n)$. Let a, b, c be three integers and define function f(x) = $a*x^2 + b*x + c$. f(x) is invertible if a is even and b is odd [Rivest]. To have an invertible quadratic function $g(x) = u*x^2 + v*x + w$ with integer u, v, w, solve equation over $Z/(2^n)$ based on the condition g(f(x))=x. It gives us a formula for u, v and w:

$u = -ab^{-3}$,
$v = 2a*b^{-3}*c+b^{-1}$,
$w = -b^{-1}*c - a*b^{-3}*c^2$,

if we assume the coefficient a of f(x) satisfies the condition $2*a^2 = 0$. For 32-bit words, a concrete example is:

$f(x) = 1502216192*x^2 + 3387143129*x + 1221118466$

with inverse function:

$g(x) = 113639424*x^2 + 841194601*x + 3173903662$.

Again, it is easy to verify f(g(x)) = g(f(x)) = x , for any x in $Z/(2^n)$.

In general, for any invertible polynomial function f(x) of degree n, we can construct its inverse in polynomial format with the given degree n under certain conditions of coefficients of f(x).

Other than polynomial functions, there exist enormous invertible functions with MBA expressions over BA algebra, as shown in [Klimov]. All those transforms can be used as data transforms.

Operation Transforms

Operation transforms replace one operation or multiple operations by some other operations. For example, over BA algebra B^n, addition x + y can be replaced by expression x^y + 2*(x&y), because they are equal over B^n. This is an example of MBA expression, and there are other interesting MBA identities that can also be used, some of which have been used in hardware implementation, such as x^y = (x|y)–(x&y). [Warren]

Such identities are quite useful for operation transforms, and we think they deserve a new definition. Over B^n, identities of the format $\sum a_i*e_i = 0$, where a_i are non-zero constant integers and e_i are bitwise expressions of certain variables, are linear MBA (Mixed Boolean and Arithmetic) identities. Linear MBA identities can be generated as follows.

For a fixed number of variables, if the columns of the truth tables of some Boolean expressions e_i of these variables form a set of linearly dependent vectors over modular ring $Z/(2^n)$ with coefficients a_i, we can construct a linear MBA identity $\sum a_i*e_i = 0$.

For any given singular (0,1)-matrix M, we have a linear MBA identity $\sum a_i*e_i = 0$, where e_i are Boolean expressions whose truth tables are columns of M, and vector a_i is the non-zero solution from the linear system M*X = 0 over $Z/(2^n)$.

Since there exists a large number of singular (0,1)-matrices, the number of linear MBA identities is enormous. Following methods mentioned above, we can also show that any bitwise expressions can be replaced by a nontrivial linear MBA expression from a linear MBA identity.

For two variables x, y from B^n, here we list more linear MBA identities:

x + y = (x^(~y)) + 2*(x|y) + 1;
x | y = x + y + 1 + ((-x-1)|(-y-1));
x & y = ((~x)|y) + x + 1;
x ^ y = x – y - 2*(x |~ y) - 2.

We encourage readers to write code to verify an interesting non-linear multiplication MBA identity:

x*y = (x&y)*(x|y) + (x&~y)*(~x&y).

EXAMPLES OF CODE TRANSFORMATIONS

Compositions for Arithmetic and Bitwise Operations

By using functional compositions of data and operation transforms, a given function can be reformatted into a new one with the same functionality. Because of the large number of diversified data and operation transforms, diversified code of different formats can be created.

In this section, we give examples of transformed code for some basic operations of a BA algebra. These examples are over BA algebra system B^{32} of 32-bit data and operations. They are generated based on the data transform f(x) = a*x + b and some linear MBA identities of two variables. Some random coefficients are assigned for integers a and b which are either split or folded with other constants in the expression. All code can be easily verified using computer programs, while noting that signed and unsigned data types make no difference in binary code.

In the following examples, we assume x and y are two input operands and z is the output operand of the given operation. Other variables are intermediate ones with the same variable size.

ADDITION, z is x + y:

```
t1 = (4211719010 ^ 2937410391 * x) + 2 * (2937410391 * x | 83248285) + 4064867995;

t2 = (2937410391 * x | 3393925841) + 638264265 * y - ((2937410391 * x) & 901041454);

z = 519915623 * t1 - ((3383387769 * t2 + 129219187) ^ 2756971371) – 2 * ((911579527 * t2 + 4165748108) | 2756971371) + 4137204492;
```

SUBTRACTION, z is x - y:

```
t1 = (268586306 ^ (904621911 * x)) + 2 * ((904621911*x) |4026380989) + 3383600763;

t2 = (904621911 * x | 898293889) + 961858761 * y - ((904621911 * x) & 3396673406);

z = 2385439847 * t1 -((673378951 * t2 + 2646655957) ^ 4036393323) -2 * ((3621588345 * t2 + 1648311338) | 4036393323) + 2704000620;
```

OR, z is x | y:

```
t1 = (223550072 ^ 1783698419 * x) + 2 * (1783698419 * x |4071417223) + 865773809;

t2 = (1783698419 * x | 3200160250) + 3498694157*y - ((1783698419 * x) & 1094807045);

z = 1905442107 * t1 -((94202053 * t2 + 1827854967) ^ 1206196916) + ((2389525189 * t1 + 1837740140) | (4200765243 * t2 + 2621826401)) -2 * ((4200765243 * t2 + 2467112328) | 1206196916) + 3508709997;
```

AND, z is x & y:

```
t1 = (543752565 ^ 3040826005 * x) + 2 * (3040826005 *x|3751214730) + 3865950549;

t2 = (3040826005 * x | 3870539833) + 1950617889*y - ((3040826005 * x) & 424427462);

z = ((830084931 * t1 + 2725441253) & (2151370015 * t2 + 843551768)) + 3464882365 * t1 - ((2143597281 * t2 + 1431133401) ^ 2824232692) -2 * ((2151370015 * t2 + 2863833894) | 2824232692) + 2119073563;
```

XOR, z is x ^ y:

```
t1 = (913079019 ^ 864001891 * x) + 2 * (864001891 * x |3381888276) + 1912011113;

t2 = (864001891 * x | 1111987895) + 3067869469 * y - ((864001891 * x) & 3182979400);

z = 4281784907 * t1 - ((722243893 * t2 + 758131618) ^ 2456696338) – 2 * ((3572723403 * t2 + 3536835677) | 2456696338) + 3205073209 + 2 * ((13182389 * t1 + 2473633363) | (3572723403 * t2 + 314825442));
```

Compositions for Comparison Operations

Comparison operations are used as decision makers in the execution path of the code. For example, in a Conditional Access application, it plays an important role for policy checks.

Here we use equal == comparison as an example to show how to use MBA functions to transform the code. We also apply the comparison results into a computation with linear MBA functions such that inequality of the two values will cause malfunction of the original computations.

Suppose we have two variables p1 and p2. In the following example, if p1 equals p2 the output z is x + y, otherwise it has a very high probability that the value of z is not x + y:

```
t1 = (224336580 ^ 1271166401 * x) - ((~p2) | (3572723403*x)) + 2 * (1271166401 * x | 4070630715) + (p1 & (3572723403 * x)) + 3753888605 - p1;

t2 = (1570335793 * p1 | 3302094725) + 1182396209 * y - ((1570335793 * p2) & 992872570);

z = 4131953217 * t1 -((687540689 * t2 + 3822681666) ^ 3209267133) – 2 * ((3607426607 * t2 + 472285629) | 3209267133)+ 56754764;
```

By saying high probability, we recognize that in some special cases (e.g. for x = 0 or -1), then the equation may still compute. These cases can be avoided in practice, or added to create further ambiguity and frustration for the attacker.

Again, interested readers can verify this with a computer program.

Attacks and Counterattacks

In a hostile environment, where the cryptographic keys and code of Conditional Access (CA) systems operate, there exist many possible attacks. Common ones are static and dynamic code analysis, powered with sophisticated debug tools, as well as tampering and emulation type attacks [Oorschot].

Although many attacks exist and more will be developed, we have focused on analysis and tampering attacks designed to either recover secrets or algorithms or change the behavior of the software. These types of attacks (as opposed to emulation type attacks) require reverse engineering (or analysis) of the code.

Reverse engineering attacks fall into two basic types: automated and non-automated. The first one is using general algorithms to attack code, while the second one is the combination of using some algorithms and manual tools to determine the functionality of a given program.

It can been proven that the code recognition problem, that is, to classify different code formats based on their functionalities, is an NP-complete problem. Therefore, it would be difficult, if not impossible, for attackers to find an efficient general algorithm to determine the functionalities of all possible code.

Since creating a general automated attack is difficult, attackers would focus on special properties of the code and tools in an attempt to simplify portions of the code. Taking examples in the previous section that use Boolean arithmetic operations, an attacker could try to use symbolic simplification packages in computer algebra systems, such as commercial products Maple [3], or Mathematica [4]. It is easy to find that those examples cannot be simplified because of the mixture of arithmetic and bitwise operations. These tools would need to be adapted to treat this special situation, but countermeasures need to either apply more algebra structures into the code expression, or to inject more MBA identities into the code. Determining such identities is a hard problem.

Only highly skilled individuals can attempt a manual attack against diversified code generated from code transformations based on different algebraic systems. While they may learn certain techniques when reverse engineering the first instance, the ability to generate specialized helper tools or utilities will be difficult. The variety of algebra systems available is extensive and the composition of such systems result in an extraordinarily high number of combinations. If the only viable attack is to manually reverse engineer the resulting code, then clearly we have achieved our objective.

Furthermore, code obfuscation techniques have been developed to seamlessly inject instances of NP complete problems, such as 3SAT, into transformed code. This makes it impossible to perform manual attacks on individual instances.

CONCLUSION

Rich algebraic structures compatible with digital processors guarantee the existence of a large number of code transforms.

As demonstrated in our examples, code transformations introduce obfuscation and diversity, which are vital for hiding secrets, hiding small code changes and preventing automated analysis which in turn prevents automated attacks against the installed base of devices.

By using Mixed Boolean Arithmetic transformations to data and operators, the resulting diverse code can be renewed and be expected to withstand attack for a reasonable period of time. These transformations are not susceptible to analysis using commercial tools such as Mathematica or Maple.

While the new DCAS system allows for renewable software, this must be diverse to prevent piracy. Hardware does not provide suitable diversity, leaving this role to software. Total system resistance to attack is a function of resistance and diversity. Code transformations provide a practical automated solution for such low-cost, renewable, software security.

ACKNOWLEDGEMENTS

The authors thank Phil Eisen, Song Chen and Heather MacIntosh for their helpful comments and remarks.

END NOTES

[1] Next Generation Network Architecture (NGNA):www.cabledatacomnews.com/ngna/ngnaprimer.html

[2] CableLabs: www.opencable.com/dcas/

[3] Maple: www.maplesoft.com

[4] Mathematica: www.wolfram.com

REFERENCES

[Anck] Anckaert, Bertrand; De Sutter, Bjorn; and De Bosschere, Koen. Software Piracy Prevention through Diversity. In *Proceedings of the 4th ACM Workshop on Digital Rights Management*, 2004.

[Dornhoff] Dornhoff, L.L., Hohn, F.E., Applied Modern Algebra, *Macmillan Publishing Co., Inc.,* 1978.

[Goodaire] Goodaire, E. G., Jespers, E., Polcino Milies, C., Alternative Loop Rings, *North Holland,* 1996.

[Greer] Greer, Dan, et al. Cyber Insecurity: The Cost of Monopoly. September 23, 2003.

[Jacobson] Jacobson, N., Basic Algebra I, *W H Freeman & Co 2nd edition,* 1985.

[Klimov] Klimov, A., Shamir, A., A New Class of Invertible Mappings, in *Cryptographic Hardware and Embedded Systems - CHES 2002, Lecture Notes in Computer Science, Vol 2523, pp 470-483,* 2003.

[Lidl] Lidl, R., Niederreiter, H., Finite Fields, *Second edition, Cambridge University Press*, 1997.

[Main] Main, Alec. Security and savings: Going digital and getting both, NCTA Technical Papers. May 2004.

[Oorschot] Main, A. and van Oorschot, P.C. Software Protection and Application Security: Understanding the Battleground. Springer LNCS. June 2003.

[Passman] Passman, D. S., The Algebraic Structure of Group Rings, *Wiley-Interscience,* 1977.

[Rivest] Rivest, R. L., Permutation Polynomials modulo 2^w, in *Finite Fields and their Applications, Vol7 (2001), pp287-292.*

[Schn] Schneider, F. B., Zhou, L. Implementing Trustworthy Services Using Replicated State Machines, *IEEE Security and Privacy, Vol3 (5), pp 34—43*,2005.

[Wan] Wan, Zhe-Xian, Lectures on Finite Fields and Galois Rings, *World Scientific Pub Co Inc*, 2003.

[Warren] Warren, H. S., Hacker's Delight, *Addison- Wesley*, 2003.

About the Authors

Yongxin Zhou is a senior mathematician at Cloakware. His name is on the author list of two published US patents on software security and 12 research papers on algebra theory. He holds a PhD in Mathematics from Memorial University of Newfoundland, Canada.
(yongxin.zhou@cloakware.com)

Alec Main is Cloakware's Chief Technology Officer. He has published numerous papers and articles on software protection and has spoken at key conferences including The National Show, RSA Conference, Intel Developer Forum, Information Highways, DRM Strategies, Mobile DRM and Certicom-PKCS.
(alec.main@cloakware.com)